

FiPy Manual

Release 2.1

Jonathan E. Guyer
Daniel Wheeler
James A. Warren

Metallurgy Division
and the Center for Theoretical and Computational Materials Science
Materials Science and Engineering Laboratory

April 01, 2010

This software was developed at the [National Institute of Standards and Technology](#) by employees of the Federal Government in the course of their official duties. Pursuant to [title 17 section 105](#) of the United States Code this software is not subject to copyright protection and is in the public domain. FiPy is an experimental system. [NIST](#) assumes no responsibility whatsoever for its use by other parties, and makes no guarantees, expressed or implied, about its quality, reliability, or any other characteristic. We would appreciate acknowledgement if the software is used.

This software 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.

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

Contents

I	Introduction	1
1	Overview	3
1.1	Even if you don't read manuals...	3
1.2	What's new in version 2.1?	3
1.3	Download and Installation	4
1.4	Support	4
1.5	Conventions and Notation	5
1.6	Solving in Parallel	6
2	Installation	7
2.1	Shortcuts	7
2.2	Privileges	8
2.3	Prerequisites	8
2.4	Obtaining FiPy	10
2.5	Testing FiPy	11
2.6	Installing FiPy	12
2.7	Using FiPy	12
2.8	Optional Packages	12
2.9	Platform-Specific Instructions	14
3	Theoretical and Numerical Background	23
3.1	General Conservation Equation	23
3.2	Finite Volume Method	24
3.3	Discretization	24
3.4	Linear Equations	27
3.5	Numerical Schemes	27
4	Design and Implementation	29
4.1	Design	29
4.2	Implementation	31
5	Frequently Asked Questions	33
5.1	How do I represent an equation in FiPy?	33
5.2	How can I see what I'm doing?	37
5.3	Iterations, timesteps, and sweeps? Oh, my!	38
5.4	Why the distinction between <code>CellVariable</code> and <code>FaceVariable</code> coefficients?	40
5.5	How do I represent boundary conditions?	40
5.6	What does this error message mean?	42
5.7	How do I change FiPy's default behavior?	43

5.8	Why don't my scripts work anymore?	44
5.9	What if my question isn't answered here?	44
6	Glossary	45
II	Examples	47
7	Diffusion Examples	51
7.1	examples.diffusion.mesh1D	51
7.2	examples.diffusion.mesh20x20	64
7.3	examples.diffusion.circle	66
7.4	examples.diffusion.electrostatics	71
7.5	examples.diffusion.nthOrder.input4thOrder1D	75
7.6	examples.diffusion.anisotropy	77
8	Convection Examples	79
8.1	examples.convection.exponential1D.mesh1D	79
8.2	examples.convection.exponential1DSource.mesh1D	80
8.3	examples.convection.robin	82
8.4	examples.convection.source	83
9	Phase Field Examples	85
9.1	examples.phase.simple	85
9.2	examples.phase.binary	93
9.3	examples.phase.quaternary	102
9.4	examples.phase.anisotropy	107
9.5	examples.phase.impingement.mesh40x1	111
9.6	examples.phase.impingement.mesh20x20	114
10	Level Set Examples	119
10.1	examples.levelSet.distanceFunction.mesh1D	119
10.2	examples.levelSet.distanceFunction.circle	120
10.3	examples.levelSet.advection.mesh1D	121
10.4	examples.levelSet.advection.circle	122
10.5	Superconformal Electrodeposition Examples	124
10.6	examples.levelSet.electroChem.simpleTrenchSystem	125
10.7	examples.levelSet.electroChem.gold	128
10.8	examples.levelSet.electroChem.leveler	130
10.9	examples.levelSet.electroChem.howToWriteAScript	133
11	Cahn Hilliard Examples	141
11.1	examples.cahnHilliard.mesh2D	141
11.2	examples.cahnHilliard.sphere	142
12	Fluid Flow Examples	147
12.1	examples.flow.stokesCavity	147
13	Updating FiPy	151
13.1	examples.updating.update0_1to1_0	151
13.2	examples.updating.update1_0to2_0	155
III	fiPy Package Documentation	161
14	How to Read the Modules Documentation	163

15 boundaryConditions Package Documentation	165
15.1 The boundaryCondition Module	165
15.2 The fixedFlux Module	165
15.3 The fixedValue Module	166
15.4 The nthOrderBoundaryCondition Module	166
15.5 The test Module	167
16 meshes Package Documentation	169
16.1 common Package Documentation	169
16.2 numMesh Package Documentation	171
16.3 pyMesh Package Documentation	183
16.4 The cylindricalGrid1D Module	187
16.5 The cylindricalGrid2D Module	187
16.6 The grid1D Module	188
16.7 The grid2D Module	188
16.8 The grid3D Module	188
16.9 The test Module	188
17 models Package Documentation	189
17.1 levelSet Package Documentation	189
17.2 The test Module	207
18 solvers Package Documentation	209
18.1 pysparse Package Documentation	209
18.2 trilinos Package Documentation	210
18.3 The solver Module	214
18.4 The test Module	215
19 steppers Package Documentation	217
19.1 The steppers Package	217
19.2 The pidStepper Module	218
19.3 The pseudoRKQSStepper Module	219
19.4 The stepper Module	219
20 terms Package Documentation	221
20.1 The cellTerm Module	221
20.2 The centralDiffConvectionTerm Module	221
20.3 The collectedDiffusionTerm Module	222
20.4 The convectionTerm Module	222
20.5 The diffusionTerm Module	223
20.6 The equation Module	224
20.7 The explicitDiffusionTerm Module	224
20.8 The explicitSourceTerm Module	224
20.9 The explicitUpwindConvectionTerm Module	224
20.10 The exponentialConvectionTerm Module	225
20.11 The faceTerm Module	227
20.12 The hybridConvectionTerm Module	227
20.13 The implicitSourceTerm Module	228
20.14 The mulTerm Module	228
20.15 The nthOrderDiffusionTerm Module	228
20.16 The powerLawConvectionTerm Module	228
20.17 The sourceTerm Module	229
20.18 The term Module	230
20.19 The test Module	232
20.20 The transientTerm Module	232

20.21 The upwindConvectionTerm Module	233
20.22 The vanLeerConvectionTerm Module	234
21 test Module Documentation	237
21.1 The test Module	237
22 tests Package Documentation	239
22.1 The doctestPlus Module	239
22.2 The lateImportTest Module	239
22.3 The testBase Module	239
22.4 The testProgram Module	239
23 tools Package Documentation	241
23.1 dimensions Package Documentation	241
23.2 The tools Package	256
23.3 The debug Module	256
23.4 The dump Module	256
23.5 The inline Module	257
23.6 The memoryLeak Module	257
23.7 The memoryLogger Module	257
23.8 The memoryUsage Module	257
23.9 The numerix Module	257
23.10 The parser Module	266
23.11 The pysparseMatrix Module	267
23.12 The sparseMatrix Module	267
23.13 The test Module	267
23.14 The trilinosMatrix Module	267
23.15 The vector Module	267
23.16 The vitals Module	267
24 variables Package Documentation	269
24.1 The addOverFacesVariable Module	269
24.2 The arithmeticCellToFaceVariable Module	269
24.3 The betaNoiseVariable Module	269
24.4 The binaryOperatorVariable Module	271
24.5 The cellToFaceVariable Module	271
24.6 The cellVariable Module	271
24.7 The cellVolumeAverageVariable Module	275
24.8 The constant Module	275
24.9 The exponentialNoiseVariable Module	275
24.10 The faceGradContributionsVariable Module	276
24.11 The faceGradVariable Module	276
24.12 The faceVariable Module	276
24.13 The fixedBCFaceGradVariable Module	277
24.14 The gammaNoiseVariable Module	277
24.15 The gaussCellGradVariable Module	278
24.16 The gaussianNoiseVariable Module	278
24.17 The harmonicCellToFaceVariable Module	280
24.18 The histogramVariable Module	280
24.19 The leastSquaresCellGradVariable Module	281
24.20 The meshVariable Module	281
24.21 The minmodCellToFaceVariable Module	281
24.22 The modCellGradVariable Module	281
24.23 The modCellToFaceVariable Module	281
24.24 The modFaceGradVariable Module	281

24.25 The <code>modPhysicalField</code> Module	281
24.26 The <code>modularVariable</code> Module	281
24.27 The <code>noiseVariable</code> Module	282
24.28 The <code>operatorVariable</code> Module	283
24.29 The <code>scharfetterGummelFaceVariable</code> Module	283
24.30 The <code>test</code> Module	283
24.31 The <code>unaryOperatorVariable</code> Module	283
24.32 The <code>uniformNoiseVariable</code> Module	283
24.33 The <code>variable</code> Module	284
25 viewers Package Documentation	291
25.1 <code>gistViewer</code> Package Documentation	291
25.2 <code>gnuplotViewer</code> Package Documentation	297
25.3 <code>matplotlibViewer</code> Package Documentation	301
25.4 <code>mayaviViewer</code> Package Documentation	309
25.5 <code>vtkViewer</code> Package Documentation	310
25.6 The <code>viewers</code> Package	312
25.7 The <code>multiViewer</code> Module	313
25.8 The <code>test</code> Module	313
25.9 The <code>testinteractive</code> Module	313
25.10 The <code>tsvViewer</code> Module	314
25.11 The <code>viewer</code> Module	315
Bibliography	317
Module Index	319
Index	323

Part I

Introduction

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 Metallurgy Division and Center for Theoretical and Computational Materials Science (CTCMS), in the Materials Science and Engineering Laboratory (MSEL) 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 [BoettingerReview:2002] [ChenReview:2002] [McFaddenReview:2002] treatments of polycrystalline, dendritic, and electrochemical phase transformations as well as a level set treatment of the electrodeposition process [NIST:damascene:2001].

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

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

...please read *Installation* and the *Frequently Asked Questions*.

1.2 What's new in version 2.1?

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

The significant changes since version 2.0.2 are:

- *Fipy* can use *Trilinos* for solving in parallel.
- We have switched from *Mayavi* 1 to *Mayavi* 2. This *Viewer* is an independent process that allows interaction with the display while a simulation is running.

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

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

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

- 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.

Tickets fixed in this release:

```
171 update the mayavi viewer to use mayavi 2
286 'matplotlib: list index out of range' when no title given, but only sometimes
197 ~binOp doesn't work on branches/version-2_0
194 'easy_install' instructions for MacOSX are broken
192 broken setuptools url with python 2.6
184 The Fipy webpage seems to be broken on Internet Explorer
168 Switch documentation to use ':math:' directive
198 Fipy2.0.2 LinearJORSolver.__init__ calls Solver rather than PysparseSolver
199 'gmshExport.exportAsMesh()' doesn't work
195 broken arithmetic face to cell distance calculations
```

Warning: *Fipy* 2 brought unavoidable syntax changes from *Fipy* 1. Please see `examples.updating.update1_0to2_0` for guidance on the changes that you will need to make to your *Fipy* 1.x scripts. Few, if any, changes should be needed to migrate from *Fipy* 2.0.x to *Fipy* 2.1.

1.3 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.

1.4 Support

You can communicate with the *Fipy* developers and with other users via our mailing list and we welcome you to use the tracking system for bugs, support requests, feature requests and patch submissions <<http://matforge.org/fipy/report>>. We welcome collaborative efforts on this project.

Fipy is a member of MatForge, a project of the Materials Digital Library Pathway. This National Science Foundation funded service provides management of our public source code repository, our bug tracking system, and a “wiki” space for public contributions of code snippets, discussions, and tutorials.

1.4.1 Mailing List

In order to discuss *Fipy* with other users and with the developers, we encourage you to sign up for the mailing list by sending a subscription email:

To: `listproc@nist.gov`

Subject: (*optional*)

Body: `subscribe fipy Your Name`

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

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

List Archive

<http://dir.gmane.org/gmane.comp.python.fipy>

The mailing list archive is hosted by GMANE. Any mail sent to fipy@nist.gov will appear in this publicly available archive.

1.5 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 **tcs**h 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.

1.6 Solving in Parallel

Fipy can use *Trilinos* to solve equations in parallel, as long as they are defined on a “Grid” mesh (`Grid1D`, `CylindricalGrid1D`, `Grid2D`, `CylindricalGrid2D`, or `Grid3D`).

Attention: *Trilinos* must be compiled with MPI support.

Note: A design wart presently *also* requires that *PySparse* be installed. We hope to alleviate this requirement in a future release.

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

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

instead of:

```
$ python myScript.py
```

A complete list of the changes to *Fipy*’s examples needed for parallel can be found at

http://www.matforge.org/fipy/wiki/upgrade2_0examplesTo2_1

Most of the changes were required to ensure that *Fipy* provides the same literal output for both single and multiple processor solutions and are not relevant to most “real” scripts. The two changes you *might* wish to make to your own scripts are:

- It is now preferable to use the `DefaultAssymmetricSolver` instead of the `LinearLUSolver`.
- 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 call `var.getGlobalValue()`.

Chapter 2

Installation

The *FiPy* finite volume PDE solver relies on several third-party packages. It is *best to obtain and install those first*, before attempting to install *FiPy*.

Note: Most of the installation steps will involve a variant on the command:

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

In addition to the specific commands given here, further information about each `setup.py` script is available by typing:

```
$ python setup.py --help
```

For each package, please follow any instructions given in its *README* or *INSTALLATION* files.

2.1 Shortcuts

Detailed prerequisites and links are given below and in platform-specific instructions, but for the courageous and the impatient, *FiPy* can be up and running quickly with one of the following methods.

2.1.1 Enthought Python Distribution

<http://www.enthought.com/epd>

This installer provides a very large number of useful scientific packages for Python, including Python, NumPy, SciPy, Matplotlib, and IPython. Installers are available for [Windows](#), [Mac OS X](#) and [RedHat Linux](#).

Attention: *PySparse* and *FiPy* are not presently included in EPD, so you will need to separately install them manually.

2.1.2 Python(x,y)

<http://www.pythonxy.com/>

Another comprehensive Python package installer for scientific applications, presently only available for [Windows](#). See [Simple Windows Installation](#) for more information.

2.2 Privileges

If you do not have administrative privileges on your computer, or if for any reason you don't want to tamper with your existing Python installation, most packages (including *FiPy*) will allow you to install to an alternate location. Instead of installing these packages with `python setup.py install`, you would use '`python setup.py install --home=dir`', where 'dir' is the desired installation directory (usually “`~`” to indicate your home directory). You will then need to append `dir/lib/python` to your **PYTHONPATH** environment variable. See the [Alternate Installation](#) section of the Python document “[Installing Python Modules](#)” [[InstallingPythonModules](#)] for more information, such as circumstances in which you should use `--prefix` instead of `--home`.

2.3 Prerequisites

2.3.1 Operating System

FiPy is tested regularly on [Mac OS X](#) 10.4 “Tiger” and 10.5 “Leopard”, [Debian Linux](#) 4.0 “etch”, [Ubuntu Linux](#) 10.10, and [Windows XP](#). We welcome reports of compatibility with other systems, particularly if any additional steps are necessary to install.

Note: Simple instructions for [Mac OS X](#) users are in [Simple Mac OS X Installation](#). Simple instructions for [Windows](#) users are in [Simple Windows Installation](#).

The only elements of *FiPy* that are likely to be platform-dependent are the viewers, but at least one viewer should work on each platform. All other aspects should function on any platform that has a recent Python installation.

Many of the packages listed below have prebuilt installers for different platforms (particularly for Windows). These installers can save considerable time and effort compared to configuring and building from source, although they frequently comprise somewhat older versions of the respective code. Whether building from source or using a prebuilt installer, please read and follow explicitly any instructions given in the respective packages’ `README` and `INSTALLATION` files.

2.3.2 Required Packages

Warning: *FiPy* will not run if the following items are not installed.

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 2.3 (#1, Sep 13 2003, 00:49:11)
...
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

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

Note: *FiPy* requires at least version 2.3 of Python. *FiPy* has not yet been tested (and will almost certainly not work) with [Python 3.0](#).

NumPy

<http://sourceforge.net/projects/numpy/>

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

Attention: *FiPy* no longer uses the older *Numeric* or *numarray* packages.

PySparse

<http://pysparse.sourceforge.net>

FiPy requires Roman Geus' *PySparse* package.

You can download the PySparse archive or check it out via anonymous SVN download. From within the `pysparse` base directory, follow its included instructions for building *PySparse* for your platform. PySparse Windows installers are available.

Note: Windows users who choose to build from source should pay particular attention to the instructions in the `INSTALL` file in the base *PySparse* directory.

Warning: If `pysparse` is installed in a local directory a further path may have to be added to the **PYTHONPATH** environment variable. For example, if:

```
$ python setup.py install --home=/some/directory/some/where
```

then both `/some/directory/some/where` and `/some/directory/some/where/lib/python` are required to be added to the **PYTHONPATH**, e.g.:

```
$ set PYTHONPATH=/some/directory/some/where:/some/directory/some/where/lib/python
```

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

Attention: In order to solve in parallel, both *PySparse* and *Trilinos* are required.

2.3.3 Viewers

FiPy will work perfectly well without them, but at least one of the following packages will be required to view the results of *FiPy* calculations. *FiPy* will select the first viewer that is available from the list below. If more than one is installed, specify a viewer by setting the **FIPY_VIEWER** environment variable to either "gist", "gnuplot" or "matplotlib".

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 structured data. It does not display unstructured 2D data or 3D data. It works on all common platforms and produces publication quality hard copies. Version 0.72.1 or higher is required. *Matplotlib* installers for specific platforms are available <http://sourceforge.net/project/showfiles.php?group_id=80706&package_id=82474>.

Note: *Matplotlib* is noticeably slower than *Pygist* or *Gnuplot.py*, but has superior image rendering and plotting functionality.

Gnuplot-py

<http://gnuplot-py.sourceforge.net>

Gnuplot.py is a Python package that interfaces to *gnuplot*, the popular open-source plotting program. It displays both 1D X-Y type plots and 2D contour plots for structured data but not for unstructured data or 3D data. It works on all common platforms and produces hard copies, however, it sometimes breaks on Windows. As a general remark, the viewing quality using either *Pygist* or *Matplotlib* is preferable.

Pygist

<http://hifweb.lbl.gov/public/software/gist/>

The *Pygist* package can be used to display simulation results. It displays both 1D X-Y type plots and 2D contour plots for both structured and unstructured data. It does not display 3D data. Although stated as working on Windows, it does not seem to do a good job of rendering on this platform. *Pygist* works fine on other common platforms. *Pygist* no longer seems to be under development, but is still recommended as a fast light weight alternative to *Matplotlib*.

Attention: *Pygist* requires the old Numeric module to be installed.

Warning: The facility to produce hard copies in *Pygist* does not work very well and may crash the *Fipy* run. “.eps” and “.cgm” export seem to work.

Note: If you experience difficulty building the native *Pygist* viewer on Mac OS X, you may wish to build the package with the `--x11` option described in its documentation.

Note: *Pygist* can have problems finding color pallets, such as “heat.gp” and “work.gs”, when installed locally. You may need to set the **GISTPATH** environment variable to point to the directory containing these files (you may find it as “g/” within the directory you specified for `--home`).

Mayavi

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

The Mayavi 2 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.

Note: Mayavi 2 is packaged for Ubuntu and Debian linux. Using the packaged versions makes installation much easier.

Note: MayaVi 1 is no longer supported.

2.4 Obtaining Fipy

Fipy is freely available for download via Subversion or as a compressed archive from <<http://www.ctcms.nist.gov/fipy/download>>. Please see *SVN usage* for instructions of obtaining *Fipy* with Subversion.

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 Subversion will handle this issue automatically).

2.4.1 Manual

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

```
$ python setup.py build_docs --latex --manual
```

Note: This mechanism is intended primarily for the developers. A command-line pdfTeX installation and several LaTeX packages are required; particularly `memoir.cls`. You will also need to add `~/path/to/fipy/utils` to your `PYTHONPATH` environment variable.

2.5 Testing FiPy

From the base directory, you can verify that *FiPy* works properly by executing:

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

Depending on the packages you chose to install in Optional Packages, be sure to set the appropriate environment variables. You can expect a few errors if you did not install all of the recommended packages.

Note: In order for Python to find the *FiPy* modules, you will need to ensure that the base directory is added to your `PYTHONPATH` environment variable, e.g.:

```
$ setenv PYTHONPATH .:$PYTHONPATH
```

or:

```
$ export PYTHONPATH=.:$PYTHONPATH
```

If you chose to install the `scipy.weave` package, you should rerun the tests with:

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

A few tests will fail the first time as a result of the messages output in the course of caching the compiled inline code, but a repeat test should have no failures (although see “repairing catalog by removing key” in the *Frequently Asked Questions*).

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
```

Note: When running in parallel, there are two expected test failures in `examples.elphf.diffusion.mesh1D` and in `examples.diffusion.nthOrder.input4thOrder-line`. These failures are problems with those particular tests, not with the parallel mechanism itself.

2.6 Installing FiPy

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 and *FiPy* passes its tests, you can install it by typing:

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

at the command line. Alternatively, you may choose not to formally install *FiPy* and to simply work within the base directory instead.

If you choose to install, Python will find your *FiPy* modules automatically. If you choose not to install, then you will need to ensure that the *FiPy* distribution directory is appended to your PYTHONPATH environment variable (either “.” if you are working within the *FiPy* directory, or “~/path/to/fipy” if you are working in your own directory).

2.7 Using FiPy

To see examples of problems that *FiPy* is capable of solving, you can run any of the scripts in examples.

Note: We strongly recommend you proceed through 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 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 \[PythonTutorial\]](#) or with [Dive Into Python \[DiveIntoPython\]](#).

As you gain experience, you will want to see [How do I change FiPy's default behavior?](#) to learn about flags and environment variable that affect *FiPy*.

2.8 Optional Packages

Note: The following packages are not required to run *FiPy*, but they can be helpful.

2.8.1 SciPy

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

Significantly improved performance has been achieved with the judicious use of C language inlining, via the `scipy.weave`. [SciPy download instructions](#) are available <<http://www.scipy.org/Download>>. We recommend version 0.5.2 or greater.

Note: A handful of test cases use functions from the SciPy library and will throw errors if it is missing.

2.8.2 Gmsh

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

Gmsh allows the creation of irregular meshes.

2.8.3 IPython

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

This interactive Python shell is nicer to use than the default, and integrates nicely with matplotlib. Depending on platform, you may be able to download a binary or build from source.

2.8.4 Trilinos

<http://trilinos.sandia.gov>

Trilinos provides solvers and preconditioners, and can be used instead of *PySparse*. *Trilinos* preconditioning allows for iterative solutions to some difficult problems that *PySparse* cannot solve, and it enables solving on parallel nodes.

Attention: In order to solve in parallel, both *PySparse* and *Trilinos* are required.

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 in a basic install of *FiPy*.

Trilinos is built using the standard **configure**, **make** and **make install** method. The best approach that we have found is as follows:

```
$ cd trilinos-X.Y/
$ mkdir BUILD_DIR
$ cd BUILD_DIR
$ ../configure CXXFLAGS="-O3" CFLAGS="-O3" FFLAGS="-O5 -funroll-all-loops \
> -malign-double" --enable-epetra --enable-aztecoo --enable-pytrilinos \
> --enable-ml --enable-ifpack --enable-amesos --with-gnumake --enable-galeri
```

Note: Recif reports that to build on a 64-bit system, the **configure** step needs to be:

```
$ ../configure CXXFLAGS="-O3 -fpic" CFLAGS="-O3 -fpic" \
> FFLAGS="-O5 -fpic -funroll-all-loops" \
> --enable-epetra --enable-aztecoo --enable-pytrilinos \
> --enable-ml --enable-ifpack --enable-amesos --with-gnumake \
> --enable-galeri

$ make $ make install
```

Depending on your platform, other options may be helpful or necessary; see `../configure --help`, the *Trilinos* user guide available from <http://trilinos.sandia.gov/documentation.html>, or <http://trilinos.sandia.gov/packages/pytrilinos/faq.html> for more in-depth documentation.

Note: Trilinos can be installed in a non-standard location by adding the `--prefix=$LOCAL_INSTALLATION_DIR` flag to the `configure` step. If *Trilinos* is installed in a nonstandard location, the path to the PyTrilinos site-packages directory should be added to the **PYTHONPATH** environment variable; this should be of the form `$INSTALL_DIR/lib/$PYTHON_VERSION/site-packages/`. Also, the path

to the *Trilinos* lib directory should be added to the **LD_LIBRARY_PATH** (on Linux) or **DYLD_LIBRARY_PATH** (on Mac OS X) environment variable; this should be of the form `$INSTALL_DIR/lib``.

Note: If swig is in a non-standard place use the ‘`--with-swig=$PATH_TO_SWIG_EXECUTABLE`’ flag with the configure step.

Note: See [SURF2007](#) for further details on installing and running Trilinos as well as installing on 64 bit machines.

Trilinos solvers can be used to replace *PySparse* solvers. If both *PySparse* and *Trilinos* are present, usage can be controlled by setting the **FIPY_SOLVERS** environment variable to *Trilinos* or *Pysparse*, or by passing a `--Trilinos` or `--Pysparse` flag to the *FiPy* script, overriding the environment. In the absence of these indicators, *FiPy* will default to using *PySparse* if it is present.

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.

2.9 Platform-Specific Instructions

2.9.1 Simple Mac OS X Installation

We present four comparatively simple routes to installing *FiPy* on Mac OS X. The Fink Installation procedure is appropriate if you are already familiar with the *Fink* package manager. Enthought Python Distribution will get *FiPy* running in a minimum number of steps. The Binary Installation procedure is the next most expedient if you have never heard of *Fink* or if you are not comfortable with it. Please see the more general [Installation](#) for detailed installation instructions. These instructions are not the only ways to set up *FiPy* on Mac OS X but represent the most expedient ways, from our experience, to have a usable installation up and running.

Attention: You must have an administrator account to install most of the following packages.

Enthought Python Distribution

<http://www.enthought.com/epd>

This installer provides a very large number of useful scientific packages for Python, including Python, NumPy, SciPy, Matplotlib, and IPython.

Attention: PySparse and *FiPy* are not presently included in EPD, so you will need to separately install them manually.

Binary Installation

Attention: Choose this method if you have never heard of [Fink](#) or if you are not comfortable with it for any reason. Binary installation is the fastest way to get *FiPy* up and running, but may offer less flexibility in the long run.

Pre-built binaries for many of the required packages are available at <http://pythonmac.org/packages/py24-fat/>.

Python

Python is pre-installed on [Mac OS X](#), but installation of other packages is much easier if you upgrade to the latest version of `python-2.4.X-XXXX-XX-XX.dmg` from pythonmac` (or possibly some variant on `Universal-MacPython-2.4.X-XXXX-XX-XX.dmg``). Your existing installation will not be harmed.

Note: Any command-line instructions that start with `python` will either need to be explicitly typed as `/usr/local/bin/python` or you will need to adjust your `$path` variable so that this version of `python` is found before the pre-installed version.

Note: Another option is [ActivePython](#), which probably is the most heavily supported installation on the Mac, but seems to lack *readline* support, but these instructions

<http://www.friday.com/bbum/2006/03/06/python-mac-os-x-and-readline/>

worked for us.

NumPy

Download and install the latest version of `numpy-X.XX-py2.4-macosx10.4.mpkg.zip` from [pythonmac](#).

matplotlib

In order to see simulation results, you will need a viewer. We recommend you download and install the latest version of `matplotlib-X.XX.X-py2.4-macosx10.4.mpkg.zip` from [pythonmac](#).

matplotlib requires:

wxPython Download and install the latest version of `wxPythonX.X-osx-unicode-X.X.X.X-universal10.4-py2.4.dmg` from [pythonmac](#).

PySparse

http://sourceforge.net/project/showfiles.php?group_id=101403

Download and install the latest version of `pysparse-X.XX.XXX.macosx-10.4-py24.dmg`

FiPy

<http://www.ctcms.nist.gov/fipy/>

Download and unpack the source archive (`FiPy-x.y.tar.gz`).

From within the *FiPy* directory, execute the command-line instruction:

```
$ python setup.py build  
$ sudo python setup.py install
```

Note: You may now choose to install Optional Packages or you may choose proceed directly to Using FiPy on Mac OS X.

Fink Installation

Attention: Choose this method if you are already familiar with *Fink* or with *Linux* package managers in general (such as *Debian* packages or *RPMs*). *Fink* installation takes considerably longer than Binary Installation, but offers a wealth of other programs that can make it worthwhile.

The *Fink* package manager automatically handles the many intricate dependencies involved in building open source software. *Fink* is based on the *Debian* tools and the package manager model will be familiar to *Linux* users.

Xcode Development Tools

<http://developer.apple.com/tools/xcode>

Some required packages are not available from *Fink* as binaries, so you will need to have the developer tools for *Mac OS X*. They may already be installed in the */Developer/* directory, but a different version may be required by *Fink*; see the recommendations at <http://fink.sourceforge.net/download>

Note: Free registration with the Apple Developer Connection is required.

X11

Open the X11 application.

Set your **DISPLAY** environment variable to :0.0.

Note: If the X11 application is not already present in the */Applications/Utilities/* directory, it should be available as an optional package on the OS installation media that came with your computer.

Fink

<http://fink.sourceforge.net/download>

Ensure that *Fink* is installed and up to date for your OS.

Note: The following steps have been tested with *Fink* 0.8.1 on *Mac OS X* 10.4 “Tiger”. Variations may be necessary for other OS versions.

unstable tree Follow the directions at <http://www.finkproject.org/faq/usage-fink.php#unstable>

Note: We recommend that you accept all defaults presented by `fink selfupdate`.

Note: “unstable” is not as scary as it sounds. The *Fink* administrators tend to be very conservative about what packages are designated “stable”.

Remaining Fink packages Execute the following commands from Terminal application (you can use `xterm` or any other terminal application of your choosing):

```
$ fink --use-binary-dist install python
```

Take note of the version of Python that gets installed (`python --version`). Many other packages, indicated by “`-pyXX`” suffix, require you substitute the Python version. E.g., Python 2.4 takes “`-py24`”, Python 2.5 takes “`-py25`”, and so on:

```
$ fink --use-binary-dist install matplotlib-pyXX
```

Attention: The `matplotlib` installation will automatically download and build a number of other packages. This process can take quite awhile. We recommend that you accept all defaults offered at the beginning of this process.

Note: If the installation of ‘`matplotlib-pyXX`’ fails for some reason, we recommend you execute the `install` command again.

A few changes are needed to allow `matplotlib` to run:

```
$ mkdir ~/.matplotlib
$ curl http://matplotlib.sourceforge.net/matplotlibrc \
> ~/.matplotlib/matplotlibrc
```

You may now choose to either edit the “backend” configuration in `~/.matplotlib/matplotlibrc` to read:

```
backend      : TkAgg
```

or you can install wxPython with:

```
$ fink --use-binary-dist install wxpython-pyXX
```

(the second choice takes awhile, as it needs to build things).

PySparse installation

http://sourceforge.net/project/showfiles.php?group_id=101403

Download and unpack the latest version of `pysparse-X.XX.XXX.tar.gz`

From within the PySparse directory, execute:

```
$ python setup.py build
$ sudo python setup.py install
```

FiPy installation

Install FiPy packages as explained above.

Note: You may now choose to install Optional Packages or you may choose proceed directly to Using FiPy on Mac OS X.

Optional Packages

IPython

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

This interactive Python shell is nicer to use than the default, and integrates nicely with matplotlib. Download the source and follow the building and installation instructions for [Mac OS X](#).

Gmsh

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

If you wish to run examples that have unstructured meshes, it is necessary to install Gmsh. Download and unpack the latest version of Gmsh for *Mac OS X*. Create a link on your \$path or a shell alias that points to <Gmsh path>/Gmsh.app/Contents/MacOS/Gmsh.

Note: This is a required package for superfill examples.

Mayavi

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

[Mayavi 2](#) is a requirement if you wish to view 3D problems or improve the viewing capabilities of the superfill examples. The standard instructions for either installing with Enthought Python Distribution or installing manually work fine.

If you have already followed the Fink Installation instructions, then you should be able to go to the command line and type:

```
$ sudo apt-get install mayavi2-pyXX
```

SciPy

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

This is a very powerful set of tools that augments the capabilities of [FiPy](#). Although not required for using [FiPy](#), some tests will fail if it is not present:

- If you followed the Binary Installation procedure, there are a few different choices for obtaining prebuilt binaries of SciPy, each with their own issues:
 - We presently recommend obtaining SciPy from the [ScipySuperpack](#)

Warning: We do *not* recommend installing the other components from the [ScipySuperpack](#). In particular, matplotlib was not usable when we tried it.

- pythonmac includes a build of SciPy, but the latest version we tried from [pythonmac](#), [scipy-0.5.1-py2.4-macosx10.4.mpkg.zip](#) (MD5: 15daecd1b5709f04a41154102269359f), was apparently not linked correctly and does not work properly, c.f.

<http://projects.scipy.org/pipermail/scipy-user/2007-January/010820.html>

- We *may* provide builds of SciPy from our own site if we conclude that we can better serve *FiPy* users that way.
- If you followed the Fink Installation procedure, then you should be able to type:

```
$ sudo apt-get install scipy-pyXX
```

Note: You are now ready to proceed to Using FiPy on Mac OS X.

Using FiPy on Mac OS X

We do a substantial amount of our *FiPy* development on Mac OS X, so you can assume that it is well-tested on this platform. See *Using FiPy* for more information.

IDLE Environment

For those that are averse to the command line, the IDLE environment is installed by the pythonmac Python installer and will appear in the MacPython 2.4 folder of the Applications folder.

Note: We are not aware of a Fink package for IDLE.

Attention: We have no experience with using the IDLE environment on Mac OS X, but the following steps do work.

You can use the IDLE file browser to open the examples and run the module.

- Open the IDLE application, located in /Applications/MacPython 2.4/
- Select the Python Shell window. You can close the Console window if it appears.
- Choose File > Open
- Select /Path/To/Base/FiPy/Directory/examples/diffusion/mesh1D.py and click the Open button

The script will open in an editor window.

- Choose Run > Run Module

A matplotlib viewer should appear and the Python Shell should prompt you through a series of examples.

2.9.2 Simple Windows Installation

These instructions are for the Windows XP and Windows 2000 platforms. Please see the more general *Installation* for detailed installation instructions. These instructions are not the only way to set up *FiPy* on a Windows OS but represent the most expedient way from our experience to have a usable installation up and running.

Required Packages

Python

<http://www.pythonxy.com>

<http://www.enthought.com>

We recommend the use of either [Enthought Python Distribution](#) or [Python\(x,y\)](#). These versions of Python have some of the prerequisite packages for *FiPy* already included. Download and install the latest version.

PySparse

http://sourceforge.net/project/showfiles.php?group_id=101403

Download and install the latest version of PySparse for Windows (`pysparse-x.y.z.win32-py2.X.exe`). Be sure to select the version compiled with the correct version of [Python](#) to match the [Python](#) installation.

FiPy

<http://www.ctcms.nist.gov/fipy/download/>

Download and unpack the zip file (`Fipy-x.y.win32.zip`). Run the *FiPy* installer `Fipy-x.y.win32.exe`, which is in the base `Fipy-x.y` directory.

Optional Packages

Gmsh

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

If you wish to run examples that have unstructured meshes, it is necessary to install Gmsh. Download and unpack the latest version of Gmsh for Windows. Open the unpacked folder with a browser and make sure that `gmsh.exe` is placed somewhere on the execution path.

Mayavi

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

Mayavi is a requirement if you wish to view 3D problems or improve the viewing capabilities of the superfill examples. Either [Enthought Python Distribution](#) or [Python\(x,y\)](#) is advised.

Using FiPy on Windows

A number of interactive python environments are available such as the [IDLE](#) and [IPython](#) environments. The following videos may be useful for explaining the use of [IPython](#) on Windows:

<http://showmedo.com/videos/series?name=PythonIPythonSeries>

Testing

If you have a working copy of the source, not an installed version of *FiPy*, you can run the tests using [IPython](#) from the base *FiPy* directory, by typing

`>>> run setup.py test`

in the [IPython](#) shell.

Running Examples

To run the *FiPy* examples in IPython simply use the *run* command:

```
>>> run examples/diffusion/mesh20x20.py
```

2.9.3 SVN usage

All stages of *FiPy* development are archived in a Subversion (SVN) repository at [MatForge](#). You can browse through the code at <http://matforge.org/fipy/browser> and, using an SVN 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*.

SVN client

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

Mac OS X client

You can obtain a binary installer of svn from

<http://www.codingmonkeys.de/mbo/>

Alternatively, if you are using Fink, then you can execute the command:

```
$ sudo apt-get install svn-client
```

If you prefer a GUI, after you install svn, you can obtain svnX from

<http://www.lachoseinteractive.net/en/community/subversion/svnx>

SVN tags

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 “tracking tags” are used to indicate different stages of *FiPy* development. You will need to decide on your own risk tolerance when deciding which stage of development to track.

A fresh copy of *FiPy* that is designated by a particular <tag> can be obtained with:

```
$ svn checkout http://matforge.org/svn/fipy/<tag>
```

An existing SVN checkout of FiPy can be shifted to a different state of development by issuing the command:

```
$ svn switch http://matforge.org/svn/fipy/<tag> .
```

in the base directory of the working copy. The main tags (<tag>) for FiPy are:

tags/version-x_y designates a released version x.y. Any released version of Fipy will be designated with a fixed tag: The current version of Fipy is 2.1.

branches/version-x_y designates a line of development based on a previously released version (i.e., if current development work is being spent on version 0.2, but a bug is found and fixed in version 0.1, that fix will be tagged as `tags/version-0_1_1`, and can be obtained from the tip of `branches/version-0_1`).

In addition:

trunk designates the latest revision of any file present in the repository. Fipy is not guaranteed to pass its tests or to be in a consistent state when checked out under this tag.

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

Note: We formerly provided `tags/STABLE` and `tags/CURRENT`. Our experience has been that these tags serve little purpose. They were invariably set to point at the same revision and that was frequently far out of date from what we were using for our own research. Rather than trying to make these tags relevant, we think it's preferable to direct users to track either `trunk` or some specific `version-x_y`. An existing working copy can be switched with, e.g.:

```
$ svn switch http://matforge.org/svn/fipy/trunk
```

For some time now, we have done all significant development work on branches, only merged back to `trunk` when the tests pass successfully. Although we cannot guarantee that `trunk` will never be broken, you can always check our build status page

<http://matforge.org/fipy/build>

to find the most recent revision that it is running acceptably.

For those who are interested in learning more about Subversion, the canonical manual is the [online Subversion Red Bean book \[SubversionRedBean\]](#).

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 [[croftphd](#)], Patanker [[patanker](#)] or Versteeg and Malalasekera [[versteegMalalasekera](#)].

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 [[Mattiussi:1997](#)]. Analogously, the the discretization of equations with the FVM reduces to the FDM on Cartesian grids.

3.1 General Conservation Equation

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

$$\frac{\partial \phi}{\partial t} = H(\phi, \lambda_i) \quad (3.1)$$

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

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

$$\underbrace{\frac{\partial(\rho\phi)}{\partial t}}_{\text{transient}} = \underbrace{\nabla \cdot (\vec{u}\phi)}_{\text{convection}} + \underbrace{[\nabla \cdot (\Gamma_i \nabla)]^n \phi}_{\text{diffusion}} + \underbrace{S_\phi}_{\text{source}} \quad (3.2)$$

where ρ , \vec{u} and Γ_i represent coefficients in the transient, convection and diffusion terms, respectively. These coefficients can be arbitrary functions of any parameters or variables in the system. The variable ϕ represents the unknown quantity in the equation. The diffusion term can represent any higher order diffusion-like term, where the order is given by the exponent n . For example, the diffusion term can represent conventional Fickian diffusion [*i.e.*, $\nabla \cdot (\Gamma \nabla \phi)$] when the exponent $n = 1$ or a Cahn-Hilliard term [*i.e.*, $\nabla \cdot (\Gamma_1 \nabla [\nabla \cdot \Gamma_2 \nabla \phi])$] [[CahnHilliardI](#)] [[CahnHilliardII](#)] [[CahnHilliardIII](#)] when $n = 2$. Of course, higher order terms ($n > 2$) are also possible.

3.2 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.

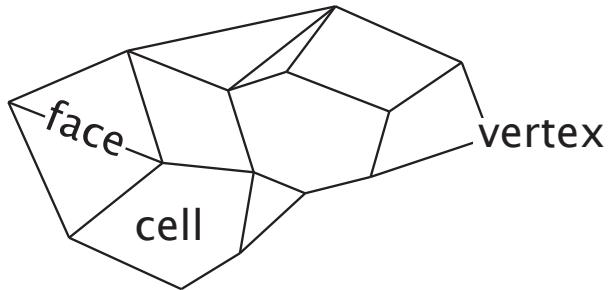


Figure 3.1: Mesh

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.

3.2.1 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.

3.2.2 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.

3.3 Discretization

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

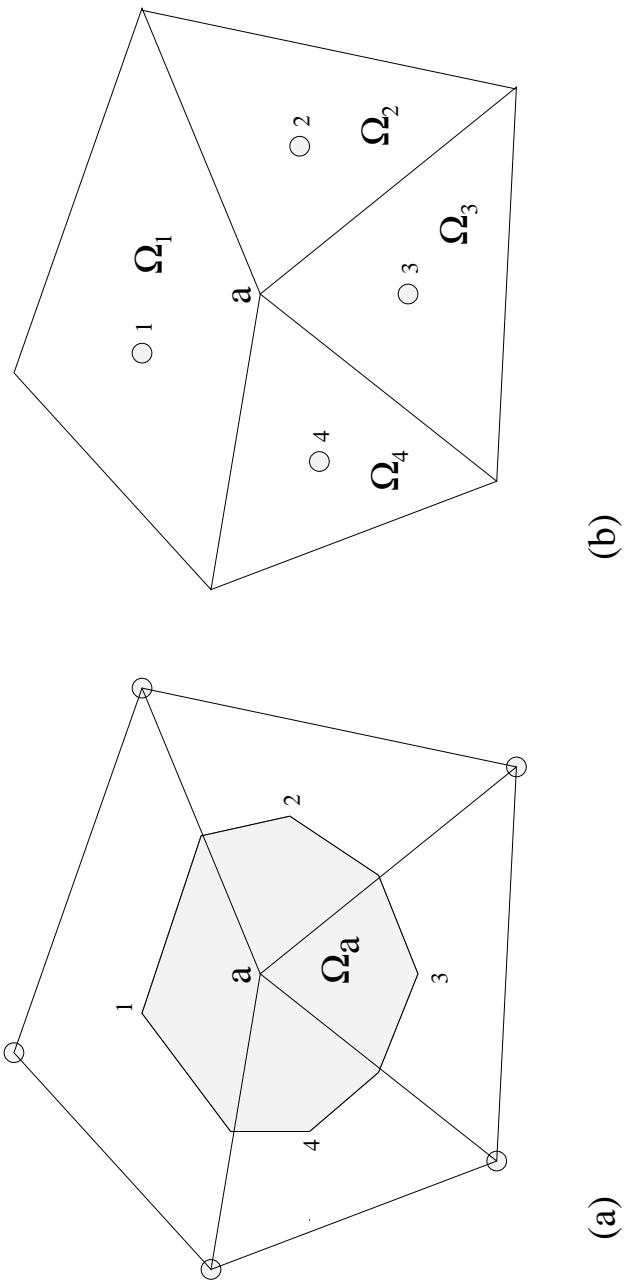


Figure 3.2: CV structure for an unstructured mesh
(a) Ω_a represents a vertex-based CV and (b) $\Omega_1, \Omega_2, \Omega_3$ and Ω_4 represent cell centered CVs.

3.3.1 Transient Term

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

$$\int_V \frac{\partial(\rho\phi)}{\partial t} dV \simeq \frac{(\rho_P\phi_P - \rho_P^{\text{old}}\phi_P^{\text{old}})V_P}{\Delta t} \quad (3.3)$$

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

3.3.2 Convection Term

The discretization for the convection term is given by

$$\begin{aligned} \int_V \nabla \cdot (\vec{u}\phi) dV &= \int_S (\vec{n} \cdot \vec{u})\phi dS \\ &\simeq \sum_f (\vec{n} \cdot \vec{u})_f \phi_f A_f \end{aligned} \quad (3.4)$$

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

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

The weighting factor α_f is determined by the convection scheme, described later in this chapter.

3.3.3 Diffusion Term

The discretization for the diffusion term is given by

$$\begin{aligned} \int_V \nabla \cdot (\Gamma \nabla \{\dots\}) dV &= \int_S \Gamma (\vec{n} \cdot \nabla \{\dots\}) dS \\ &\simeq \sum_f \Gamma_f (\vec{n} \cdot \nabla \{\dots\})_f A_f \end{aligned} \quad (3.5)$$

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

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

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

3.3.4 Source Term

The discretization for the source term is given by,

$$\int_V S_\phi dV \simeq S_\phi V_P. \quad (3.6)$$

Including any negative dependence of S_ϕ on ϕ increases solution stability. The dependence can only be included in a linear manner so Equation (3.6) becomes

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

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

3.4 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* suite of solvers and include most common solvers such as the conjugate gradient method and LU decomposition. There are plans to include other solver suites that are compatible with *Python*.

Combining Equations (3.3), (3.4), (3.5) and (3.6), the complete discretization for equation (??) can now be written for each CV as

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

Equation (3.4) 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

$$a_P \phi_P = \sum_f a_A \phi_A + b_P, \tag{3.7}$$

where

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

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

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

3.5 Numerical Schemes

The coefficients of equation (??) must remain positive, since an increase in a neighboring value must result in an increase in ϕ_P to obtain physically realistic solutions. Thus, the inequalities $a_A > 0$ and $a_A - F_f > 0$ must be

satisfied. The Peclet number $P_f \equiv -F_f/D_f$ is the ratio between convective strength and diffusive conductance. To achieve physically realistic solutions, the inequality

$$\frac{1}{1-\alpha_f} > P_f > -\frac{1}{\alpha_f} \quad (3.8)$$

must be satisfied. The parameter α_f is defined by the chosen scheme, depending on Equation (3.8). The various differencing schemes are:

the central differencing scheme, where

$$\alpha_f = \frac{1}{2} \quad (3.9)$$

so that $|P_f| < 2$ satisfies Equation (3.8). Thus, the central differencing scheme is only numerically stable for a low values of P_f .

the upwind scheme, where

$$\alpha_f = \begin{cases} 1 & \text{if } P_f > 0, \\ 0 & \text{if } P_f < 0. \end{cases} \quad (3.10)$$

Equation (3.10) satisfies the inequality in Equation (3.8) 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

$$\alpha_f = \frac{(P_f - 1) \exp(P_f) + 1}{P_f(\exp(P_f) - 1)}. \quad (3.11)$$

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

$$\alpha_f = \begin{cases} \frac{P_f - 1}{P_f} & \text{if } P_f > 2, \\ \frac{1}{2} & \text{if } |P_f| < 2, \\ -\frac{1}{P_f} & \text{if } P_f < -2. \end{cases} \quad (3.12)$$

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

$$\alpha_f = \begin{cases} \frac{P_f - 1}{P_f} & \text{if } P_f > 10, \\ \frac{(P_f - 1) + (1 - P_f/10)^5}{P_f} & \text{if } 0 < P_f < 10, \\ \frac{(1 - P_f/10)^5 - 1}{P_f} & \text{if } -10 < P_f < 0, \\ -\frac{1}{P_f} & \text{if } P_f < -10. \end{cases} \quad (3.13)$$

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.

4.1 Design

4.1.1 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.

4.1.2 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.

4.1.3 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.

4.1.4 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.

4.1.5 High-Level Scripting Language

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

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

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

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

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

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

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.

4.1.6 Python Programming Language

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

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

4.2 Implementation

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

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

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

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

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

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

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

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

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

A **Term** object represents any of the terms in Equation (??) 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 (??), 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 (??).

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

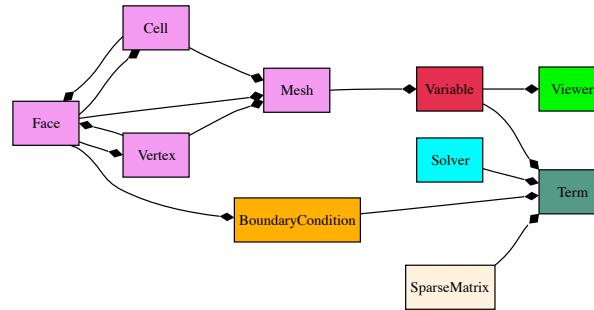


Figure 4.1: Primary object relationships in *FiPy*.

A **Mesh** object is composed of **Cell** objects. Each **Cell** is defined by its bounding **Face** objects and each **Face** is defined by its bounding **Vertex** objects. 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.

Frequently Asked Questions

5.1 How do I represent an equation in FiPy?

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

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

A physical problem can involve many different coupled governing equations, one for each variable. Numerous specific examples are presented in Part *Examples*, but let us examine this general expression term-by-term:

5.1.1 How do I represent a transient term $\partial(\rho\phi)/\partial t$?

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

Note: We have specified neither the variable ϕ nor the time step. Both are handled when we actually solve the equation.

5.1.2 How do I represent a convection term $\nabla \cdot (\vec{u}\phi)$?

```
>>> <SpecificConvectionTerm>(coeff = u,
...                               diffusionTerm = diffTerm)
```

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.

Note: As discussed in *Numerical Schemes*, the convection schemes need to calculate a Peclet number, and therefore need to know about any diffusion term used in the problem. It is hoped that this dependency can be automated in the future.

5.1.3 How do I represent a diffusion term $\nabla \cdot (\Gamma_1 \nabla \phi)$?

Either

```
>>> ImplicitDiffusionTerm(coeff = Gamma1)
```

or

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

`ExplicitDiffusionTerm` is provided only for illustrative purposes. `ImplicitDiffusionTerm` is almost always preferred (`DiffusionTerm` is a synonym for `ImplicitDiffusionTerm` to reinforce this preference). It is theoretically possible to create an explicit diffusion term with

```
>>> (Gamma1 * phi.getFaceGrad()).getDivergence()
```

Unfortunately, in this form, any boundary conditions on ϕ will not be accounted for.

5.1.4 How do I represent a term $\nabla^4\phi$ or $\nabla \cdot (\Gamma_1 \nabla (\nabla \cdot (\Gamma_2 \nabla \phi)))$ such as for Cahn-Hilliard?

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

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

5.1.5 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 the FAQ, [How do I represent a term or such as for Cahn-Hilliard?](#)). 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.getCellCenters()
>>> DiffusionTerm([[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.

5.1.6 What if the term isn't one of those?

Any term that cannot be written in one of the previous forms is considered a source S_ϕ . An explicit source is written in Python essentially as it appears in mathematical form, *e.g.*, $3\kappa^2 + b \sin \theta$ would be written

```
>>> 3 * kappa**2 + b * 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))
```

5.1.7 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 above, if the dependent variable *for that PDE* does not appear in the appropriate place, then that term should be treated as a source.

How do I represent a diffusive source?

If the governing equation for ϕ is

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

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

```
>>> (D2 * xi.getFaceGrad()).getDivergence()
```

Higher order diffusive sources can be obtained by simply nesting the calls to `getFaceGrad()` and `getDivergence()`.

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

How do I represent a convective source?

The convection of an independent field ξ as in

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

can be rendered as

```
>>> (u * xi.getArithmeticFaceValue()).getDivergence()
```

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

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

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

How do I represent a transient source?

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

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

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

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

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

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

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

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

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

```
>>> diffTerm = DiffusionTerm(coeff = D1)
>>> convTerm = <Specific>ConvectionTerm(coeff = D2 * xi.getFaceGrad(),
...                                         diffusionTerm = diffTerm)
>>> eq = TransientTerm() == diffTerm + convTerm
```

5.1.9 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!*.

5.2 How can I see what I'm doing?

5.2.1 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`.

5.2.2 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:

`Pygist` accepts “`.eps`” (Encapsulated PostScript) and “`.cgm`” (Computer Graphics Metafile).

`gnuplot` accepts “`.eps`.”

`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`. 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>.

5.2.3 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` should be adaptable to other `Viewers`.

5.2.4 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` for an example of creating a custom `Matplotlib` `Viewer` class.

5.3 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.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=1)
```

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.binary`, 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.mesh2D`.

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, ...)
...     elapsedTme += timeStep
```

5.4 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 `cellCenter` 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.getArithmeticFaceValue()`. 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.

5.5 How do I represent boundary conditions?

5.5.1 What is a `FixedValue` boundary condition?

This is simply a Dirichlet boundary condition by another name.

5.5.2 What does the `FixedFlux` boundary condition actually represent?

In *FiPy* a `FixedFlux` boundary condition object represents the quantity

$$\Gamma \vec{n} \cdot \nabla \phi - \vec{u} \cdot \vec{n} \phi$$

on a given boundary edge with \vec{n} pointing out of the boundary. The quantity Γ represents the diffusion coefficient and \vec{u} represents the convection coefficient for a general convection-diffusion equation of the type given in Eq. (??). See `examples.convection.robin` for a usage case.

5.5.3 I can't get the `FixedValue` or `FixedFlux` boundary condition objects to work right. What do I do now?

There have been a number of questions on the mailing list about boundary conditions and from the feedback it is clear that there are some problematic issues with the design and implementation of the boundary condition objects. We hope to rectify this in future releases. However, it is possible to specify almost any boundary condition by using a rank 1 `FaceVariable` to represent the external flux value and apply the `getDivergence()` method to this object and then use it as a source term in the given equation. The following code demonstrates how to implement this technique. First define the coefficients,

```
>>> convectionCoeff = FaceVariable(..., rank=1)
>>> diffusionCoeff = FaceVariable(...)
```

where the `convectionCoeff` and `diffusionCoeff` are defined over all the faces. We will define a third `FaceVariable` to represent the boundary source term and then set the values of the coefficients to zero on the exterior faces.

```
>>> boundarySource = FaceVariable(..., rank=1)
>>> convectionCoeff.setValue(0, where=mesh.getExteriorFaces())
>>> diffusionCoeff.setValue(0, where=mesh.getExteriorFaces())
>>> boundarySource.setValue(vectorValues, where=mesh.getExteriorFaces())
```

The `vectorValues` quantity can be set to whatever value is required for the particular boundary condition. The variable `boundarySource` could be a variable that defines a relationship between other variables rather than a simple container object. To finish off, the `boundarySource.getDivergence()` object must be added to the regular equation

```
>>> eqn = (TransientTerm() + ConvectionTerm(convectionSource)
...           = DiffusionSource(diffusionCoeff) + boundarySource.getDivergence())
```

No other boundary conditions need to be applied. It may be necessary to reset or update the values of `boundarySource`, `diffusionCoeff` and `convectionCoeff` at each sweep if they are not automatically updated or if the exterior values need to be reset to zero. For complex boundary conditions, it is often easier to implement the technique described here rather than trying to get the `FixedValue` and `FixedFlux` boundary conditions to work correctly.

5.5.4 How do I apply an outlet or inlet boundary condition?

There is no good way to do this with the standard boundary conditions in `FiPy` and thus one needs to use the method suggested above, see *I can't get the `FixedValue` or `FixedFlux` boundary condition objects to work right. What do I do now?*. Currently, boundary conditions for the `ConvectionTerm` assume a `FixedFlux` boundary condition with a value of 0. This is in fact not the most intuitive default boundary condition, a natural outlet or inlet boundary condition would in fact be more sensible. In order to apply an inlet/outlet boundary condition one needs a separate exterior convection coefficient (velocity vector) to hold the boundary values,

```
>>> convectionCoeff = FaceVariable(..., rank=1)
>>> exteriorCoeff = FaceVariable(..., value=0, rank=1)
```

The `exteriorCoeff` can now be given non-zero values on `inletOutletFaces` and the `convectionCoeff` can be set to zero on these faces.

```
>>> exteriorCoeff.setValue(convectionCoeff, where=inletOutletFaces)
>>> convectionCoeff.setValue(0, where=inletOutletFaces)
```

where the `inletOutletFaces` object are the faces over which the inlet/outlet boundary condition applies. The divergence of the `exteriorCoeff` is then included in the equations with an `ImplicitSourceTerm`. This allows an implicit formulation for outlet boundary conditions and an explicit formulation for inlet boundary conditions, consistent with an upwind convection scheme.

```
>>> eqn = (TransientTerm() + ConvectionTerm(convectionCoeff)
...           + ImplicitSourceTerm(exteriorCoeff.getDivergence())
...           == DiffusionTerm(diffusionCoeff))
```

As in the previous section, the coefficient values may need updating on the exterior faces between sweeps. See `examples.convection.source` for an example of this usage.

5.5.5 How do I apply a fixed gradient?

In general, it is not currently possible to apply a fixed gradient or Von Neumann type boundary condition explicitly. Of course, it is often possible to use `FixedValue` or `FixedFlux` boundary conditions to mimic a fixed gradient condition. In the case when there is no convection, one can simply use a `FixedFlux` condition and multiply through by the diffusion coefficient to create the boundary condition,

```
>>> FixedFlux(value=gradient * diffusionCoeff, faces=myFaces)
```

where `gradient` is the value of the boundary gradient and `myFaces` are the faces over which the boundary condition applies. If the equation contains a `ConvectionTerm` and the boundary condition has a zero gradient then one would use a `FixedValue` boundary condition of the form

```
>>> FixedValue(value=phi.getFaceValue(), faces=myFaces)
```

This is not an “implicit” boundary condition so would in general require sweeps to reach convergence. See `examples.convection.source` for an example of this usage. In the case of a non-zero gradient one would need to employ the techniques in both *I can't get the FixedValue or FixedFlux boundary condition objects to work right. What do I do now?* and *How do I apply an outlet or inlet boundary condition?* without using either a `FixedValue` or a `FixedFlux` object.

5.5.6 How do I apply 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.getFaceCenters()
>>> mask = ((x < 0.5) | (y < 0.5))
>>> BCs = [FixedFlux(value=0, faces=mesh.getExteriorFaces() & mask),
...         FixedValue(value=x * y, faces=mesh.getExteriorFaces() & ~mask)]
```

The `BCs` list can then be passed to the equation’s `solve()` method when its called,

```
>>> eqn.solve(..., boundaryConditions=BCs)
```

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

5.6 What does this error message mean?

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

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

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

repairing catalog by removing key This message (not really an error, but may cause test failures) can result when using the `scipy.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/mailingslists/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>.

5.7 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.

5.7.1 Command-line Flags

You can add any of the following flags after the name of a script you call from the command line

-inline

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

-PySparse

Forces the use of the *PySparse* solvers. This flag takes precedence over the **FIPY_SOLVERS** environment variable.

-Trilinos

Forces the use of the *Trilinos* solvers. This flag takes precedence over the **FIPY_SOLVERS** environment variable.

5.7.2 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()`. If set to “terms,” causes the display of the matrix for each `Term` that composes the equation. Requires the `Matplotlib` package.

FIPY_INLINE

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

FIPY_INLINE_COMMENT

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

FIPY_SOLVERS

Forces the use of the specified suite of linear solvers. Valid (case-insensitive) choices are “PySparse” and “Trilinos”.

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.

5.8 Why don't my scripts work anymore?

FiPy has experienced two major API changes. The steps necessary to upgrade older scripts are discussed in [Updating FiPy](#).

5.9 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 a Tracker request at <<http://matforge.org/fipy/report>>.

Glossary

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

gist Another name for *Pygist*.

gnuplot A popular open-source plotting program. See <http://gnuplot.sourceforge.net/>.

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.nist.gov/cgi-bin/exit_nist.cgi?url=http://www.geuz.org/gmsh.

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://www.nist.gov/cgi-bin/exit_nist.cgi?url=http://matplotlib.sourceforge.net.

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.

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

numarray An archaic predecessor to *NumPy*.

Numeric An archaic predecessor to *NumPy*.

NumPy The *numpy Python* package provides array arithmetic facilities. See http://www.nist.gov/cgi-bin/exit_nist.cgi?url=http://www.scipy.org/NumPy.

Pygist The *Pygist* package can be used to display simulation results. It displays both 1D X-Y type plots and 2D contour plots for both structured and unstructured data. It does not display 3D data. Although stated as working on Windows, it does not seem to do a good job of rendering on this platform. *Pygist* works fine on other common platforms. *Pygist* no longer seems to be under development, but is still recommended as a fast light weight alternative to *Matplotlib*.

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://www.nist.gov/cgi-bin/exit_nist.cgi?url=http://pysparse.sourceforge.net.

Python The programming language that *FiPy* (and your scripts) are written in. See http://www.nist.gov/cgi-bin/exit_nist.cgi?url=http://www.python.org/.

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

SciPy The *scipy Python* pacakge provides a wide range of scientific and mathematical operations. *FiPy* can use *scipy.weave* for enhanced performance with C language inlinine. See http://www.nist.gov/cgi-bin/exit_nist.cgi?url=http://www.scipy.org/.

Sphinx The tools used to generate the *FiPy* documentation. See http://www.nist.gov/cgi-bin/exit_nist.cgi?url=http://sphinx.pocoo.org/.

Trilinos This package proves 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://www.nist.gov/cgi-bin/exit_nist.cgi?url=http://trilinos.sandia.gov.

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*:

```
$ 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](#).

Diffusion Examples

<code>examples.diffusion.mesh1D</code>	To run this example from the base <i>FiPy</i> directory, type:
<code>examples.diffusion.mesh20x2</code>	This example solves a diffusion problem and demonstrates the use of applying boundary condition patches.
<code>examples.diffusion.circle</code>	This example demonstrates how to solve a simple diffusion problem on a non-standard mesh with varying boundary conditions.
<code>examples.diffusion.electro</code>	The Poisson equation is a particular example of the steady-state diffusion
<code>examples.diffusion.nthOrder</code>	This example uses the <code>DiffusionTerm</code> class to solve the equation
<code>examples.diffusion.anisotrop</code>	This example demonstrates how to solve diffusion with an anisotropic coefficient.

7.1 examples.diffusion.mesh1D

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

```
$ 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,

$$\frac{\partial \phi}{\partial t} = D \nabla^2 \phi. \quad (7.1)$$

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

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

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

We'll let

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

for now.

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

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

are formed with a value

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

and a set of faces over which they apply.

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

For example, here the exterior faces on the left of the domain are extracted by `mesh.getFacesLeft `()`. A `FixedValue` boundary condition is created with these faces and a value (`valueLeft`).

```
>>> BCs = (FixedValue(faces=mesh.getFacesRight(), value=valueRight),
...         FixedValue(faces=mesh.getFacesLeft(), value=valueLeft))
```

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

If you have ever tried to numerically solve Eq. (7.1), 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.getCellCenters()[0]
>>> t = timeStepDuration * steps

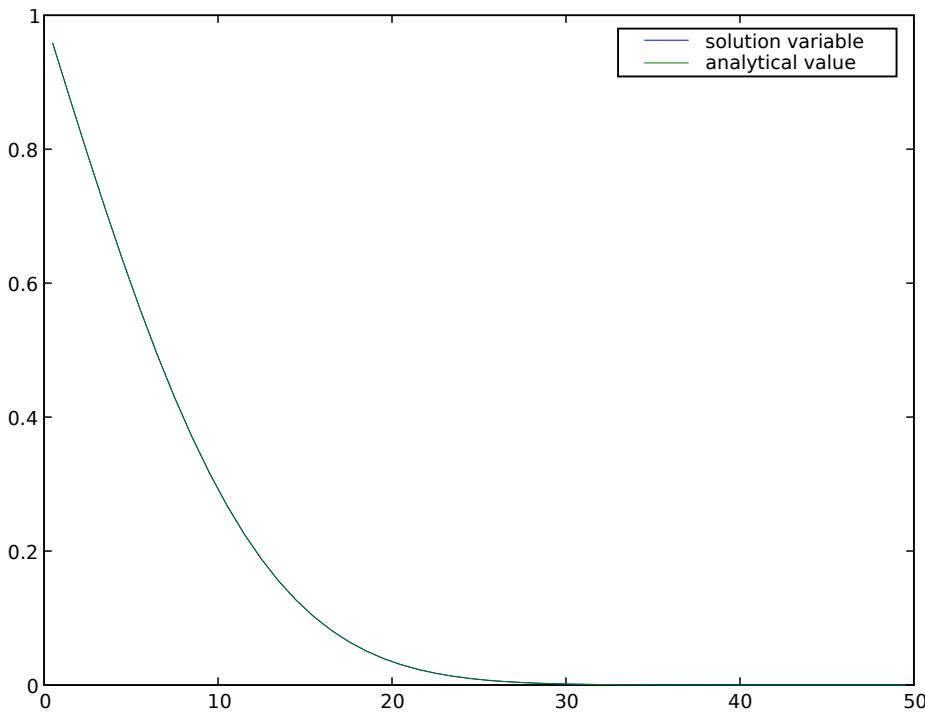
>>> try:
...     from scipy.special import erf
...     phiAnalytical.setValue(1 - erf(x / (2 * sqrt(D * t))))
... except ImportError:
...     print "The SciPy library is not available to test the solution to \
... the transient diffusion equation"
```

We then solve the equation by repeatedly looping in time:

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

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

>>> if __name__ == '__main__':
...     raw_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. (7.1) as:

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

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

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

reset the problem

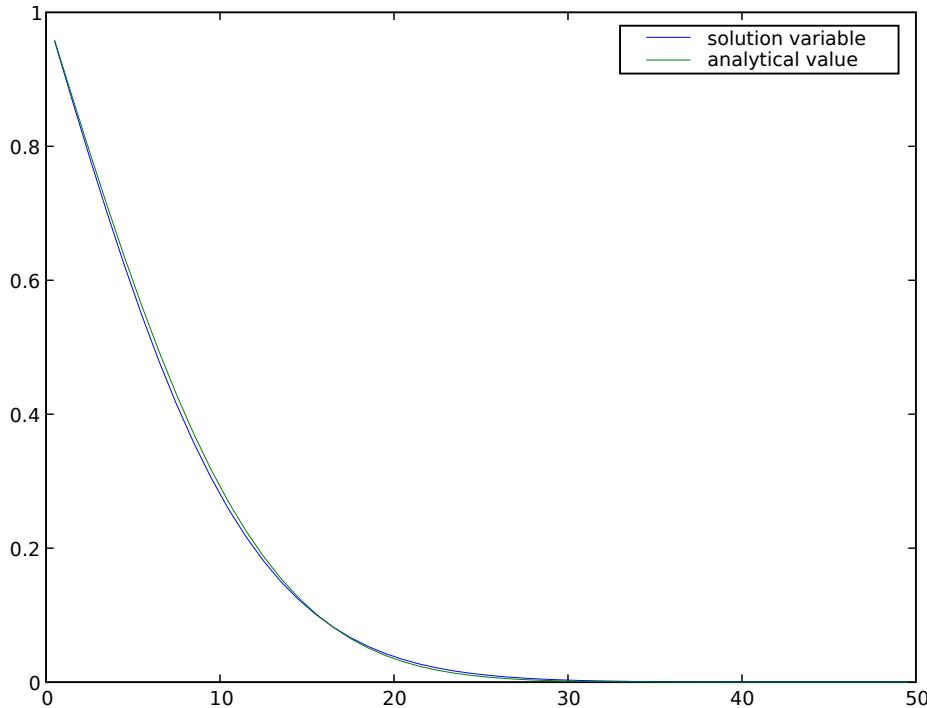
```
>>> phi.setValue(valueRight)
```

and rerun with much larger time steps

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

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

>>> if __name__ == '__main__':
...     raw_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 [NumericalRecipes]).

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

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

>>> if __name__ == '__main__':
...     raw_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

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

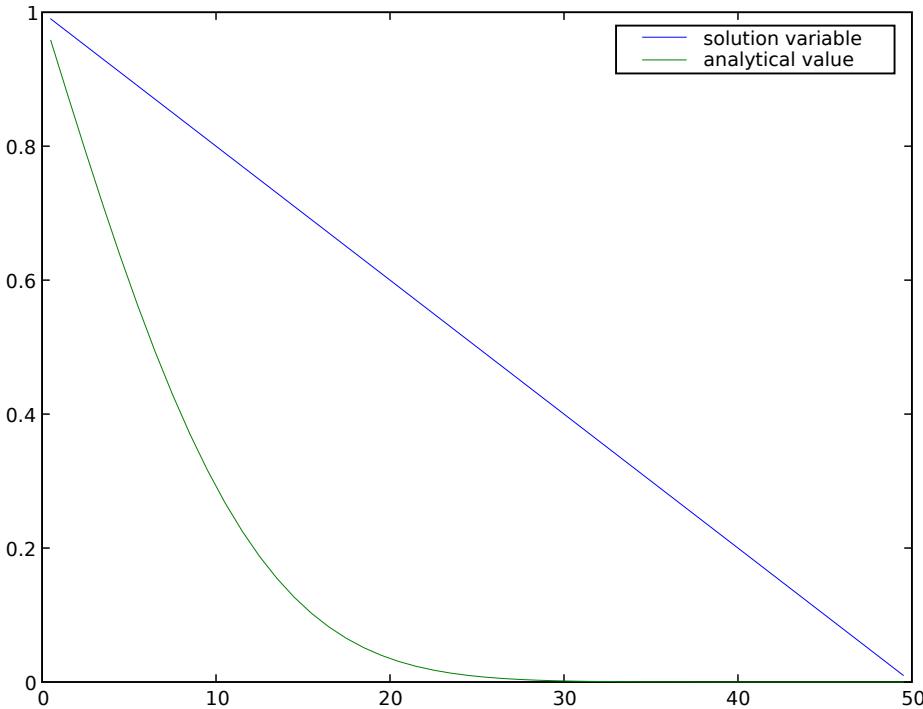
is represented in *FiPy* by

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

>>> if __name__ == '__main__':
...     raw_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()
```

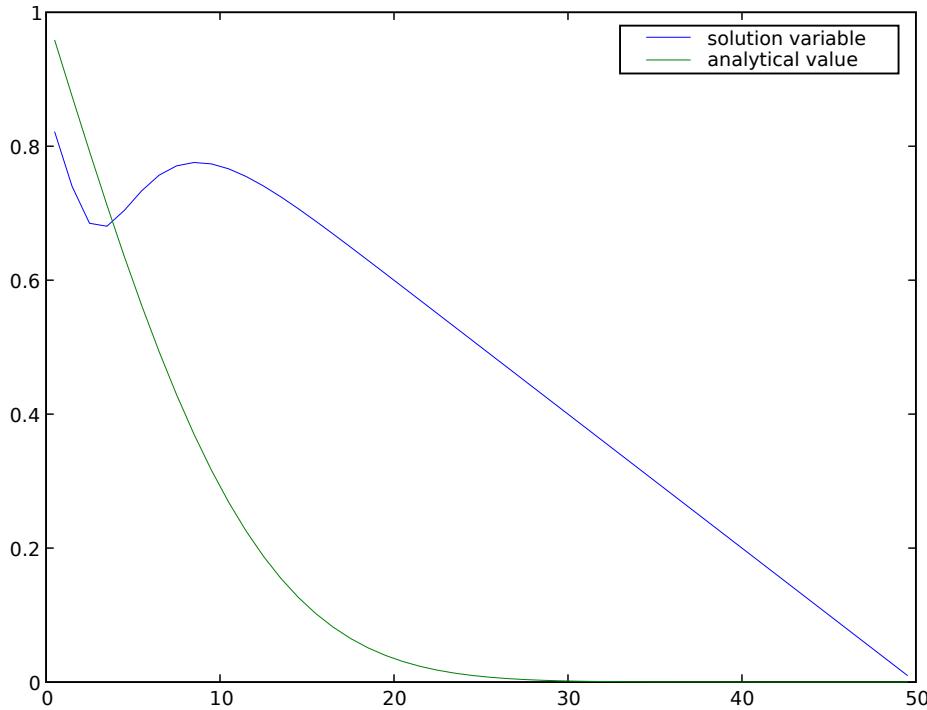
and then declare our boundary condition as a function of this `Variable`

```
>>> BCs = (FixedValue(faces=mesh.getFacesLeft(), value=0.5 * (1 + sin(time))),  
...         FixedValue(faces=mesh.getFacesRight(), value=0.))
```

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

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

```
>>> if __name__ == '__main__':
...     raw_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.getFaceCenters() [0]
>>> D.setValue(0.1, where=(L / 4. <= x) & (x < 3. * L / 4.))
```

The boundary conditions are a fixed value of

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

to the left and a fixed flux of

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

to the right:

```
>>> BCs = (FixedValue(faces=mesh.getFacesLeft(), value=valueLeft),
...         FixedFlux(faces=mesh.getFacesRight(), value=fluxRight))
```

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,
...                                 boundaryConditions = BCs)
```

The analytical solution is simply

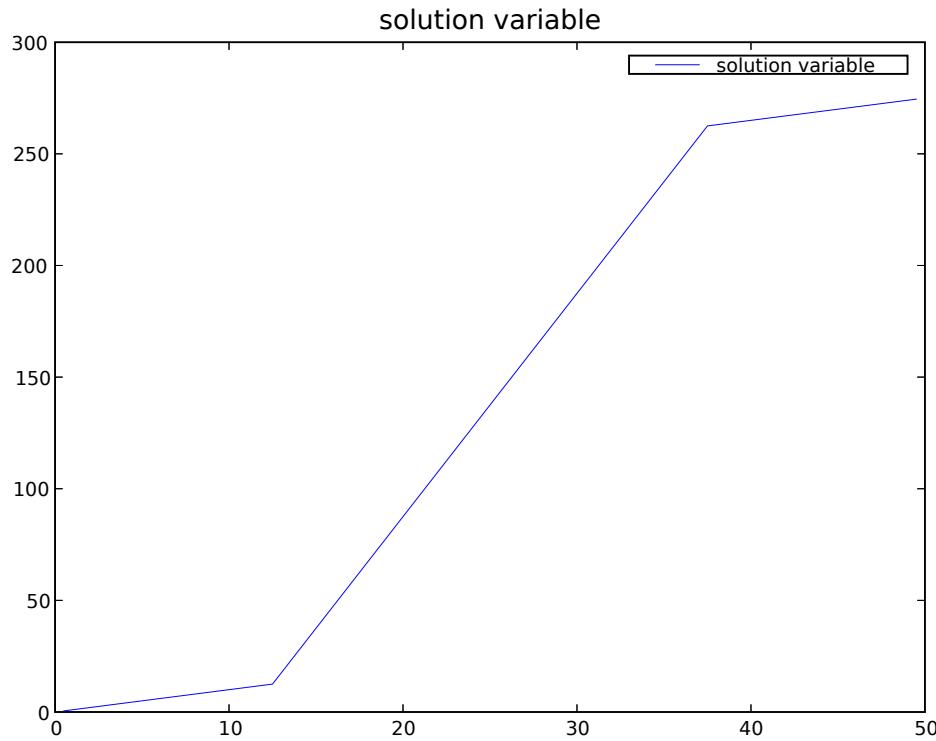
$$\phi = \begin{cases} x & \text{for } 0 < x < L/4, \\ 10x - 9L/4 & \text{for } L/4 \leq x < 3L/4, \\ x + 18L/4 & \text{for } 3L/4 \leq x < L, \end{cases}$$

or

```
>>> x = mesh.getCellCenters() [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

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



Often, the diffusivity is not only non-uniform, but also depends on the value of the variable, such that

$$\frac{\partial \phi}{\partial t} = \nabla \cdot [D(\phi) \nabla \phi]. \quad (7.2)$$

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",
...                 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)]
```

```
...           mesh=mesh)
... ]
```

If, for example,

$$D = D_0(1 - \phi)$$

we would simply write Eq. (7.2) 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

```
>>> BCs = (FixedValue(faces=mesh.getFacesRight(), value=valueRight),
...           FixedValue(faces=mesh.getFacesLeft(), value=valueLeft))
```

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.getCellCenters()[0]
>>> phiAnalytical.setValue(1. - 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.

```
>>> 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],
...                            boundaryConditions=BCs,
...                            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()
...         raw_input("Implicit variable diffusity. %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.

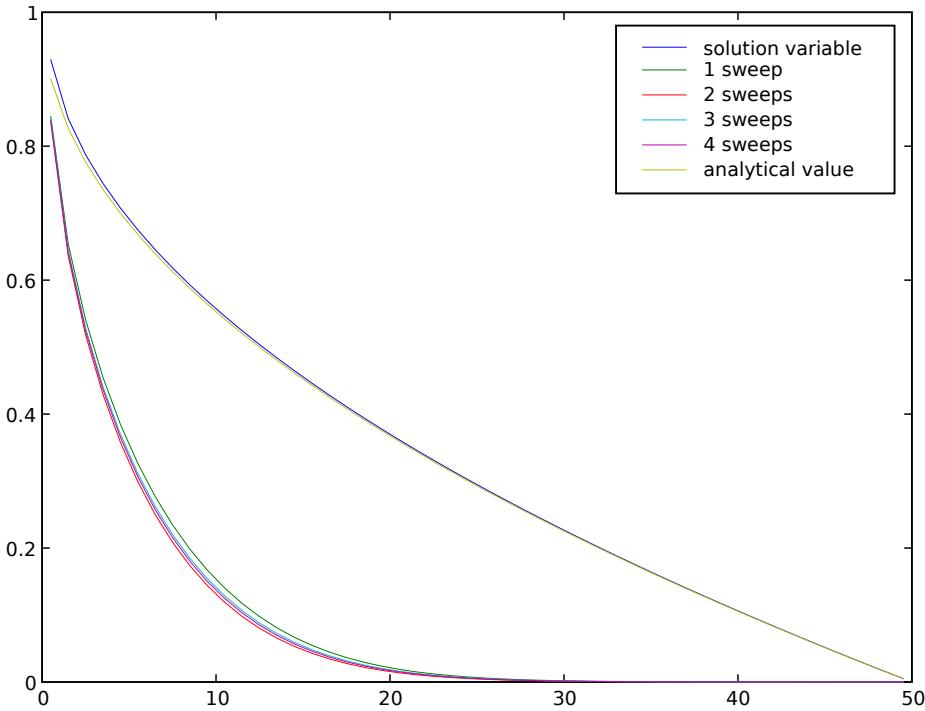
Finally, we can increase the number of steps to approach equilibrium, or we can just solve for it directly

```
>>> eq = DiffusionTerm(coeff=D0 * (1 - phi[0]))

>>> phi[0].setValue(valueRight)
>>> res = 1e+10
>>> while res > 1e-6:
...     res = eq.sweep(var=phi[0],
...                     boundaryConditions=BCs,
...                     dt=timeStepDuration)

>>> print phi[0].allclose(phiAnalytical, atol = 1e-1)
1

>>> if __name__ == '__main__':
...     viewer.plot()
...     raw_input("Implicit variable diffusity - steady-state. \
... Press <return> to proceed...")
```



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:

```
#!/usr/bin/env python

## This script was derived from
## 'examples/diffusion/mesh1D.py'

nx = 50
dx = 1.
mesh = Grid1D(nx = nx, dx = dx)
phi = CellVariable(name="solution variable",
                   mesh=mesh,
                   value=0)

eq = DiffusionTerm(coeff=D0 * (1 - phi[0]))
phi[0].setValue(valueRight)
res = 1e+10
while res > 1e-6:
    res = eq.sweep(var=phi[0],
                    boundaryConditions=BCs,
                    dt=timeStepDuration)

print phi[0].allclose(phiAnalytical, atol = 1e-1)
# Expect:
# 1
```

```
#  
if __name__ == '__main__':  
    viewer.plot()  
    raw_input("Implicit variable diffusity - 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.

7.2 examples.diffusion.mesh20x20

This example solves a diffusion problem and demonstrates the use of applying boundary condition patches.

```
>>> from fipy import *  
  
>>> 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.getFaceCenters()  
>>> facesTopLeft = ((mesh.getFacesLeft() & (y > L / 2))  
...                  | (mesh.getFacesTop() & (x < L / 2)))  
>>> facesBottomRight = ((mesh.getFacesRight() & (y < L / 2))  
...                     | (mesh.getFacesBottom() & (x > L / 2)))
```

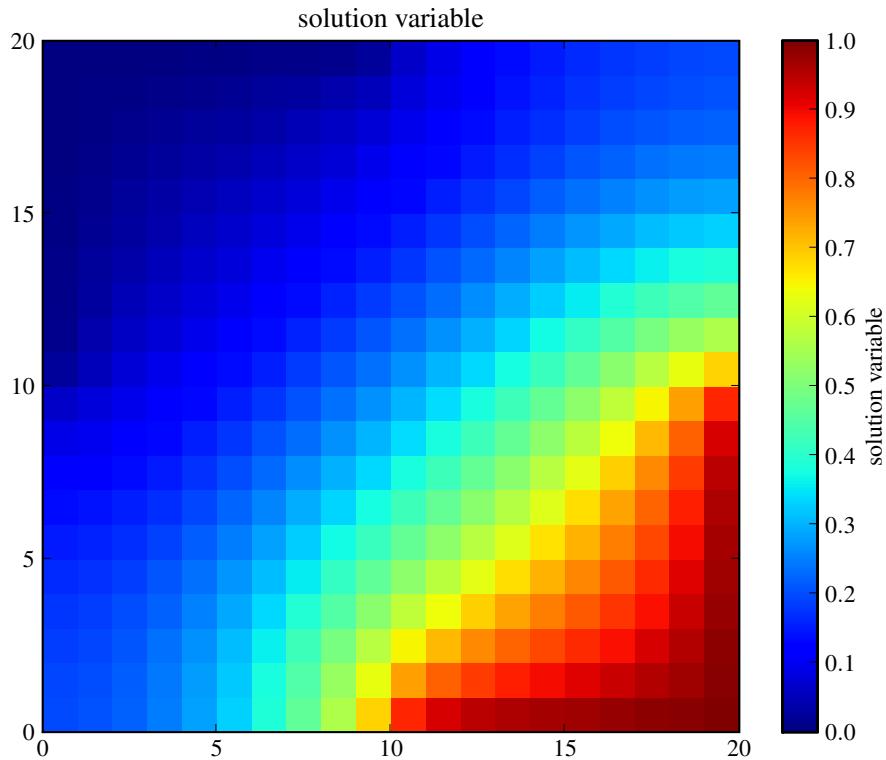
```
>>> BCs = (FixedValue(faces=facesTopLeft, value=valueTopLeft),
...           FixedValue(faces=facesBottomRight, value=valueBottomRight))
```

We create a viewer to see the results

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

and solve the equation by repeatedly looping in time:

```
>>> timeStepDuration = 10 * 0.9 * dx**2 / (2 * D)
>>> steps = 10
>>> for step in range(steps):
...     eq.solve(var=phi,
...              boundaryConditions=BCs,
...              dt=timeStepDuration)
...     if __name__ == '__main__':
...         viewer.plot()
... 
```



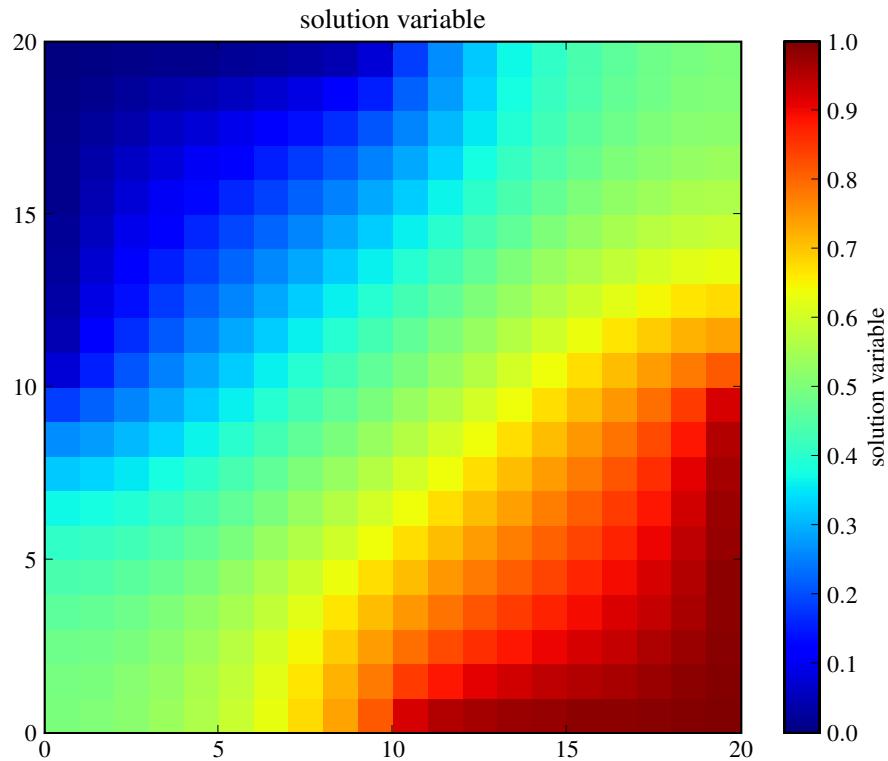
We can test the value of the bottom-right corner cell.

```
>>> print numerix.allclose(phi(((L,), (0,))), valueBottomRight, atol = 1e-2)
1
```

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

We can also solve the steady-state problem directly

```
>>> DiffusionTerm().solve(var=phi,
...                         boundaryConditions = BCs)
>>> 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

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

7.3 examples.diffusion.circle

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 `GmshImporter2D` object.

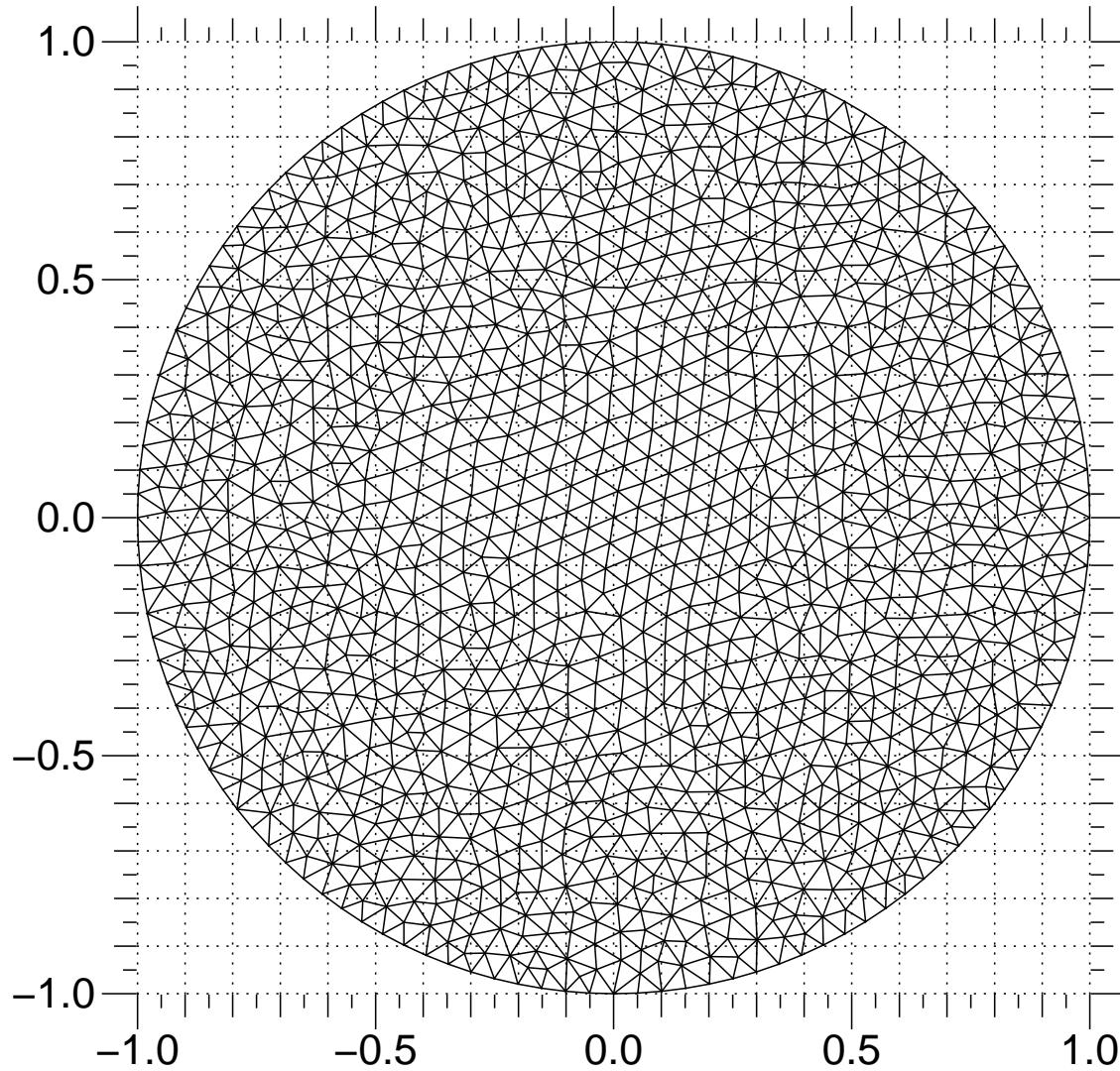
```
>>> from fipy import *
>>> mesh = GmshImporter2D('''
...
...     cellSize = %(cellSize)g;
...     radius = %(radius)g;
...     Point(1) = {0, 0, 0, cellSize};
...     Point(2) = {-radius, 0, 0, cellSize};
...     Point(3) = {0, radius, 0, cellSize};
...     Point(4) = {radius, 0, 0, cellSize};
...     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
>>> if __name__ == '__main__':
...     try:
...         viewer = Viewer(vars=phi, datamin=-1, datamax=1.)
...         viewer.plotMesh()
...         raw_input("Irregular circular mesh. Press <return> to proceed...")
...     except:
...         print "Unable to create a viewer for an irregular mesh (try Gist2DViewer, Matplotlib2DVi...
```



We set up a transient diffusion equation

```
>>> D = 1.  
>>> eq = TransientTerm() == DiffusionTerm(coeff=D)
```

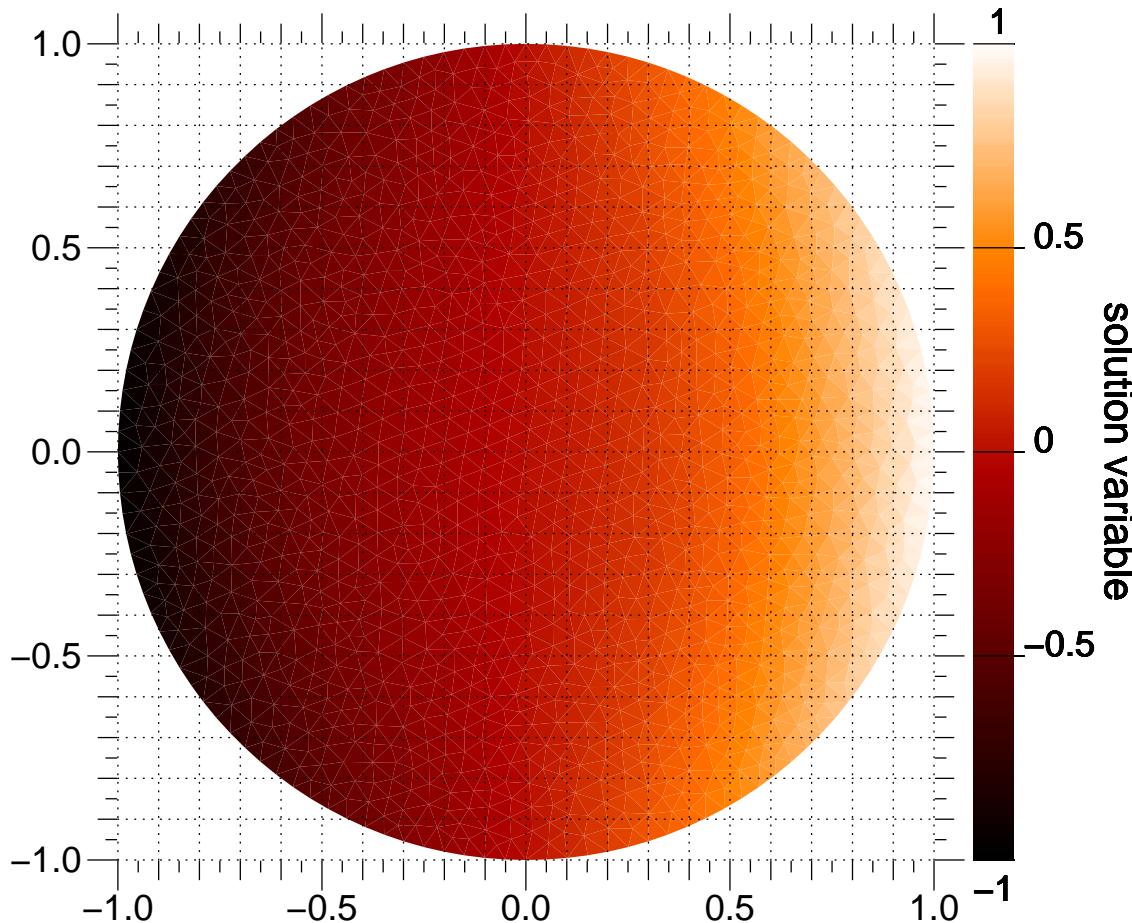
The following line extracts the x coordinate values on the exterior faces. These are used as the boundary condition fixed values.

```
>>> X, Y = mesh.getFaceCenters()  
  
>>> BCs = (FixedValue(faces=mesh.getExteriorFaces(), value=X), )
```

We first step through the transient problem

```
>>> timeStepDuration = 10 * 0.9 * cellSize**2 / (2 * D)  
>>> steps = 10  
>>> for step in range(steps):  
...     eq.solve(var=phi,  
...              boundaryConditions=BCs,  
...              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.getGrad())).plot(filename="myTSV.tsv")
```

x	y	solution variable	solution variable_grad_x	solution variable_grad_y
0.975559734792414	0.0755414402612554	0.964844363287199	-0.229687917881182	
0.0442864953037566	0.79191893162384	0.0375859836421991	-0.773936613923853	
0.0246775505084069	0.771959648896982	0.020853932412869	-0.723540342405813	
0.223345558247991	-0.807931073108895	0.203035857140125	-0.777466238738658	
-0.00726763301939488	-0.775978916110686	-0.00412895434496877	-0.650055516507232	
-0.0220279064527904	-0.187563765977912	-0.012771874945585	-0.35707168379437	
0.111223320911545	-0.679586798311355	0.0911595298310758	-0.613455176718145	
-0.78996770899909	-0.0173672729866294	-0.693887874335319	-1.00671109050419	
-0.703545986179876	-0.435813500559859	-0.635004192597412	-0.896203033957194	
0.888641841567831	-0.408558914368324	0.877939107374768	-0.32195762184087	
0.38212257821916	-0.51732949653553	0.292889724306196	-0.854466141879776	
-0.359068256998365	0.757882581524374	-0.323541041763627	-0.870534227755687	
-0.459673905457569	-0.701526587772079	-0.417577664032421	-0.725460726303266	
-0.338256179134518	-0.523565732643067	-0.254030052182524	-0.923505840608445	

0.87498754712638	0.174119064688993	0.836057900916614	-1.11590500805745
-0.484106960369249	0.0705987421869745	-0.319827850867342	-0.867894407968447
-0.0221203060940465	-0.216026820080053	-0.0152729438559779	-0.341246696530392

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.getCellCenters()
>>> t = timeStepDuration * steps

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

>>> x0 = radius * cos(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 * sqrt(D * t)))
...                            - erf((x0-x) / (2 * 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

>>> if __name__ == '__main__':
...     raw_input("Transient diffusion. Press <return> to proceed...")
```

As in the earlier examples, we can also directly solve the steady-state diffusion problem.

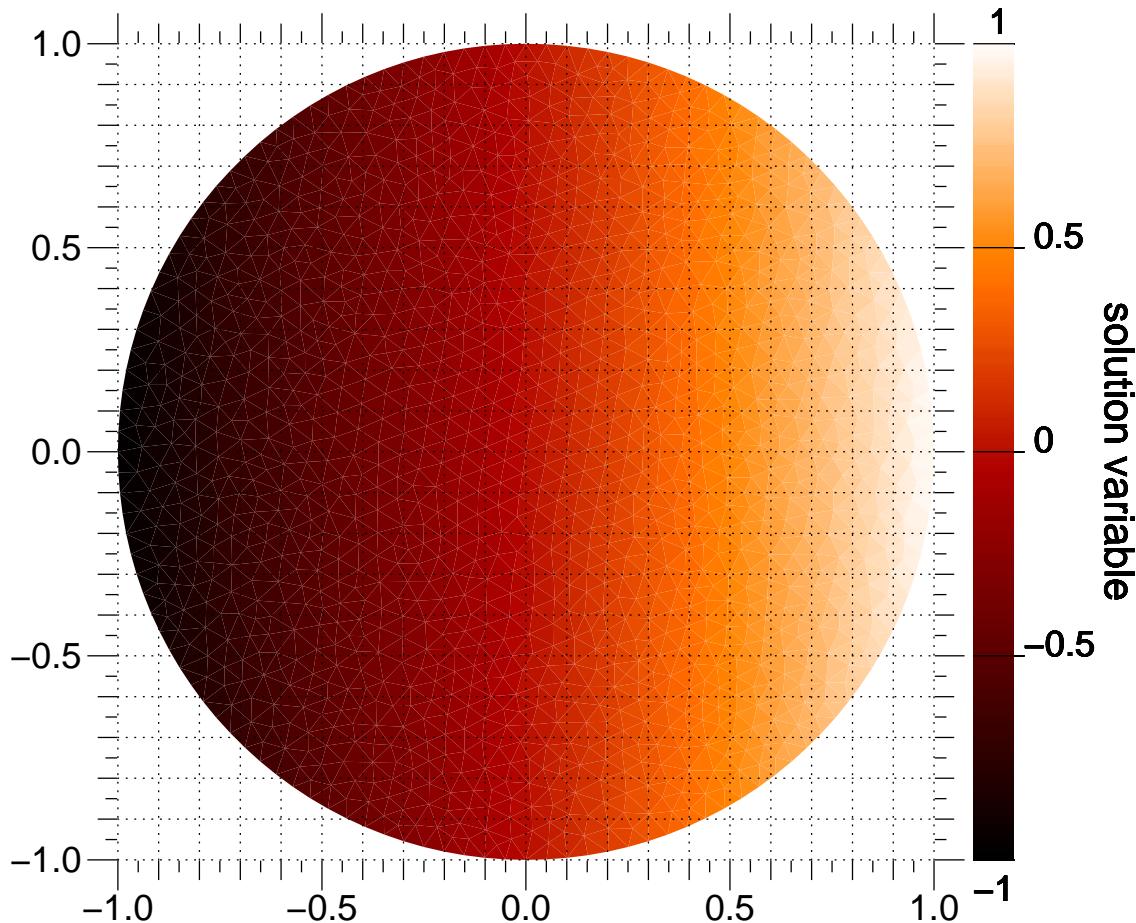
```
>>> DiffusionTerm(coeff=D).solve(var=phi,
...                                 boundaryConditions=BCs)
```

The values at the elements should be equal to their x coordinate

```
>>> print phi.allclose(x, atol = 0.02)
1
```

Display the results if run as a script.

```
>>> if viewer is not None:
...     viewer.plot()
...     raw_input("Steady-state diffusion. Press <return> to proceed...")
```



7.4 examples.diffusion.electrostatics

The Poisson equation is a particular example of the steady-state diffusion equation. We examine a few cases in one dimension.

```
>>> from fipy import *
>>> nx = 200
>>> dx = 0.01
>>> L = nx * dx
>>> mesh = Grid1D(dx = dx, nx = nx)
```

Given the electrostatic potential ϕ ,

```
>>> potential = CellVariable(mesh=mesh, name='potential', value=0.)
```

the permittivity ϵ ,

```
>>> permittivity = 1
```

the concentration C_j of the j^{th} component with valence z_j (we consider only a single component C_e^- with valence with $z_{e^-} = -1$)

```
>>> electrons = CellVariable(mesh=mesh, name='e-')
>>> electrons.valence = -1
```

and the charge density ρ ,

```
>>> charge = electrons * electrons.valence
>>> charge.name = "charge"
```

The dimensionless Poisson equation is

$$\nabla \cdot (\epsilon \nabla \phi) = -\rho = - \sum_{j=1}^n z_j C_j$$

```
>>> potential.equation = (DiffusionTerm(coeff = permittivity)
...                         + charge == 0)
```

Because this equation admits an infinite number of potential profiles, we must constrain the solution by fixing the potential at one point:

```
>>> bcs = (FixedValue(faces=mesh.getFacesLeft(), value=0),)
```

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,
...                             boundaryConditions=bcs)
```

This problem has the analytical solution

$$\psi(x) = \frac{x^2}{2} - 2x$$

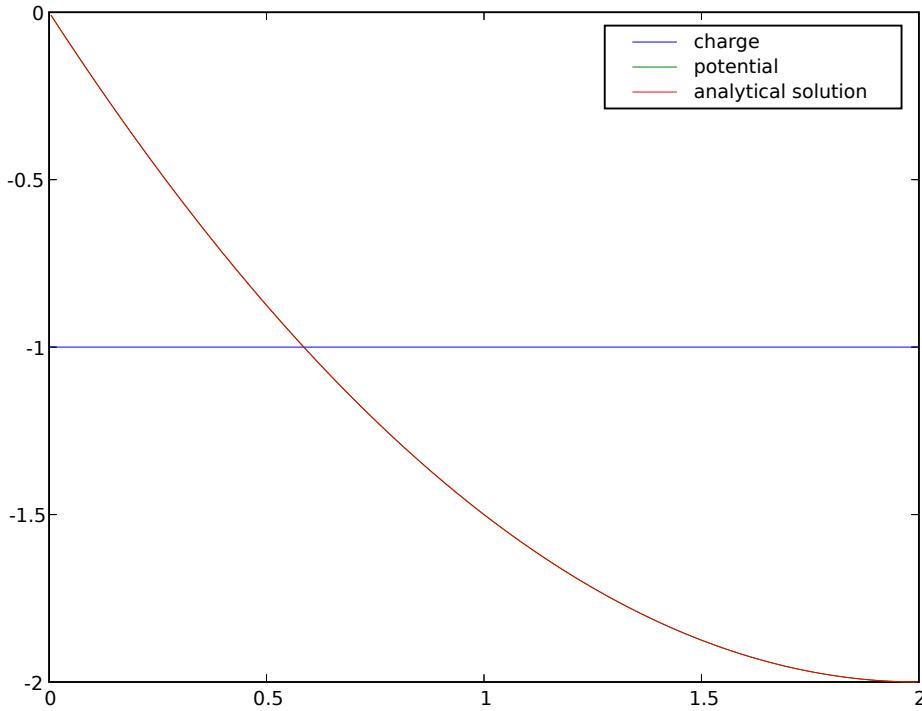
```
>>> x = mesh.getCellCenters()[0]
>>> analytical = CellVariable(mesh=mesh, name="analytical solution",
...                            value=(x**2)/2 - 2*x)
```

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

```
>>> if __name__ == '__main__':
...     viewer = Viewer(vars=(charge, potential, analytical))
...     viewer.plot()
...     raw_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.getCellCenters() [0]
>>> electrons.setValue(0.)
>>> electrons.setValue(1., where=x > L / 2.)
```

and again solve for the electrostatic potential

```
>>> potential.equation.solve(var=potential,
...                                boundaryConditions=bcs)
```

which now has the analytical solution

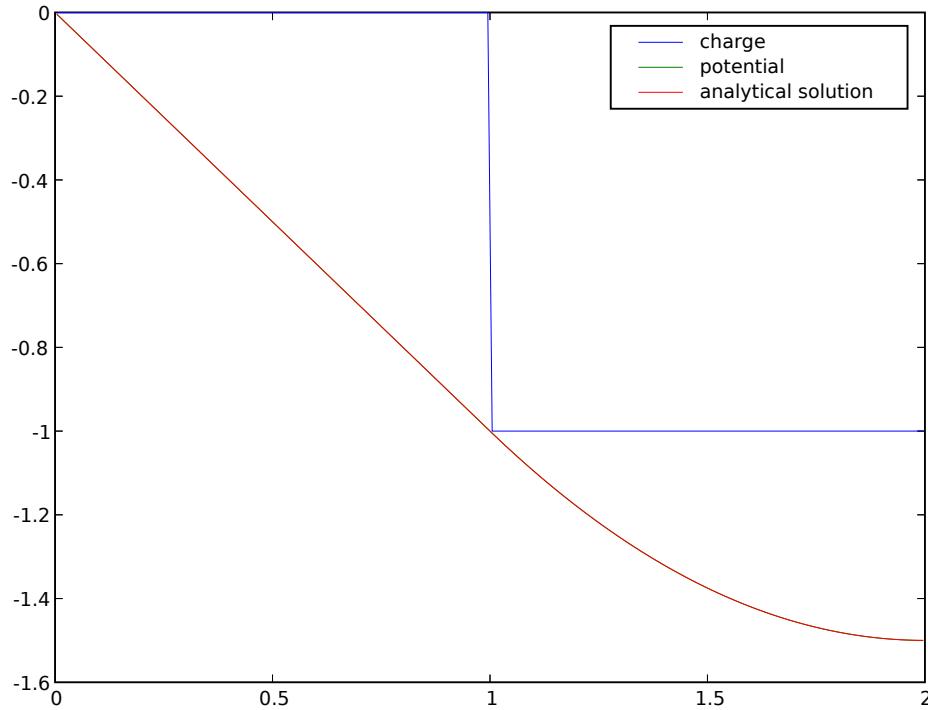
$$\psi(x) = \begin{cases} -x & \text{for } x \leq L/2, \\ \frac{(x-1)^2}{2} - x & \text{for } x > L/2. \end{cases}$$

```
>>> 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

```
>>> if __name__ == '__main__':
...     viewer.plot()
...     raw_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,
...                             boundaryConditions=bcs)
```

which has the analytical solution

$$\psi(x) = \begin{cases} \frac{x^2}{2} - x & \text{for } x \leq L/2, \\ -\frac{1}{2} & \text{for } x > L/2. \end{cases}$$

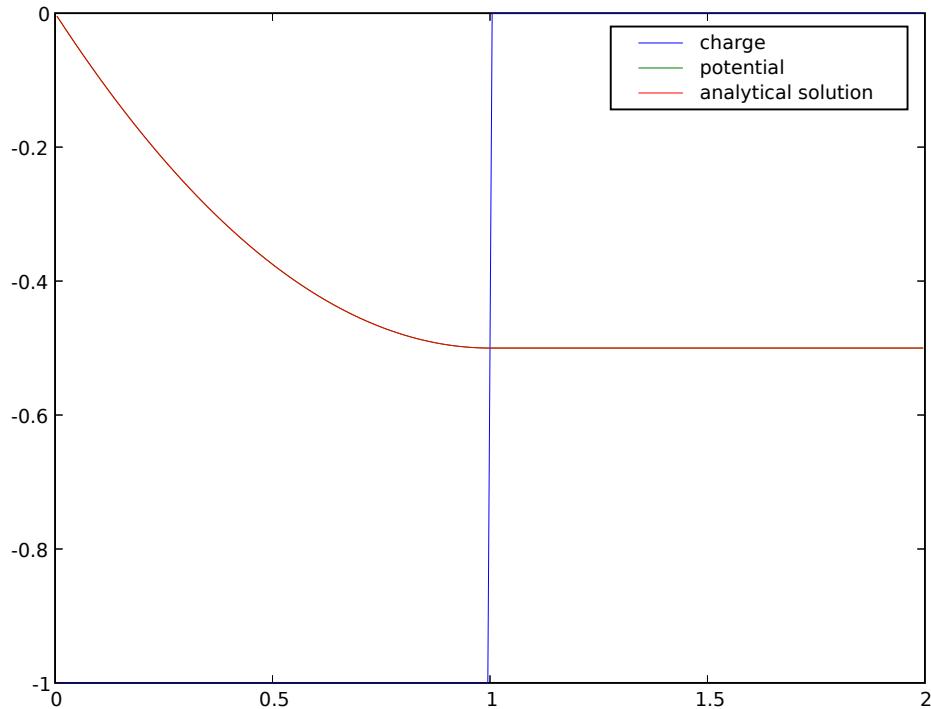
We again verify that the correct equilibrium is attained

```
>>> 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()
```



7.5 examples.diffusion.nthOrder.input4thOrder1D

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 *
```

```
>>> nx = 1000
>>> 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 = (FixedValue(faces=mesh.getFacesLeft(), value=alpha1),
...           FixedFlux(faces=mesh.getFacesRight(), value=alpha2),
...           NthOrderBoundaryCondition(faces=mesh.getFacesLeft(), value=alpha3, order=2),
...           NthOrderBoundaryCondition(faces=mesh.getFacesRight(), value=alpha4, order=3))
```

We initialize the steady-state equation

```
>>> eq = DiffusionTerm(coeff=(1, 1)) == 0
```

and use the `LinearLU Solver` for stability.

We perform one implicit timestep to achieve steady state

```
>>> eq.solve(var=var,
...            boundaryConditions=BCs,
...            solver=DefaultAsymmetricSolver())
```

The analytical solution is:

$$\phi = \frac{\alpha_4}{6}x^3 + \frac{\alpha_3}{2}x^2 + \left(\alpha_2 - \frac{\alpha_4}{2}L^2 - \alpha_3L\right)x + \alpha_1$$

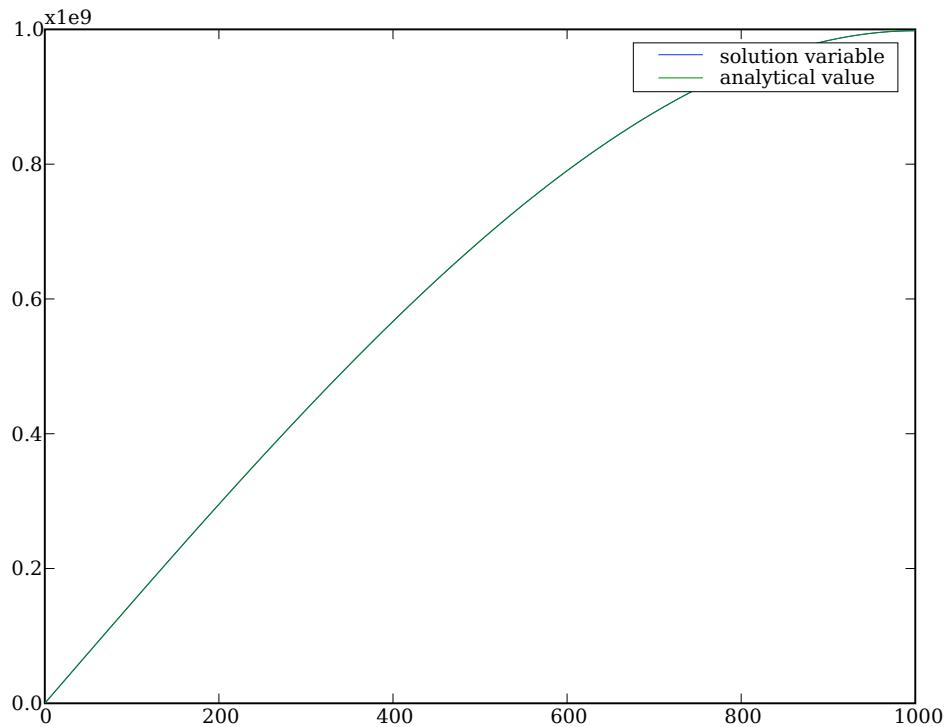
or

```
>>> analytical = CellVariable(mesh=mesh, name='analytical value')
>>> x = mesh.getCellCenters() [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()
```



7.6 examples.diffusion.anisotropy

This example demonstrates how to solve diffusion with an anisotropic coefficient. We wish to solve the following problem.

on a circular domain centred at $(0, 0)$. We can choose an anisotropy ratio of 5 such that

A new matrix is formed by rotating Γ' such that

and

In the case of a point source at $(0, 0)$ a reference solution is given by,

where $\text{left}(X, Y)^T = R \text{left}(x, y)^T$ and Q is the initial mass.

```
>>> from fipy import *
```

Import a mesh previously created using *Gmsh*.

```
>>> mesh = GmshImporter2D(os.path.splitext(__file__)[0] + '.msh')
```

Set the center most cell to have a value.

```
>>> var = CellVariable(mesh=mesh, hasOld=1)
>>> x, y = mesh.getCellCenters()
>>> var[numerix.argmax(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() == DiffusionTerm((gamma,))
>>> if __name__ == '__main__':
...     viewer = Viewer(var, datamin=0.0, datamax=0.001)

>>> mass = float(var.getCellVolumeAverage()) * numerix.sum(mesh.getCellVolumes())
>>> time = 0
>>> dt=0.00025

>>> for i in range(40):
...     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.getCellCenters(), 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))
>>> print max(abs((var - solution) / max(solution))) < 0.05
True
```

Chapter 8

Convection Examples

<code>examples.convection.exponential1D.mesh1D</code>	This example solves the steady-state convection-diffusion equation given by ..
<code>examples.convection.exponential1DSouLike.mesh1D</code>	<code>examples.diffusion.convection.exponential1D.mesh1D</code>
<code>examples.convection.robin</code>	This example demonstrates how to apply a Robin boundary condition to
<code>examples.convection.source</code>	This example solves the equation ..

8.1 examples.convection.exponential1D.mesh1D

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 $\vec{u} = (10,)$, or

```
>>> diffCoeff = 1.  
>>> convCoeff = (10.,)
```

We define a 1D mesh

```
>>> from fipy import *  
  
>>> L = 10.  
>>> nx = 10  
>>> mesh = Grid1D(dx=L / nx, nx=nx)
```

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.  
>>> boundaryConditions = (  
...     FixedValue(faces=mesh.getFacesLeft(), value=valueLeft),  
...     FixedValue(faces=mesh.getFacesRight(), value=valueRight),  
... )
```

The solution variable is initialized to `valueLeft`:

```
>>> var = CellVariable(mesh=mesh, name = "variable")
```

The equation is created with the `DiffusionTerm` and `ExponentialConvectionTerm`. The scheme used by the convection term needs to calculate a Peclet 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, boundaryConditions=boundaryConditions)
```

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]
>>> CC = 1. - exp(-convCoeff[axis] * x / diffCoeff)
>>> DD = 1. - 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()
```

8.2 examples.convection.exponential1DSource.mesh1D

Like `examples.diffusion.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 *
```

```
>>> nx = 1000
>>> L = 10.
>>> mesh = Grid1D(dx=L / 1000, nx=nx)
```

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.
>>> boundaryConditions = (
...     FixedValue(faces=mesh.getFacesRight(), value=valueRight),
...     FixedValue(faces=mesh.getFacesLeft(), value=valueLeft),
... )
```

The solution variable is initialized to `valueLeft`:

```
>>> var = CellVariable(name="variable", mesh=mesh)
```

We define the convection-diffusion equation with source

```
>>> eq = (DiffusionTerm(coeff=diffCoeff)
...         + ExponentialConvectionTerm(coeff=convCoeff)
...         + sourceCoeff)

>>> eq.solve(var=var,
...            boundaryConditions=boundaryConditions,
...            solver=DefaultAsymmetricSolver(tolerance=1.e-15, iterations=10000))
```

and test the solution against the analytical result:

$$\phi = -\frac{S_0 x}{u_x} + \left(1 + \frac{S_0 x}{u_x}\right) \frac{1 - \exp(-u_x x/D)}{1 - \exp(-u_x L/D)}$$

or

```
>>> axis = 0
>>> x = mesh.getCellCenters()[axis]
>>> AA = -sourceCoeff * x / convCoeff[axis]
>>> BB = 1. + sourceCoeff * L / convCoeff[axis]
>>> CC = 1. - exp(-convCoeff[axis] * x / diffCoeff)
>>> DD = 1. - 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()
```

8.3 examples.convection.robin

This example demonstrates how to apply a Robin boundary condition to an advection-diffusion equation. The equation we wish to solve is given by,

$$\begin{aligned} 0 &= \frac{\partial^2 C}{\partial x^2} - P \frac{\partial C}{\partial x} - DC \quad 0 < x < 1 \\ x = 0 : P &= -\frac{\partial C}{\partial x} + PC \\ x = 1 : \frac{\partial C}{\partial x} &= 0 \end{aligned}$$

The analytical solution for this equation is given by,

$$C(x) = \frac{2P \exp\left(\frac{Px}{2}\right) \left[(P+A) \exp\left(\frac{A}{2}(x-1)\right) - (P-A) \exp\left(-\frac{A}{2}(x-1)\right)\right]}{(P+A)^2 \exp\left(\frac{A}{2}\right) - (P-A)^2 \exp\left(-\frac{A}{2}\right)}$$

where

$$A = \sqrt{P + 4D^2}$$

```
>>> from fipy import *
>>> nx = 100
>>> dx = 1.0 / nx

>>> mesh = Grid1D(nx=nx, dx=dx)
>>> C = CellVariable(mesh=mesh)

>>> D = 2.0
>>> P = 3.0
```

From the main equation, the flux into the domain at $x = 0$ is given by

$$\frac{\partial C}{\partial x} - PC$$

Using the boundary condition at $x = 0$ this flux should be equal to $-P$. Setting the $x = 1$ boundary condition to be a fixed value equal to $C(1)$ fixes the edge derivative on both the convection and diffusion terms to be zero.

```
>>> BCs = (FixedFlux(faces=mesh.getFacesLeft(), value=-P),
...           FixedValue(faces=mesh.getFacesRight(), value=C.getFaceValue()))

>>> eq = PowerLawConvectionTerm((P,)) == \
...       DiffusionTerm() - ImplicitSourceTerm(D)

>>> A = numerix.sqrt(P**2 + 4 * D)

>>> x = mesh.getCellCenters()[0]
>>> CAnalytical = CellVariable(mesh=mesh)
>>> CAnalytical.setValue(2 * P * exp(P * x / 2) * ((P + A) * exp(A / 2 * (1 - x)) -
...                                         - (P - A) * exp(-A / 2 * (1 - x))) /
...                                         ((P + A)**2 * exp(A / 2) - (P - A)**2 * exp(-A / 2)))
```

```

>>> if __name__ == '__main__':
...     viewer = Viewer(vars=(C, CAnalytical))

>>> res = 1e+10
>>> while res > 1e-5:
...     res = eq.sweep(var=C, boundaryConditions=BCs)
...     if __name__ == '__main__':
...         viewer.plot()

>>> print C.allclose(CAnalytical, rtol=1.e-3, atol=1.e-3)
True

```

8.4 examples.convection.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 $x = L$ will require the implementation of an outflow boundary condition, which is not currently implemented in FiPy. An `ImplicitSourceTerm` object will be used to represent this term. The derivative of ϕ can be represented by a `ConvectionTerm` with a constant unitary velocity field from left to right. The following is an example code that includes a test against the analytical result.

```

>>> from fipy import *

>>> 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 * exp(-alpha * mesh.getCellCenters()))

>>> if __name__ == "__main__":
...     viewer = Viewer(vars=(phi, solution))
...     viewer.plot()
...     raw_input("press key to continue")

>>> BCs = [FixedValue(faces=mesh.getFacesLeft(), value=phi0)]

```

The RHSBC variable acts like an outflow boundary condition when applied as a source term.

```

>>> RHSBC = (((1,)) * mesh.getFacesRight()).getDivergence()
>>> eq = PowerLawConvectionTerm((1,)) + ImplicitSourceTerm(alpha + RHSBC)
>>> eq.solve(phi, boundaryConditions=BCs)
>>> print numerix.allclose(phi, phi0 * exp(-alpha * mesh.getCellCenters()[0]), atol=1e-3)
True

```

```
>>> if __name__ == "__main__":
...     viewer = Viewer(vars=(phi, solution))
...     viewer.plot()
...     raw_input("finished")
```

Phase Field Examples

`examples.phase.simple` To run this example from the base FiPy directory, type

`examples.phase.binary` It is straightforward to extend a phase field model to include binary alloys.

`examples.phase.quate` 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.

`examples.phase.aniso` To convert a liquid material to a solid, it must be cooled to a temperature below its melting point (known as “undercooling” or “supercooling”).

`examples.phase.impin` In this example we solve a coupled phase and orientation equation on a one dimensional grid.

`examples.phase.impin` In the following examples, we solve the same set of equations as in

9.1 examples.phase.simple

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,

$$\frac{1}{M_\phi} \frac{\partial \phi}{\partial t} = \kappa_\phi \nabla^2 \phi - \frac{\partial f}{\partial \phi} \quad (9.1)$$

For solidification problems, the Helmholtz free energy is frequently given by

$$f(\phi, T) = \frac{W}{2} g(\phi) + L_v \frac{T - T_M}{T_M} p(\phi)$$

where W is the double-well barrier height between phases, L_v is the latent heat, T is the temperature, and T_M is the melting point.

One possible choice for the double-well function is

$$g(\phi) = \phi^2(1 - \phi)^2$$

and for the interpolation function is

$$p(\phi) = \phi^3(6\phi^2 - 15\phi + 10).$$

We create a 1D solution mesh

```
>>> from fipy import *
```

```
>>> 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)
```

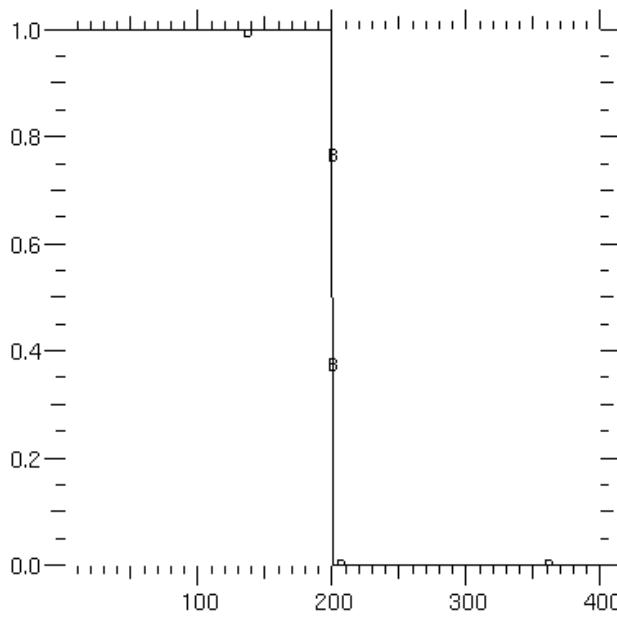
and set a step-function initial condition

$$\phi = \begin{cases} 1 & \text{for } x \leq L/2 \\ 0 & \text{for } x > L/2 \end{cases} \quad \text{at } t = 0$$

```
>>> x = mesh.getCellCenters() [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

```
>>> if __name__ == '__main__':
...     viewer = Viewer(vars = (phase,))
...     viewer.plot()
...     raw_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

$$\phi = \frac{1}{2} \left[1 - \tanh \frac{x - L/2}{2\sqrt{\kappa/W}} \right] \quad (9.2)$$

or

```
>>> x = mesh.getCellCenters() [0]
>>> analyticalArray = 0.5*(1 - tanh((x - L/2) / (2*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

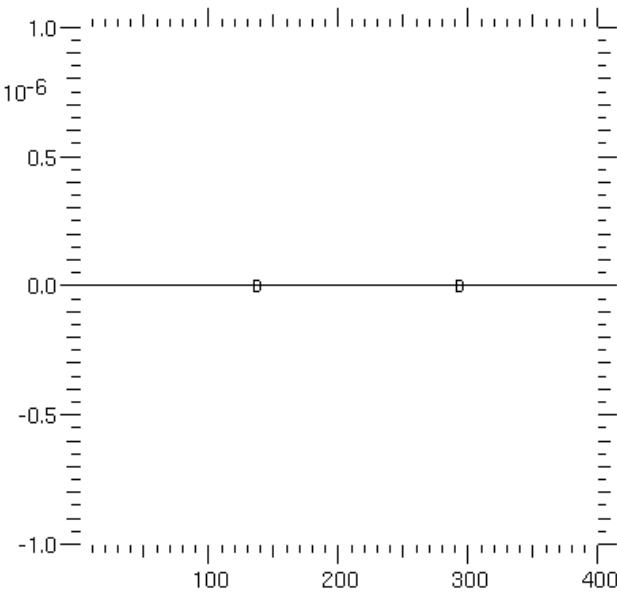
```
>>> mPhi = -((1 - 2 * phase) * W + 30 * phase * (1 - phase) * enthalpy)
>>> S0 = mPhi * phase * (1 - phase)
>>> eq = S0 + DiffusionTerm(coeff=kappa)
```

After solving this equation

```
>>> eq.solve(var = phase)
```

we obtain the surprising result that ϕ is zero everywhere.

```
>>> print phase.allclose(analyticalArray, rtol = 1e-4, atol = 1e-4)
0
>>> if __name__ == '__main__':
...     viewer.plot()
...     raw_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 (9.1)

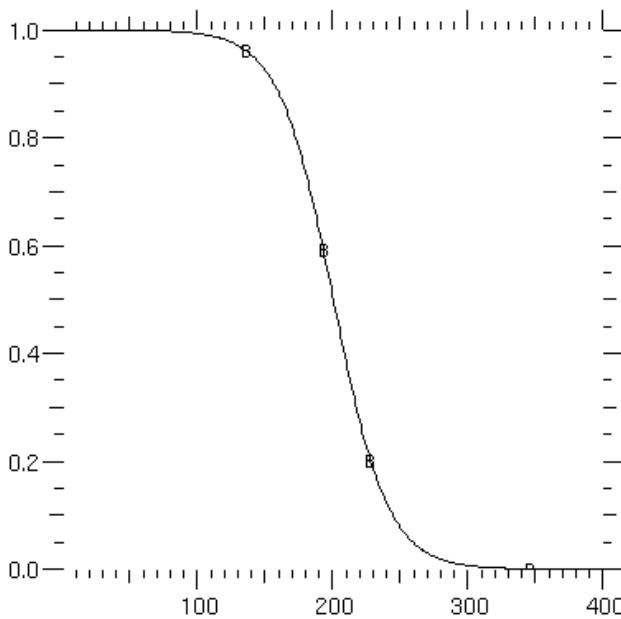
```
>>> eq = TransientTerm() == DiffusionTerm(coeff=kappa) + S0

>>> phase.setValue(1.)
>>> phase.setValue(0., where=x > L/2)

>>> for i in range(13):
...     eq.solve(var = phase)
...     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
>>> if __name__ == '__main__':
...     raw_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 (9.2) is not an exact representation for the solution in a finite domain of length L .

Setting fixed-value boundary conditions of 1 and 0 would still require the relaxation method with the fully explicit source.

Solution performance can be improved if we exploit the dependence of the source on ϕ . By doing so, we can make the source semi-implicit, improving the rate of convergence over the fully explicit approach. The source can only be semi-implicit because we employ sparse linear algebra routines to solve the PDEs, i.e., there is no fully implicit way to represent a term like ϕ^4 in the linear set of equations $M\vec{\phi} - \vec{b} = 0$.

By linearizing a source as $S = S_0 - S_1\phi$, we make it more implicit by adding the coefficient S_1 to the matrix diagonal. For numerical stability, this linear coefficient must never be negative.

There are an infinite number of choices for this linearization, but many do not converge very well. One choice is that used by Ryo Kobayashi:

```
>>> S0 = mPhi * phase * (mPhi > 0)
>>> S1 = mPhi * ((mPhi < 0) - phase)
>>> eq = DiffusionTerm(coeff=kappa) + S0 \
...     + ImplicitSourceTerm(coeff = S1)
```

Note: Because `mPhi` is a variable field, the quantities `(mPhi > 0)` and `(mPhi < 0)` evaluate to variable *fields* of `True` and `False`, instead of single boolean values.

This expression converges to the same value given by the explicit relaxation approach, but in only 8 sweeps (note that because there is no transient term, these sweeps are not time steps, but rather repeated iterations at the same time step to reach a converged solution).

Note: We use `solve()` instead of `sweep()` because we don't care about the residual. Either function would work, but `solve()` is a bit faster.

```
>>> phase.setValue(1.)
>>> phase.setValue(0., where=x > L/2)

>>> for i in range(8):
...     eq.solve(var = phase)
```

```
>>> print phase.allclose(analyticalArray, rtol = 1e-4, atol = 1e-4)
1
>>> if __name__ == '__main__':
...     viewer.plot()
...     raw_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 $S = S_{\text{old}} + \frac{\partial S}{\partial \phi} \Big|_{\text{old}} (\phi - \phi_{\text{old}}) = (S - \frac{\partial S}{\partial \phi} \phi)_{\text{old}} + \frac{\partial S}{\partial \phi} \Big|_{\text{old}} \phi$.

Now, if our source term is represented by $S = S_0 + S_1\phi$, then $S_1 = \frac{\partial S}{\partial \phi} \Big|_{\text{old}}$ and $S_0 = (S - \frac{\partial S}{\partial \phi} \phi)_{\text{old}} = S_{\text{old}} - S_1\phi_{\text{old}}$. In this way, the linearized source will be tangent to the curve of the actual source as a function of the dependent variable.

For our source, $S = m_\phi \phi(1 - \phi)$,

$$\frac{\partial S}{\partial \phi} = \frac{\partial m_\phi}{\partial \phi} \phi(1 - \phi) + m_\phi(1 - 2\phi)$$

and

$$\frac{\partial m_\phi}{\partial \phi} = 2W - 30(1 - 2\phi)L \frac{T - T_M}{T_M},$$

or

```
>>> dmPhidPhi = 2 * W - 30 * (1 - 2 * phase) * enthalpy
>>> S1 = dmPhidPhi * phase * (1 - phase) + mPhi * (1 - 2 * phase)
>>> S0 = mPhi * phase * (1 - phase) - S1 * phase
>>> eq = DiffusionTerm(coeff=kappa) + S0 \
...     + ImplicitSourceTerm(coeff = S1)
```

Using this scheme, where the coefficient of the implicit source term is tangent to the source, we reach convergence in only 5 sweeps

```
>>> phase.setValue(1.)
>>> phase.setValue(0., where=x > L/2)

>>> for i in range(5):
...     eq.solve(var = phase)
>>> print phase.allclose(analyticalArray, rtol = 1e-4, atol = 1e-4)
1
>>> if __name__ == '__main__':
...     viewer.plot()
...     raw_input("Tangent, semi-implicit. Press <return> to proceed...")
```

Although, for this simple problem, there is no appreciable difference in run-time between the fully explicit source and the optimized semi-implicit source, the benefit of 60% fewer sweeps should be obvious for larger systems and longer iterations.

This example has focused on just the region of the phase field interface in equilibrium. Problems of interest, though, usually involve the dynamics of one phase transforming to another. To that end, let us recast the problem using physical parameters and dimensions. We'll need a new mesh

```
>>> nx = 400
>>> dx = 5e-6 # cm
>>> L = nx * dx
```

```
>>> mesh = Grid1D(dx = dx, nx = nx)
```

and thus must redeclare ϕ on the new mesh

```
>>> phase = CellVariable(name="phase",
...                         mesh=mesh,
...                         hasOld=1)
>>> x = mesh.getCellCenters() [0]
>>> phase.setValue(1.)
>>> phase.setValue(0., where=x > L/2)
```

We choose the parameter values appropriate for nickel, given in [Warren:1995]

```
>>> Lv = 2350 # J / cm**3
>>> Tm = 1728. # K
>>> T = Variable(value=Tm)
>>> enthalpy = Lv * (T - Tm) / Tm # J / cm**3
```

The parameters of the phase field model can be related to the surface energy σ and the interfacial thickness δ by

$$\begin{aligned}\kappa &= 6\sigma\delta \\ W &= \frac{6\sigma}{\delta} \\ M_\phi &= \frac{T_m\beta}{6L\delta}.\end{aligned}$$

We take $\delta \approx \Delta x$.

```
>>> delta = 1.5 * dx
>>> sigma = 3.7e-5 # J / cm**2
>>> beta = 0.33 # cm / (K s)
>>> kappa = 6 * sigma * delta # J / cm
>>> W = 6 * sigma / delta # J / cm**3
>>> Mphi = Tm * beta / (6. * Lv * delta) # cm**3 / (J s)

>>> analyticalArray = CellVariable(name="tanh", mesh=mesh,
...                                   value=0.5 * (1 - tanh((x - (L / 2. + L / 10.)) /
...                                   (2 * delta))))
```

and make a new viewer

```
>>> if __name__ == '__main__':
...     viewer2 = Viewer(vars = (phase, analyticalArray))
...     viewer2.plot()
```

Now we can redefine the transient phase field equation, using the optimal form of the source term shown above

```
>>> mPhi = -((1 - 2 * phase) * W + 30 * phase * (1 - phase) * enthalpy)
>>> dmPhidPhi = 2 * W - 30 * (1 - 2 * phase) * enthalpy
>>> S1 = dmPhidPhi * phase * (1 - phase) + mPhi * (1 - 2 * phase)
>>> S0 = mPhi * phase * (1 - phase) - S1 * phase
>>> eq = TransientTerm(coeff=1/Mphi) == DiffusionTerm(coeff=kappa) \
... + S0 + ImplicitSourceTerm(coeff = S1)
```

In order to separate the effect of forming the phase field interface from the kinetics of moving it, we first equilibrate at the melting point. We now use the `sweep()` method instead of `solve()` because we require the residual.

```
>>> timeStep = 1e-6
>>> for i in range(10):
...     phase.updateOld()
...     res = 1e+10
...     while res > 1e-5:
...         res = eq.sweep(var=phase, dt=timeStep)
>>> if __name__ == '__main__':
...     viewer2.plot()
```

and then quench by 1 K

```
>>> T.setValue(T() - 1)
```

In order to have a stable numerical solution, the interface must not move more than one grid point per time step, we thus set the timestep according to the grid spacing Δx , the linear kinetic coefficient β , and the undercooling $|T_m - T|$. Again we use the `sweep()` method as a replacement for `solve()`.

```
>>> velocity = beta * abs(Tm - T()) # cm / s
>>> timeStep = .1 * dx / velocity # s
>>> elapsed = 0
>>> while elapsed < 0.1 * L / 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.

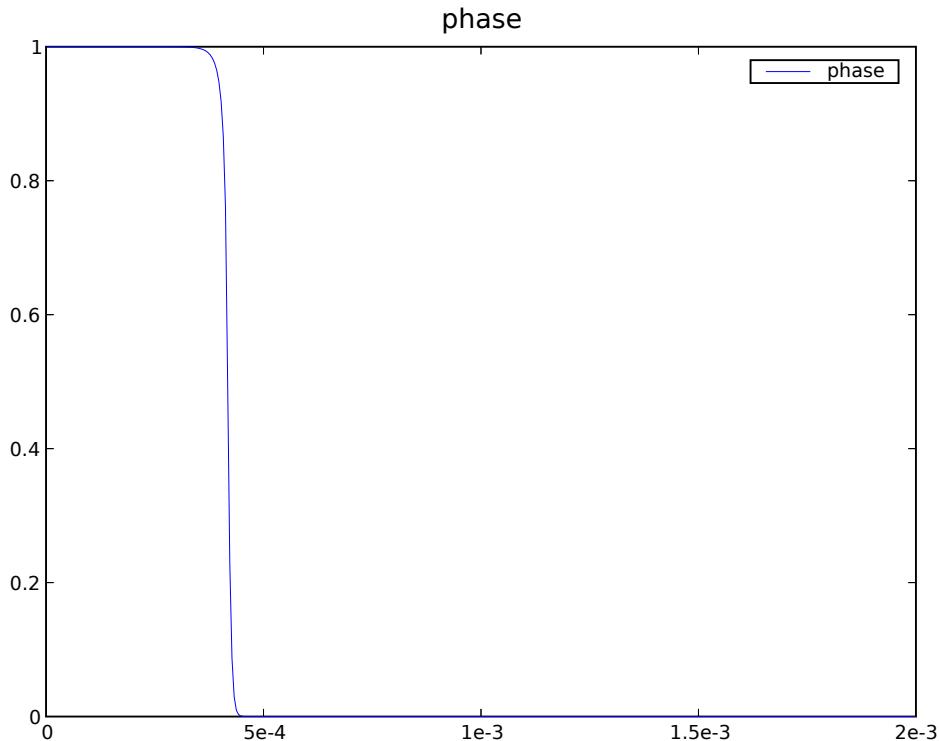
```
>>> print phase.allclose(analyticalArray, rtol = 5, atol = 2e-3)
1
```

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 - tanh((x - V * t - L / 2.) / (2*d)))
...     from scipy.optimize import leastsq
...     x = mesh.getCellCenters()[0]
...     (V_fit, d_fit), msg = leastsq(tanhResiduals, [L/2., delta],
...                                   args=(phase.getGlobalValue(), x.getGlobalValue(), 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) < 3.3e-2
True
```

```
>>> print abs(1 - d_fit / delta) < 2e-2
True

>>> if __name__ == '__main__':
...     raw_input("Dimensional, semi-implicit. Press <return> to proceed...")
```



9.2 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 *
>>> 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 [BoettgerReview:2002] [McFaddenReview:2002] [Wheeler:1992]

$$\mathcal{F}(\phi, C, T) = \int_{\mathcal{V}} \left\{ f(\phi, C, T) + \frac{\kappa_\phi}{2} |\nabla \phi|^2 + \frac{\kappa_C}{2} |\nabla C|^2 \right\} dV$$

over the volume \mathcal{V} as a function of phase ϕ ¹

```
>>> phase = CellVariable(name="phase", mesh=mesh, hasOld=1)
```

composition C

```
>>> C = CellVariable(name="composition", mesh=mesh, hasOld=1)
```

and temperature T ²

```
>>> T = Variable(name="temperature")
```

Frequently, the gradient energy term in concentration is ignored and we can derive governing equations

$$\frac{\partial \phi}{\partial t} = M_\phi \left(\kappa_\phi \nabla^2 \phi - \frac{\partial f}{\partial \phi} \right) \quad (9.3)$$

for phase and

$$\frac{\partial C}{\partial t} = \nabla \cdot \left(M_C \nabla \frac{\partial f}{\partial C} \right) \quad (9.4)$$

for solute.

The free energy density $f(\phi, C, T)$ can be constructed in many different ways. One approach is to construct free energy densities for each of the pure components, as functions of phase, *e.g.*

$$f_A(\phi, T) = p(\phi) f_A^S(T) + (1 - p(\phi)) f_A^L(T) + \frac{W_A}{2} g(\phi)$$

where $f_A^L(T)$, $f_B^L(T)$, $f_A^S(T)$, and $f_B^S(T)$ are the free energy densities of the pure components. There are a variety of choices for the interpolation function $p(\phi)$ and the barrier function $g(\phi)$,

such as those shown in mod:*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,

$$f(\phi, C, T) = (1 - C)f_A(\phi, T) + Cf_B(\phi, T) + RT [(1 - C)\ln(1 - C) + C\ln C] + C(1 - C)[\Omega_{SP}(\phi) + \Omega_L(1 - p(\phi))]$$

where

```
>>> R = 8.314 # J / (mol K)
```

¹ We will find that we need to “sweep” this non-linear problem (see *e.g.* the composition-dependent diffusivity example in *examples.diffusion.mesh1D*), so we declare ϕ and C to retain an “old” value.

² we are going to want to examine different temperatures in this example, so we declare T as a *Variable*

is the gas constant and Ω_S and Ω_L are the regular solution interaction parameters for solid and liquid.

Another approach is useful when the free energy densities $f^L(C, T)$ and $f^S(C, T)$ of the alloy in the solid and liquid phases are known. This might be the case when the two different phases have different thermodynamic models or when one or both is obtained from a Calphad code. In this case, we can construct

$$f(\phi, C, T) = p(\phi)f^S(C, T) + (1 - p(\phi))f^L(C, T) + \left[(1 - C)\frac{W_A}{2} + C\frac{W_B}{2} \right]g(\phi).$$

When the thermodynamic models are the same in both phases, both approaches should yield the same result.

We choose the first approach and make the simplifying assumptions of an ideal solution and that

$$\begin{aligned} f_A^L(T) &= 0 \\ f_A^S(T) - f_A^L(T) &= \frac{L_A(T - T_M^A)}{T_M^A} \end{aligned}$$

and likewise for component B .

```
>>> LA = 2350. # J / cm**3
>>> LB = 1728. # J / cm**3
>>> TmA = 1728. # K
>>> TmB = 1358. # K

>>> enthalpyA = LA * (T - TmA) / TmA
>>> enthalpyB = LB * (T - TmB) / TmB
```

This relates the difference between the free energy densities of the pure solid and pure liquid phases to the latent heat L_A and the pure component melting point T_M^A , such that

$$f_A(\phi, T) = \frac{L_A(T - T_M^A)}{T_M^A}p(\phi) + \frac{W_A}{2}g(\phi).$$

With these assumptions

$$\begin{aligned} \frac{\partial f}{\partial \phi} &= (1 - C)\frac{\partial f_A}{\partial \phi} + C\frac{\partial f_B}{\partial \phi} \\ &= \left\{ (1 - C)\frac{L_A(T - T_M^A)}{T_M^A} + C\frac{L_B(T - T_M^B)}{T_M^B} \right\}p'(\phi) + \left\{ (1 - C)\frac{W_A}{2} + C\frac{W_B}{2} \right\}g'(\phi) \end{aligned}$$

and

$$\begin{aligned} \frac{\partial f}{\partial C} &= \left[f_B(\phi, T) + \frac{RT}{V_m} \ln C \right] - \left[f_A(\phi, T) + \frac{RT}{V_m} \ln(1 - C) \right] \\ &= [\mu_B(\phi, C, T) - \mu_A(\phi, C, T)]/V_m \end{aligned}$$

where μ_A and μ_B are the classical chemical potentials for the binary species. $p'(\phi)$ and $g'(\phi)$ are the partial derivatives of p and g with respect to ϕ

```
>>> def pPrime(phi):
...     return 30. * g(phi)

>>> def gPrime(phi):
...     return 2. * phi * (1 - phi) * (1 - 2 * phi)
```

V_m is the molar volume, which we take to be independent of concentration and phase

```
>>> Vm = 7.42 # cm**3 / mol
```

On comparison with `examples.phase.simple`, we can see that the present form of the phase field equation is identical to the one found earlier, with the source now composed of the concentration-weighted average of the source for either pure component. We let the pure component barriers equal the previous value

```
>>> deltaA = deltaB = 1.5 * dx
>>> sigmaA = 3.7e-5 # J / cm**2
>>> sigmaB = 2.9e-5 # J / cm**2
>>> betaA = 0.33 # cm / (K s)
>>> betaB = 0.39 # cm / (K s)
>>> kappaA = 6 * sigmaA * deltaA # J / cm
>>> kappaB = 6 * sigmaB * deltaB # J / cm
>>> WA = 6 * sigmaA / deltaA # J / cm**3
>>> WB = 6 * sigmaB / deltaB # J / cm**3
```

and define the averages

```
>>> W = (1 - C) * WA / 2. + C * WB / 2.
>>> enthalpy = (1 - C) * enthalpyA + C * enthalpyB
```

We can now linearize the source exactly as before

```
>>> mPhi = -((1 - 2 * phase) * W + 30 * phase * (1 - phase) * enthalpy)
>>> dmPhidPhi = 2 * W - 30 * (1 - 2 * phase) * enthalpy
>>> S1 = dmPhidPhi * phase * (1 - phase) + mPhi * (1 - 2 * phase)
>>> S0 = mPhi * phase * (1 - phase) - S1 * phase
```

Using the same gradient energy coefficient and phase field mobility

```
>>> kappa = (1 - C) * kappaA + C * kappaB
>>> Mphi = TmA * betaA / (6 * LA * deltaA)
```

we define the phase field equation

```
>>> phaseEq = TransientTerm(1/Mphi) == DiffusionTerm(coeff=kappa) \
...     + S0 + ImplicitSourceTerm(coeff=S1)
```

When coding explicitly, it is typical to simply write a function to evaluate the chemical potentials μ_A and μ_B and then perform the finite differences necessary to calculate their gradient and divergence, e.g.:

```
def deltaChemPot(phase, C, T):
    return ((Vm * (enthalpyB * p(phase) + WA * g(phase)) + R * T * log(1 - C)) -
            (Vm * (enthalpyA * p(phase) + WA * g(phase)) + R * T * log(C)))

for j in range(faces):
    flux[j] = ((Mc[j+.5] + Mc[j-.5]) / 2) \
              * (deltaChemPot(phase[j+.5], C[j+.5], T) \
                 - deltaChemPot(phase[j-.5], C[j-.5], T)) / dx

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).getFaceGrad()
eq = TransientTerm() == flux.getDivergence()
```

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. (9.4) to the canonical form of Eq. (??), hence we must expand Eq. (9.2) 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. (9.4) to

$$\begin{aligned} \frac{\partial C}{\partial t} &= \nabla \cdot \left(M_C \nabla \left\{ \frac{RT}{V_m} [\ln C - \ln(1 - C)] \right\} \right) \\ &= \nabla \cdot \left[\frac{M_C RT}{C(1 - C)V_m} \nabla C \right] \end{aligned}$$

and, by comparison with Fick's second law

$$\frac{\partial C}{\partial t} = \nabla \cdot [D \nabla C],$$

we can associate the mobility M_C with the intrinsic diffusivity D by $M_C \equiv DC(1 - C)V_m/RT$ and write Eq. (9.4) 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

```
>>> Dl = Variable(value=1e-5) # cm**2 / s
>>> Ds = Variable(value=1e-9) # cm**2 / s
>>> D = (Dl - Ds) * phase.getArithmeticFaceValue() + Dl

>>> phaseTransformationVelocity = \
...     ((enthalpyB - enthalpyA) * p(phase).getFaceGrad() \
...     + 0.5 * (WB - WA) * g(phase).getFaceGrad()) \
...     * D * (1. - C).getHarmonicFaceValue() * 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.getCellCenters() [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. - exp(-enthalpyA * Vm / (R * T))) \
...     / (exp(-enthalpyB * Vm / (R * T)) - exp(-enthalpyA * Vm / (R * T)))
>>> Cs = exp(-enthalpyB * Vm / (R * T)) * Cl
```

The phase fraction is predicted by the lever rule

```
>>> Cavg = C.getCellVolumeAverage()
>>> 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 [array(enthalpyA * Vm + R * T * log(1 - C[0]) - R * T * log(1 - C[1])),
...             array(enthalpyB * Vm + R * T * log(C[0]) - R * T * 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 * 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.getCellCenters() [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

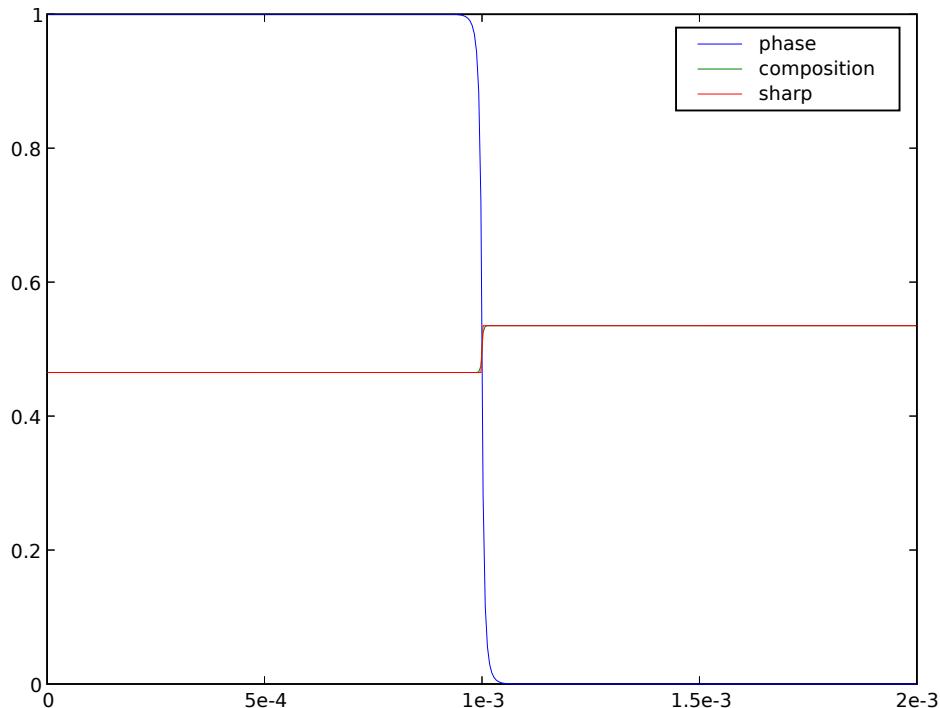
```
>>> dt = 1.e2
```

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)  
>>> if __name__ == '__main__':  
...     viewer.plot()  
...     raw_input("stationary phase field")
```



We verify that the bulk phases have shifted to the predicted solidus and liquidus compositions

```
>>> X = mesh.getFaceCenters() [0]  
>>> print Cs.allclose(C.getFaceValue() [X==0], atol=2e-4)  
True  
>>> print Cl.allclose(C.getFaceValue() [X==L], atol=2e-4)  
True
```

and that the phase fraction remains unchanged

```
>>> print fraction.allclose(phase.getCellVolumeAverage(), 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

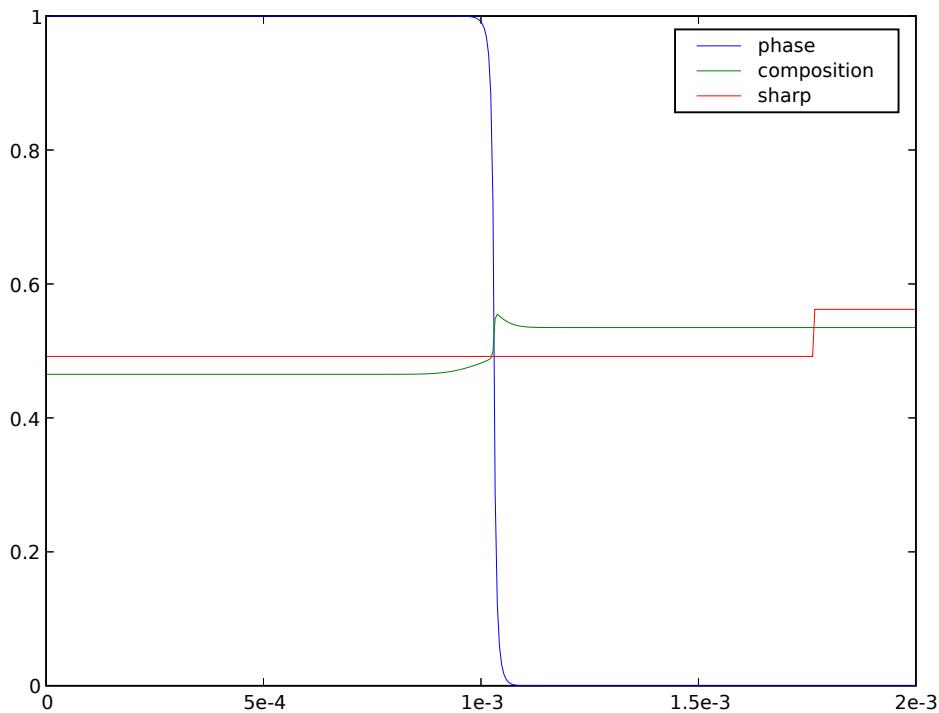
>>> sharp.setValue(Cs, where=x < L * fraction)
>>> sharp.setValue(Cl, where=x >= L * fraction)
```

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).

```
>>> dt = 1.e-6

>>> for i in range(100):
...     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()

>>> if __name__ == '__main__':
...     raw_input("moving phase field")
```



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.

9.3 examples.phase.quaternary

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 *

>>> nx = 400
>>> dx = 0.01
>>> L = nx * dx
>>> mesh = Grid1D(dx = dx, nx = nx)
```

We consider a free energy density $f(\phi, C_0, \dots, C_N, T)$ that is a function of phase ϕ

```
>>> phase = CellVariable(mesh=mesh, name='phase', value=1., hasOld=1)
```

interstitial components $C_0 \dots C_M$

```
>>> interstitials = [
...     CellVariable(mesh=mesh, name='C0', hasOld=1)
... ]
```

substitutional components $C_{M+1} \dots C_{N-1}$

```
>>> substitutionals = [
...     CellVariable(mesh=mesh, name='C1', hasOld=1),
...     CellVariable(mesh=mesh, name='C2', hasOld=1),
... ]
```

a “solvent” C_N that is constrained by the concentrations of the other substitutional species, such that $C_N = 1 - \sum_{j=M}^{N-1} C_j$,

```
>>> solvent = 1
>>> for Cj in substitutionals:
...     solvent -= Cj
>>> solvent.name = 'CN'
```

and temperature T

```
>>> T = 1000
```

The free energy density of such a system can be written as

$$f(\phi, C_0, \dots, C_N, T) = \sum_{j=0}^N C_j \left[\mu_j^\circ(\phi, T) + RT \ln \frac{C_j}{\rho} \right]$$

where

```
>>> R = 8.314 # J / (mol K)
```

is the gas constant. As in the binary case,

$$\mu_j^\circ(\phi, T) = p(\phi)\mu_j^{\circ S}(T) + (1 - p(\phi))\mu_j^{\circ L}(T) + \frac{W_j}{2}g(\phi)$$

is constructed with the free energies of the pure components in each phase, given the “tilting” function

```
>>> def p(phi):
...     return phi**3 * (6 * phi**2 - 15 * phi + 10)
```

and the “double well” function

```
>>> def g(phi):
...     return (phi * (1 - phi))**2
```

We consider a very simplified model that has partial molar volumes $\bar{V}_0 = \dots = \bar{V}_M = 0$ for the “interstitials” and $\bar{V}_{M+1} = \dots = \bar{V}_N = 1$ for the “substitutionals”. This approximation has been used in a number of models where density effects are ignored, including the treatment of electrons in electrodeposition processes [ElPhFI] [ElPhFII]. Under these constraints

$$\begin{aligned}\frac{\partial f}{\partial \phi} &= \sum_{j=0}^N C_j \frac{\partial f_j}{\partial \phi} \\ &= \sum_{j=0}^N C_j \left[\mu_j^{\circ SL}(T)p'(\phi) + \frac{W_j}{2}g'(\phi) \right] \\ \frac{\partial f}{\partial C_j} &= \left[\mu_j^\circ(\phi, T) + RT \ln \frac{C_j}{\rho} \right] \\ &= \mu_j(\phi, C_j, T) \quad \text{for } j = 0 \dots M\end{aligned}$$

and

$$\begin{aligned}\frac{\partial f}{\partial C_j} &= \left[\mu_j^\circ(\phi, T) + RT \ln \frac{C_j}{\rho} \right] - \left[\mu_N^\circ(\phi, T) + RT \ln \frac{C_N}{\rho} \right] \\ &= [\mu_j(\phi, C_j, T) - \mu_N(\phi, C_N, T)] \quad \text{for } j = M+1 \dots N-1\end{aligned}$$

where $\mu_j^{\circ SL}(T) \equiv \mu_j^{\circ S}(T) - \mu_j^{\circ L}(T)$ and where μ_j is the classical chemical potential of component j for the binary species and $\rho = 1 + \sum_{j=0}^M C_j$ is the total molar density.

```
>>> rho = 1.
>>> for Cj in interstitials:
...     rho += Cj
```

$p'(\phi)$ and $g'(\phi)$ are the partial derivatives of p and g with respect to ϕ

```
>>> def pPrime(phi):
...     return 30. * g(phi)

>>> def gPrime(phi):
...     return 2. * phi * (1 - phi) * (1 - 2 * phi)
```

We “cook” the standard potentials to give the desired solid and liquid concentrations, with a solid phase rich in interstitials and the solvent and a liquid phase rich in the two substitutional species.

```
>>> interstitials[0].S = 0.3
>>> interstitials[0].L = 0.4
>>> substitutionals[0].S = 0.4
>>> substitutionals[0].L = 0.3
>>> substitutionals[1].S = 0.2
>>> substitutionals[1].L = 0.1
>>> solvent.S = 1.
>>> solvent.L = 1.
>>> for Cj in substitutionals:
...     solvent.S -= Cj.S
...     solvent.L -= Cj.L

>>> rhoS = rhoL = 1.
>>> for Cj in interstitials:
...     rhoS += Cj.S
...     rhoL += Cj.L

>>> for Cj in interstitials + substitutionals + [solvent]:
...     Cj.standardPotential = R * T * (log(Cj.L/rhoL) - log(Cj.S/rhoS))

>>> for Cj in interstitials:
...     Cj.diffusivity = 1.
...     Cj.barrier = 0.

>>> for Cj in substitutionals:
...     Cj.diffusivity = 1.
...     Cj.barrier = R * T

>>> solvent.barrier = R * T
```

We create the phase equation

$$\frac{1}{M_\phi} \frac{\partial \phi}{\partial t} = \kappa_\phi \nabla^2 \phi - \sum_{j=0}^N C_j \left[\mu_j^{\circ SL}(T) p'(\phi) + \frac{W_j}{2} g'(\phi) \right]$$

with a semi-implicit source just as in `examples.phase.simple` and `examples.phase.binary`

```
>>> enthalpy = 0.
>>> barrier = 0.
>>> for Cj in interstitials + substitutionals + [solvent]:
...     enthalpy += Cj * Cj.standardPotential
...     barrier += Cj * Cj.barrier

>>> mPhi = -((1 - 2 * phase) * barrier + 30 * phase * (1 - phase) * enthalpy)
>>> dmPhidPhi = 2 * barrier - 30 * (1 - 2 * phase) * enthalpy
>>> S1 = dmPhidPhi * phase * (1 - phase) + mPhi * (1 - 2 * phase)
>>> S0 = mPhi * phase * (1 - phase) - S1 * phase
```

```
>>> phase.mobility = 1.
>>> phase.gradientEnergy = 25
>>> phase.equation = TransientTerm(coeff=1/phase.mobility) \
...     == DiffusionTerm(coeff=phase.gradientEnergy) \
...     + S0 + ImplicitSourceTerm(coeff = S1)
```

We could construct the diffusion equations one-by-one, in the manner of `examples.phase.binary`, but it is better to take advantage of the full scripting power of the Python language, where we can easily loop over components or even make “factory” functions if we desire. For the interstitial diffusion equations, we arrange in canonical form as before:

$$\underbrace{\frac{\partial C_j}{\partial t}}_{\text{transient}} = \underbrace{D_j \nabla^2 C_j}_{\text{diffusion}} + D_j \nabla \cdot \underbrace{\frac{C_j}{1 + \sum_{k=0}^M C_k} \left\{ \underbrace{\frac{\rho}{RT} \left[\mu_j^{\circ SL} \nabla p(\phi) + \frac{W_j}{2} \nabla g(\phi) \right]}_{\text{phase transformation}} - \underbrace{\sum_{i=0}^M \nabla C_i}_{\text{counter diffusion}} \right\}}_{\text{convection}}$$

```
>>> for Cj in interstitials:
...     phaseTransformation = (rho.getHarmonicFaceValue() / (R * T)) \
...         * (Cj.standardPotential * p(phase).getFaceGrad()
...             + 0.5 * Cj.barrier * g(phase).getFaceGrad())
...
...     CkSum = CellVariable(mesh=mesh, value=0.)
...     for Ck in [Ck for Ck in interstitials if Ck is not Cj]:
...         CkSum += Ck
...
...     counterDiffusion = CkSum.getFaceGrad()
...
...     convectionCoeff = counterDiffusion + phaseTransformation
...     convectionCoeff *= (Cj.diffusivity
...                         / (1. + CkSum.getHarmonicFaceValue()))
...
...     Cj.equation = (TransientTerm()
...                   == DiffusionTerm(coeff=Cj.diffusivity)
...                   + PowerLawConvectionTerm(coeff=convectionCoeff))
```

The canonical form of the substitutional diffusion equations is

$$\underbrace{\frac{\partial C_j}{\partial t}}_{\text{transient}} = \underbrace{D_j \nabla^2 C_j}_{\text{diffusion}} + D_j \nabla \cdot \underbrace{\frac{C_j}{1 - \sum_{k=2}^{n-1} C_k} \left\{ \underbrace{\frac{C_N}{RT} \left[(\mu_j^{\circ SL} - \mu_N^{\circ SL}) \nabla p(\phi) + \frac{W_j - W_N}{2} \nabla g(\phi) \right]}_{\text{phase transformation}} + \underbrace{\sum_{i=M+1}^N \nabla C_i}_{\text{counter diffusion}} \right\}}_{\text{convection}}$$

```
>>> for Cj in substitutionals:
...     phaseTransformation = (solvent.getHarmonicFaceValue() / (R * T)) \
...         * ((Cj.standardPotential - solvent.standardPotential) * p(phase).getFaceGrad() \
...             + 0.5 * (Cj.barrier - solvent.barrier) * g(phase).getFaceGrad())
...
...     CkSum = CellVariable(mesh=mesh, value=0.)
...     for Ck in [Ck for Ck in substitutionals if Ck is not Cj]:
...         CkSum += Ck
...
...     counterDiffusion = CkSum.getFaceGrad()
...
...     convectionCoeff = counterDiffusion + phaseTransformation
...     convectionCoeff *= (Cj.diffusivity \
...         / (1. - CkSum.getHarmonicFaceValue())))
...
...     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.getCellCenters()[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

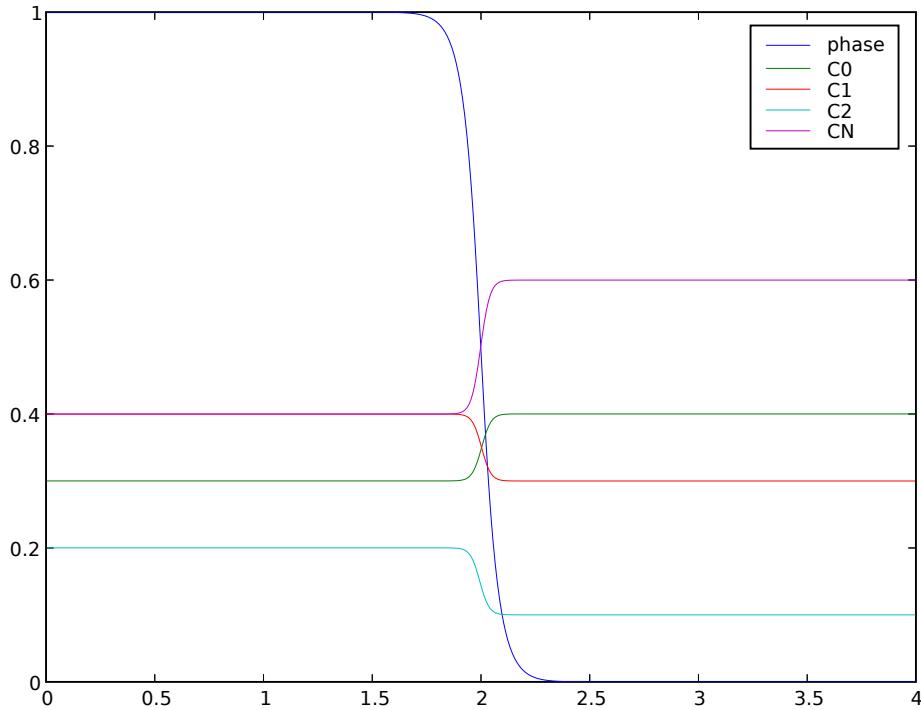
```
>>> solver = DefaultAsymmetricSolver(tolerance=1e-10)

>>> dt = 10000
>>> for i in range(5):
...     for field in [phase] + substitutionals + interstitials:
...         field.updateOld()
...     phase.equation.solve(var = phase, dt = dt)
...     for field in substitutionals + interstitials:
...         field.equation.solve(var = field,
```

```

...
        dt = dt,
        solver = solver)
...
if __name__ == '__main__':
    viewer.plot()

```



We can confirm that the far-field phases have remained separated

```

>>> X = mesh.getFaceCenters() [0]
>>> print allclose(phase.getFaceValue() [X==0], 1.0, rtol = 1e-5, atol = 1e-5)
True
>>> print allclose(phase.getFaceValue() [X==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 &= allclose(Cj.getFaceValue() [X==0], Cj.S, rtol = 3e-3, atol = 3e-3).getValue()
...     equilibrium &= allclose(Cj.getFaceValue() [X==L], Cj.L, rtol = 3e-3, atol = 3e-3).getValue()
>>> print equilibrium
True

```

9.4 examples.phase.anisotropy

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 [WarrenPolycrystal] 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 *
>>> dx = dy = 0.025
>>> if __name__ == '__main__':
...     nx = ny = 500
... else:
...     nx = ny = 20
>>> mesh = Grid2D(dx=dx, dy=dy, nx=nx, ny=ny)
```

and we'll take fixed timesteps

```
>>> dt = 5e-4
```

We consider the simultaneous evolution of a “phase field” variable ϕ (taken to be 0 in the liquid phase and 1 in the solid)

```
>>> phase = CellVariable(name=r'$\phi$', mesh=mesh, hasOld=True)
```

and a dimensionless undercooling ΔT ($\Delta T = 0$ at the melting point)

```
>>> dT = CellVariable(name=r'$\Delta T$', mesh=mesh, hasOld=True)
```

The `hasOld` flag causes the storage of the value of variable from the previous timestep. This is necessary for solving equations with non-linear coefficients or for coupling between PDEs.

The governing equation for the temperature field is the heat flux equation, with a source due to the latent heat of solidification

$$\frac{\partial \Delta T}{\partial t} = D_T \nabla^2 \Delta T + \frac{\partial \phi}{\partial t}$$

```
>>> DT = 2.25
>>> heatEq = (TransientTerm()
...             == DiffusionTerm(DT)
...             + (phase - phase.getOld()) / dt)
```

The governing equation for the phase field is

$$\tau_\phi \frac{\partial \phi}{\partial t} = \nabla \cdot D \nabla \phi + \phi(1-\phi)m(\phi, \Delta T)$$

where

$$m(\phi, \Delta T) = \phi - \frac{1}{2} - \frac{\kappa_1}{\pi} \arctan(\kappa_2 \Delta T)$$

represents a source of anisotropy. The coefficient D is an anisotropic diffusion tensor in two dimensions

$$D = \alpha^2 (1 + c\beta) \begin{bmatrix} 1 + c\beta & -c \frac{\partial \beta}{\partial \psi} \\ c \frac{\partial \beta}{\partial \psi} & 1 + c\beta \end{bmatrix}$$

where $\beta = \frac{1-\Phi^2}{1+\Phi^2}$, $\Phi = \tan\left(\frac{N}{2}\psi\right)$, $\psi = \theta + \arctan\frac{\partial\phi/\partial y}{\partial\phi/\partial x}$, θ is the orientation, and N is the symmetry.

```

>>> alpha = 0.015
>>> c = 0.02
>>> N = 6.
>>> theta = pi / 8.
>>> psi = theta + arctan2(phase.getFaceGrad() [1],
...                           phase.getFaceGrad() [0])
>>> Phi = 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 / pi * 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.getCellCenters()
>>> 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()

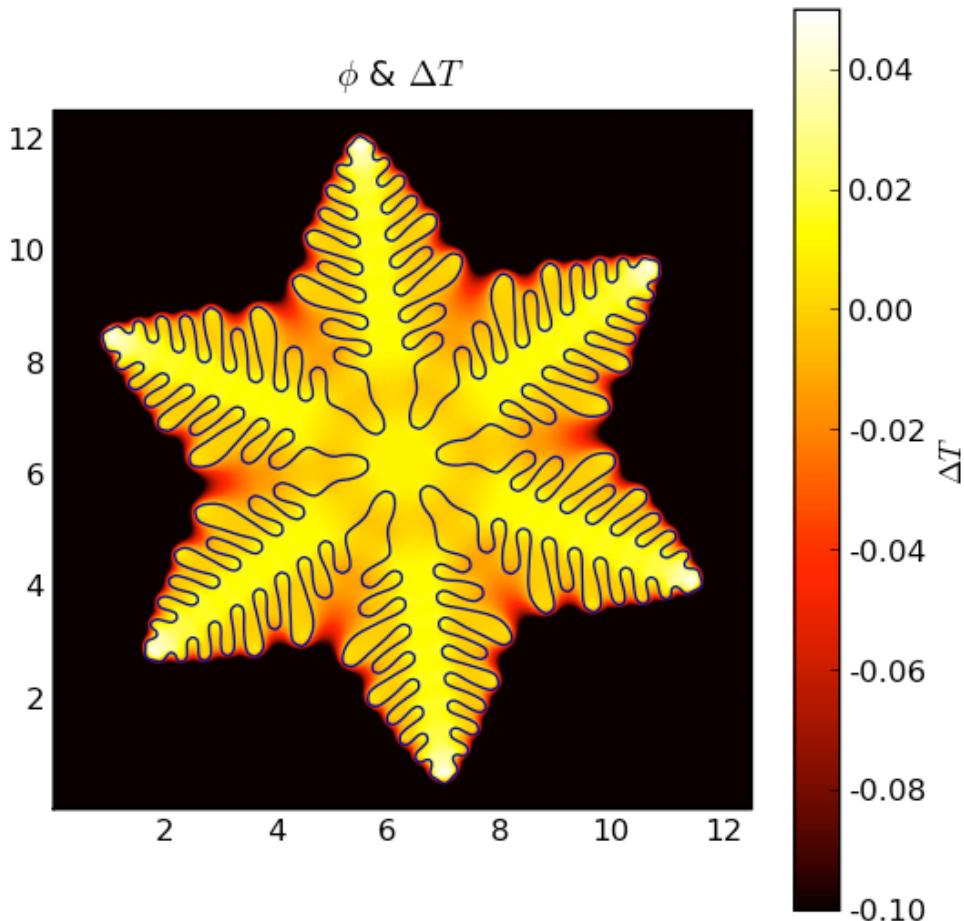
...
    mesh = self.phase.getMesh()
    shape = mesh.getShape()
    x, y = mesh.getCellCenters()
    z = self.phase.getValue()
    x, y, z = [a.reshape(shape, order="FORTRAN") for a in (x, y, z)]

...
    self.contour = pylab.contour(x, y, z, (0.5,))

...
    viewer = DendriteViewer(phase=phase, dT=dT,
                           title=r"%s & %s" % (phase.name, dT.name),
                           datamin=-0.1, datamax=0.05)
...
except ImportError:
    viewer = MultiViewer(viewers=(Viewer(vars=phase),
                                 Viewer(vars=dT,
                                         datamin=-0.5,
                                         datamax=0.5)))
```

and iterate the solution in time, plotting as we go,

```
>>> if __name__ == '__main__':
...     steps = 10000
... else:
...     steps = 10
>>> 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 [WarrenPolycrystal].

9.5 examples.phase.impingement.mesh40x1

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 [WarrenPolycrystal]

```
>>> from fipy import *
>>> nx = 40
>>> Lx = 2.5 * nx / 100.
```

```
>>> dx = Lx / nx
>>> mesh = Grid1D(dx=dx, nx=nx)
```

This problem simulates the wet boundary that forms between grains of different orientations. The phase equation is given by

$$\tau_\phi \frac{\partial \phi}{\partial t} = \alpha^2 \nabla^2 \phi + \phi(1 - \phi)m_1(\phi, T) - 2s\phi|\nabla\theta| - \epsilon^2\phi|\nabla\theta|^2$$

where

$$m_1(\phi, T) = \phi - \frac{1}{2} - T\phi(1 - \phi)$$

and the orientation equation is given by

$$P(\epsilon|\nabla\theta|)\tau_\theta\phi^2\frac{\partial\theta}{\partial t} = \nabla \cdot \left[\phi^2 \left(\frac{s}{|\nabla\theta|} + \epsilon^2 \right) \nabla\theta \right]$$

where

$$P(w) = 1 - \exp(-\beta w) + \frac{\mu}{\epsilon} \exp(-\beta w)$$

The initial conditions for this problem are set such that $\phi = 1$ for $0 \leq x \leq L_x$ and

$$\theta = \begin{cases} 1 & \text{for } 0 \leq x < L_x/2, \\ 0 & \text{for } L_x/2 \leq x \leq L_x. \end{cases}$$

Here the phase and orientation equations are solved with an explicit and implicit technique respectively.

The parameters for these equations are

```
>>> timeStepDuration = 0.02
>>> phaseTransientCoeff = 0.1
>>> thetaSmallValue = 1e-6
>>> beta = 1e5
>>> mu = 1e3
>>> thetaTransientCoeff = 0.01
>>> gamma = 1e3
>>> epsilon = 0.008
>>> s = 0.01
>>> alpha = 0.015
```

The system is held isothermal at

```
>>> temperature = 1.
```

and is initially solid everywhere

```
>>> phase = CellVariable(
...     name='phase field',
...     mesh=mesh,
...     value=1.
... )
```

Because `theta` is an S^1 -valued variable (i.e. it maps to the circle) and thus intrinsically has 2π -periodicity, we must use `ModularVariable` instead of a `CellVariable`. A `ModularVariable` confines `theta` to $-\pi < \theta \leq \pi$ by adding or subtracting 2π where necessary and by defining a new subtraction operator between two angles.

```
>>> theta = ModularVariable(
...     name='theta',
...     mesh=mesh,
...     value=1.,
...     hasOld=1
... )
```

The left and right halves of the domain are given different orientations.

```
>>> theta.setValue(0., where=mesh.getCellCenters() [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.getOld().getGrad().getMag()
>>> 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.getGrad().getMag()
>>> expo = (expo < 100.) * (expo - 100.) + 100.
>>> pFunc = 1. + exp(-expo) * (mu / epsilon - 1.)

>>> phaseFace = phase.getArithmeticFaceValue()
>>> phaseSq = phaseFace * phaseFace
>>> gradMag = theta.getFaceGrad().getMag()
>>> 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.getFaceGradNoMod()` evaluates the gradient without modular arithmetic.

```
>>> thetaGradDiff = theta.getFaceGrad() - theta.getFaceGradNoMod()
>>> sourceCoeff = (diffusionCoeff * thetaGradDiff).getDivergence()
```

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=-pi, datamax=pi)
...     phaseViewer.plot()
...     thetaProductViewer.plot()
```

we iterate the solution in time, plotting as we go if running interactively,

```
>>> steps = 10
>>> for i in range(steps):
...     theta.updateOld()
...     phase.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
>>> testData = loadtxt(os.path.splitext(__file__)[0] + '.gz')
>>> testData = CellVariable(mesh=mesh, value=testData)
>>> print theta.allclose(testData)
1
```

9.6 examples.phase.impingement.mesh20x20

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 *
```

```
>>> steps = numberOfSteps
>>> N = int(sqrt(numberOfElements))
>>> L = 2.5 * N / 100.
>>> dL = L / N
>>> mesh = Grid2D(dx=dL, dy=dL, nx=N, ny=N)
```

The initial conditions are given by $\phi = 1$ and

$$\theta = \begin{cases} \frac{2\pi}{3} & \text{for } x^2 - y^2 < L/2, \\ \frac{-2\pi}{3} & \text{for } (x - L)^2 - y^2 < L/2, \\ \frac{-2\pi}{3} + 0.3 & \text{for } x^2 - (y - L)^2 < L/2, \\ \frac{2\pi}{3} & \text{for } (x - L)^2 - (y - L)^2 < L/2. \end{cases}$$

This defines four solid regions with different orientations. Solidification occurs and then boundary wetting occurs where the orientation varies.

The parameters for this example are

```
>>> timeStepDuration = 0.02
>>> phaseTransientCoeff = 0.1
>>> thetaSmallValue = 1e-6
>>> beta = 1e5
>>> mu = 1e3
>>> thetaTransientCoeff = 0.01
>>> gamma = 1e3
>>> epsilon = 0.008
>>> s = 0.01
>>> alpha = 0.015
```

The system is held isothermal at

```
>>> temperature = 10.
```

and is initialized to liquid everywhere

```
>>> phase = CellVariable(name='phase field', mesh=mesh)
```

The orientation is initialized to a uniform value to denote the randomly oriented liquid phase

```
>>> theta = ModularVariable(
...     name='theta',
...     mesh=mesh,
...     value=-pi + 0.0001,
...     hasOld=1
... )
```

Four different solid circular domains are created at each corner of the domain with appropriate orientations

```
>>> x, y = mesh.getCellCenters()
>>> for a, b, thetaValue in ((0., 0., 2. * pi / 3.),
...                           (L, 0., -2. * pi / 3.),
...                           (0., L, -2. * pi / 3. + 0.3),
...                           (L, L, 2. * 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.getOld().getGrad().getMag()
...     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.getGrad().getMag()
...     expo = (expo < 100.) * (expo - 100.) + 100.
...     pFunc = 1. + exp(-expo) * (mu / epsilon - 1.)
...
...     phaseFace = phase.getArithmeticFaceValue()
...     phaseSq = phaseFace * phaseFace
...     gradMag = theta.getFaceGrad().getMag()
...     eps = 1. / gamma / 10.
...     gradMag += (gradMag < eps) * eps
...     IGamma = (gradMag > 1. / gamma) * (1 / gradMag - gamma) + gamma
...     diffusionCoeff = phaseSq * (s * IGamma + epsilon**2)
...
...     thetaGradDiff = theta.getFaceGrad() - theta.getFaceGradNoMod()
...     sourceCoeff = (diffusionCoeff * thetaGradDiff).getDivergence()
...
...     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 = -pi + phase * (theta + pi)
...     thetaProductViewer = Viewer(vars=thetaProd,
...                                 datamin=-pi, datamax=pi)
```

```
...     phaseViewer.plot()
...     thetaProductViewer.plot()
```

The solution will be tested against data that was created with `steps = 10` with a FORTRAN code written by Ryo Kobayashi for phase field modeling. The following code opens the file `mesh20x20.gz` extracts the data and compares it with the `theta` variable.

```
>>> import os
>>> testData = loadtxt(os.path.splitext(__file__)[0] + '.gz').flat
```

We step the solution in time, plotting as we go if running interactively,

```
>>> for i in range(steps):
...     theta.updateOld()
...     phase.updateOld()
...     thetaEq.solve(theta, dt=timeStepDuration)
...     phaseEq.solve(phase, dt=timeStepDuration)
...     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(-pi + 0.0001)
>>> x, y = mesh.getCellCenters()
>>> for a, b, thetaValue in ((0., 0., 2. * pi / 3.),
...                           (L, 0., -2. * pi / 3.),
...                           (0., L, -2. * pi / 3. + 0.3),
...                           (L, L, 2. * pi / 3.)):
...     segment = (x - a)**2 + (y - b)**2 < (L / 2.)**2
...     phase.setValue(1., where=segment)
...     theta.setValue(thetaValue, where=segment)
```

Step through half the time steps.

```
>>> for i in range(steps / 2):
...     theta.updateOld()
...     phase.updateOld()
...     thetaEq.solve(theta, dt=timeStepDuration)
...     phaseEq.solve(phase, dt=timeStepDuration)
```

We confirm that the solution has not yet converged to that given by Ryo Kobayashi's FORTRAN code:

```
>>> print theta.allclose(testData)
0
```

We save the variables to disk.

```
>>> (f, filename) = dump.write({'phase' : phase, 'theta' : theta}, extension = '.gz')
```

and then recall them to test the data pickling mechanism

```
>>> data = dump.read(filename, f)
>>> newPhase = data['phase']
>>> newTheta = data['theta']
>>> newThetaEq = buildThetaEquation(newPhase, newTheta)
>>> newPhaseEq = buildPhaseEquation(newPhase, newTheta)
```

and finish the iterations,

```
>>> for i in range(steps / 2):
...     newTheta.updateOld()
...     newPhase.updateOld()
...     newThetaEq.solve(newTheta, dt=timeStepDuration)
...     newPhaseEq.solve(newPhase, dt=timeStepDuration)
```

The solution is compared against Ryo Kobayashi's test data

```
>>> print newTheta.allclose(testData, rtol=1e-7)
1
```

Level Set Examples

<code>examples.levelSet.distanceFunction.</code>	Here we create a level set variable in one dimension.
<code>examples.levelSet.distanceFunction.</code>	Here we solve the level set equation in two dimensions for a circle.
<code>examples.levelSet.advection.mesh1D</code>	This example first solves the distance function equation in one dimension: ..
<code>examples.levelSet.advection.circle</code>	This example first imposes a circular distance function: ..

10.1 examples.levelSet.distanceFunction.mesh1D

Here we 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 *
>>> dx = 0.5
>>> nx = 10
```

Construct the mesh.

```
>>> from fipy.tools import serial
>>> mesh = Grid1D(dx=dx, nx=nx, parallelModule=serial)
```

Construct a *distanceVariable* object.

```
>>> var = DistanceVariable(name='level set variable',
...                         mesh=mesh,
...                         value=-1,
...                         hasOld=1)
>>> x = mesh.getCellCenters()[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 allclose(var, x - dx * nx / 2)
1
```

10.2 examples.levelSet.distanceFunction.circle

Here we 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 *
>>> dx = 1.
>>> dy = 1.
>>> nx = 11
>>> ny = 11
>>> Lx = nx * dx
>>> Ly = ny * dy
```

Construct the mesh.

```
>>> from fipy.tools import serial
>>> mesh = Grid2D(dx=dx, dy=dy, nx=nx, ny=ny, parallelModule=serial)
```

Construct a *distanceVariable* object.

```
>>> var = DistanceVariable(name='level set variable',
...                         mesh=mesh,
...                         value=-1,
...                         hasOld=1)

>>> x, y = mesh.getCellCenters()
>>> var.setValue(1, where=(x - Lx / 2.)**2 + (y - Ly / 2.)**2 < (Lx / 4.)**2)

>>> var.calcDistanceFunction()

>>> 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 / 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 = 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, -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, 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), 'd')))

>>> var[numerix.array(trialValues == MASK)] = MASK
>>> print numerix.allclose(var, trialValues)
True
```

10.3 examples.levelSet.advection.mesh1D

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 *AdvectionTerm* 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 *
```

```
>>> velocity = 1.
>>> dx = 1.
>>> nx = 10
>>> timeStepDuration = 1.
>>> steps = 2
>>> L = nx * dx
>>> interfacePosition = L / 5.
```

Construct the mesh.

```
>>> from fipy.tools import serial
>>> mesh = Grid1D(dx=dx, nx=nx, parallelModule=serial)
```

Construct a *distanceVariable* object.

```
>>> var = DistanceVariable(name='level set variable',
...                           mesh=mesh,
...                           value=-1.,
...                           hasOld=1)
>>> var.setValue(1., where=mesh.getCellCenters() [0] > interfacePosition)
>>> var.calcDistanceFunction()
```

The *advectionEquation* is constructed.

```
>>> advEqn = buildAdvectionEquation(advectionCoeff=velocity)
```

The problem can then be solved by executing a series of time steps.

```
>>> if __name__ == '__main__':
...     viewer = Viewer(vars=var, datamin=-10., datamax=10.)
...     viewer.plot()
...     for step in range(steps):
...         var.updateOld()
...         advEqn.solve(var, dt=timeStepDuration)
...         viewer.plot()
```

The result can be tested with the following code:

```
>>> for step in range(steps):
...     var.updateOld()
...     advEqn.solve(var, dt=timeStepDuration)
>>> x = mesh.getCellCenters() [0]
>>> distanceTravelled = timeStepDuration * steps * velocity
>>> answer = x - interfacePosition - timeStepDuration * steps * velocity
>>> answer = where(x < distanceTravelled,
...                  x[0] - interfacePosition, answer)
>>> print var.allclose(answer)
1
```

10.4 examples.levelSet.advection.circle

This example first imposes a circular distance function:

$$\phi(x, y) = \left[\left(x - \frac{L}{2} \right)^2 + \left(y - \frac{L}{2} \right)^2 \right]^{1/2} - \frac{L}{4}$$

The variable is advected with,

$$\frac{\partial \phi}{\partial t} + \vec{u} \cdot \nabla \phi = 0$$

The scheme used in the `_AdvectionTerm` 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 *

>>> 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)
```

Initialise the *distanceVariable* to be a circular distance function.

```
>>> x, y = mesh.getCellCenters()
>>> initialArray = sqrt((x - L / 2.)**2 + (y - L / 2.)**2) - radius
>>> var.setValue(initialArray)
```

The `AdvectionEquation` is constructed.

```
>>> advEqn = buildAdvectionEquation(
...     advectionCoeff=velocity)
```

The problem can then be solved by executing a series of time steps.

```
>>> if __name__ == '__main__':
...     viewer = Viewer(vars=var, datamin=-radius, datamax=radius)
...     viewer.plot()
...     for step in range(steps):
...         var.updateOld()
```

```
...     advEqn.solve(var, dt=timeStepDuration)
...     viewer.plot()
```

The result can be tested with the following commands.

```
>>> for step in range(steps):
...     var.updateOld()
...     advEqn.solve(var, dt=timeStepDuration)
>>> x = array(mesh.getCellCenters() [0])
>>> distanceTravelled = timeStepDuration * steps * velocity
>>> answer = initialArray - distanceTravelled
>>> answer = where(answer < 0., -1001., answer)
>>> solution = where(answer < 0., -1001., array(var))
>>> allclose(answer, solution, atol=4.7e-3)
1
```

If the AdvectionEquation is built with the `_HigherOrderAdvectionTerm` the result is more accurate,

```
>>> var.setValue(initialArray)
>>> advEqn = buildHigherOrderAdvectionEquation(
...     advectionCoeff = velocity)
>>> for step in range(steps):
...     var.updateOld()
...     advEqn.solve(var, dt=timeStepDuration)
>>> solution = where(answer < 0., -1001., array(var))
>>> allclose(answer, solution, atol=1.02e-3)
1
```

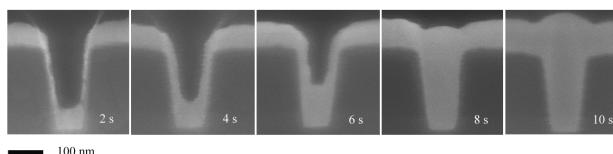
10.5 Superconformal Electrodeposition Examples

10.5.1 The Damascene Process

State of the art manufacturing of semiconductor devices involves the electrodeposition of copper for on-chip wiring of integrated circuits. In the Damascene process interconnects are fabricated by first patterning trenches in a dielectric medium and then filling by metal electrodeposition over the entire wafer surface. This metalization process, pioneered by IBM, depends on the use of electrolyte additives that effect the local metal deposition rate.

10.5.2 Superfill

The additives in the electrolyte affect the local deposition rate in such a way that bottom-up filling occurs in trenches or vias. This process, known as superconformal electrodeposition or superfill, is demonstrated in the following figure. The figure shows sequential images of bottom-up superfilling of submicrometer trenches by copper deposition from an electrolyte containing PEG-SPS-Cl. Preferential metal deposition at the bottom of the trenches followed by bump formation above the filled trenches is evident.



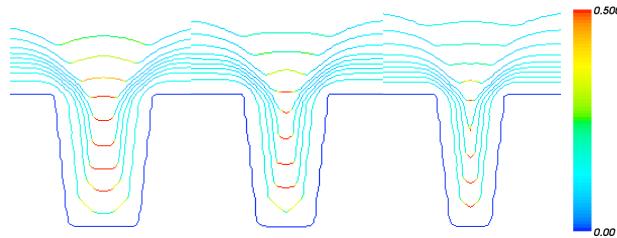
10.5.3 The CEAC Mechanism

This process has been demonstrated to depend critically on the inclusion of additives in the electrolyte. Recent publications propose Curvature Enhanced Accelerator Coverage (CEAC) as the mechanism behind the superfilling process [NIST:damascene:2001]. In this mechanism, molecules that accelerate local metal deposition displace molecules that inhibit local metal deposition on the metal/electrolyte interface. For electrolytes that yield superconformal filling of fine features, this buildup happens relatively slowly because the concentration of accelerator species is much more dilute compared to the inhibitor species in the electrolyte. The mechanism that leads to the increased rate of metal deposition along the bottom of the filling trench is the concurrent local increase of the accelerator coverage due to decreasing local surface area, which scales with the local curvature (hence the name of the mechanism). A good overview of this mechanism can be found in [moffatInterface:2004].

10.5.4 Using FiPy to model Superfill

Example ?? provides a simple way to use *FiPy* to model the superfill process. The example includes a detailed description of the governing equations and feature geometry. It requires the user to import and execute a function at the python prompt. The model parameters can be passed as arguments to this function. In future all superfill examples will be provided with this type of interface. Example ?? has the same functionality as ?? but demonstrates how to write a new script in the case where the existing suite of scripts do not meet the required needs.

In general it is a good idea to obtain the *Mayavi* plotting package for which a specialized superfill viewer class has been created, see *Installation*. The other standard viewers mentioned in *Installation* are still adequate although they do not give such clear images that are tailored for the superfill problem. The images below demonstrate the *Mayavi* viewing capability. Each contour represents sequential positions of the interface and the color represents the concentration of accelerator as a surfactant. The areas of high surfactant concentration have an increased deposition rate.



```
examples.levelSet.electroChem This input file
examples.levelSet.electroChem This input file
examples.levelSet.electroChem This input file is a demonstration of the use of FiPy for
examples.levelSet.electroChem This input file demonstrates how to create a new superfill script if the
existing suite of scripts do not meet the required needs.
```

10.6 examples.levelSet.electroChem.simpleTrenchSystem

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 [NIST:damascene:2003].

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. In order to alter the number of timesteps, the python function that encapsulates the system of equations must first be imported (at the python command line),

```
>>> from examples.levelSet.electroChem.simpleTrenchSystem import runSimpleTrenchSystem
```

and then the function can be run with a different number of time steps with the `numberOfSteps` argument as follows,

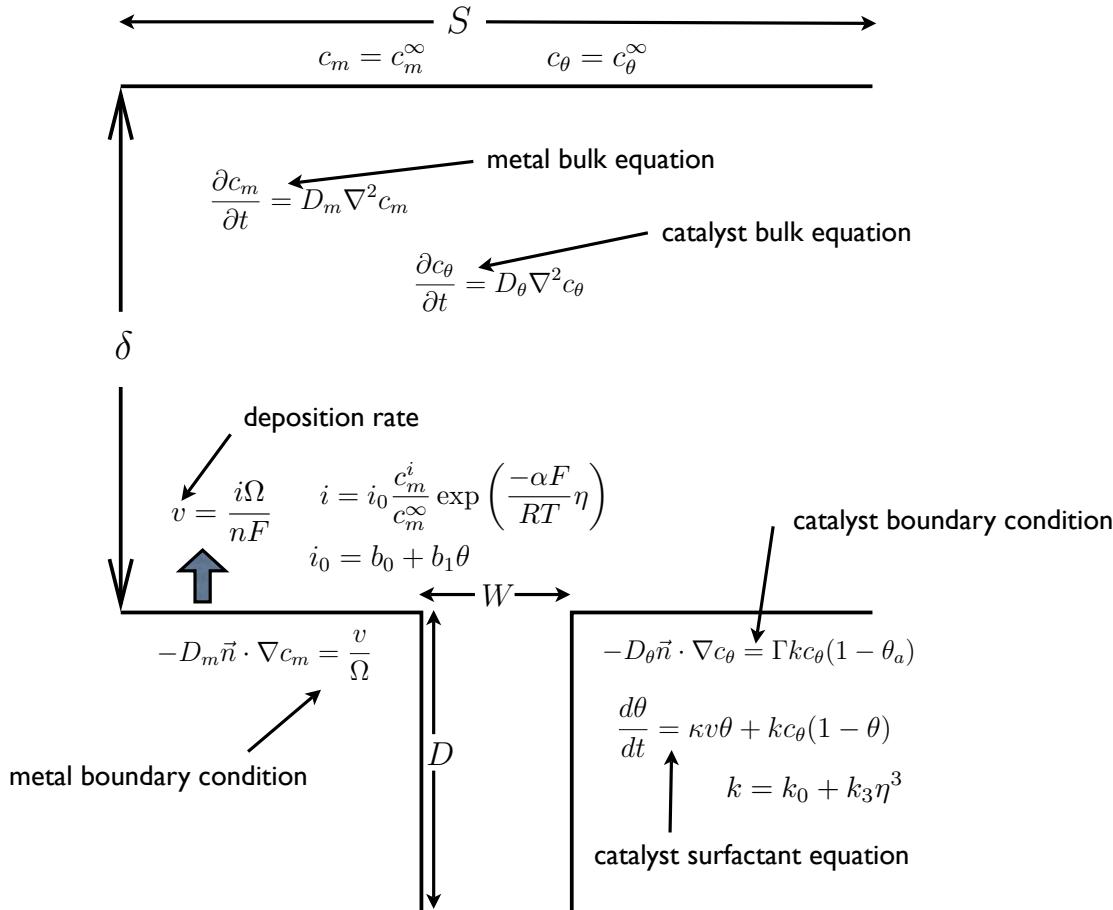
```
>>> runSimpleTrenchSystem(numberOfSteps=5, displayViewers=False)
1
```

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(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 `runSimpleTrenchSystem()` function. As stated above, all the parameters in the equations can be changed with function arguments.

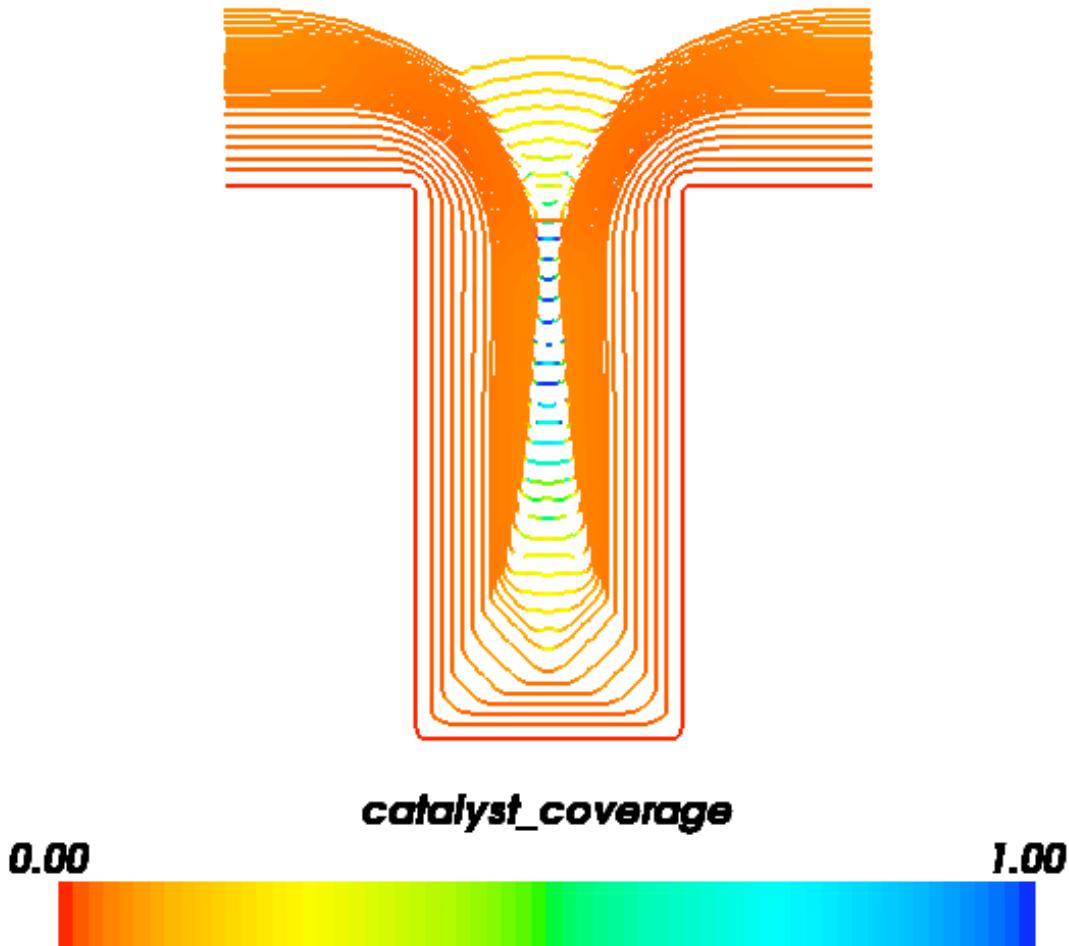


The following table shows the symbols used in the governing equations and their corresponding arguments to the `runSimpleTrenchSystem()` function. The boundary layer depth is intentionally small in this example in order

not to complicate the mesh. Further examples will simulate more realistic boundary layer depths but will also have more complex meshes requiring the **gmsh** software.

Symbol	Description	Keyword Argument	Value	Unit
Deposition Rate Parameters				
v	deposition rate			m s^{-1}
i	current density			A m^{-2}
Ω	molar volume	molarVolume	7.1×10^{-6}	$\text{m}^3 \text{ mol}^{-1}$
n	ion charge	charge	2	
F	Faraday's constant	faradaysConstant	9.6×10^{-4}	C mol^{-1}
i_0	exchange current density			A m^{-2}
α	transfer coefficient	transferCoefficient	0.5	
η	overpotential	overpotential	-0.3	V
R	gas constant	gasConstant	8.314	$\text{J K}^{-1} \text{ mol}^{-1}$
T	temperature	temperature	298.0	K
b_0	current density for θ^0	currentDensity0	0.26	A m^{-2}
b_1	current density for θ	currentDensity1	45.0	A m^{-2}
Metal Ion Parameters				
c_m	metal ion concentration	metalConcentration	250.0	mol m^{-3}
c_m^∞	far field metal ion concentration	metalConcentration	250.0	mol m^{-3}
D_m	metal ion diffusion coefficient	metalDiffusion	5.6×10^{-10}	$\text{m}^2 \text{ s}^{-1}$
Catalyst Parameters				
θ	catalyst surfactant concentration	catalystCoverage	0.0	
c_θ	bulk catalyst concentration	catalystConcentration	5.0×10^{-3}	mol m^{-3}
c_θ^∞	far field catalyst concentration	catalystConcentration	5.0×10^{-3}	mol m^{-3}
D_θ	catalyst diffusion coefficient	catalystDiffusion	1.0×10^{-9}	$\text{m}^2 \text{ s}^{-1}$
Γ	catalyst site density	siteDensity	9.8×10^{-6}	mol m^{-2}
k	rate constant			$\text{m}^3 \text{ mol}^{-1} \text{ s}^{-1}$
k_0	rate constant for η^0	rateConstant0	1.76	$\text{m}^3 \text{ mol}^{-1} \text{ s}^{-1}$
k_3	rate constant for η^3	rateConstant3	-245.0×10^{-6}	$\text{m}^3 \text{ mol}^{-1} \text{ s}^{-1} \text{ V}^{-3}$
Geometry Parameters				
D	trench depth	trenchDepth	0.5×10^{-6}	m
D/W	trench aspect ratio	aspectRatio	2.0	
S	trench spacing	trenchSpacing	0.6×10^{-6}	m
δ	boundary layer depth	boundaryLayerDepth	0.3×10^{-6}	m
Simulation Control Parameters				
computational cell size		cellSize	0.1×10^{-7}	m
number of time steps		numberOfSteps	5	
whether to display the viewers		displayViewers	True	

If the MayaVi plotting software is installed (see [Installation](#)) then a plot should appear that is updated every 20 time steps and will eventually resemble the image below.



Functions

```
runSimpleTrenchSystem([faradaysConstant, ...])
```

10.7 examples.levelSet.electroChem.gold

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 [NIST:damascene:2005].

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. In order to alter the number of timesteps, the Python function that encapsulates the system of equations must first be imported (at the Python command line),

```
>>> from examples.levelSet.electroChem.gold import runGold
```

and then the function can be run with a different number of time steps with 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,

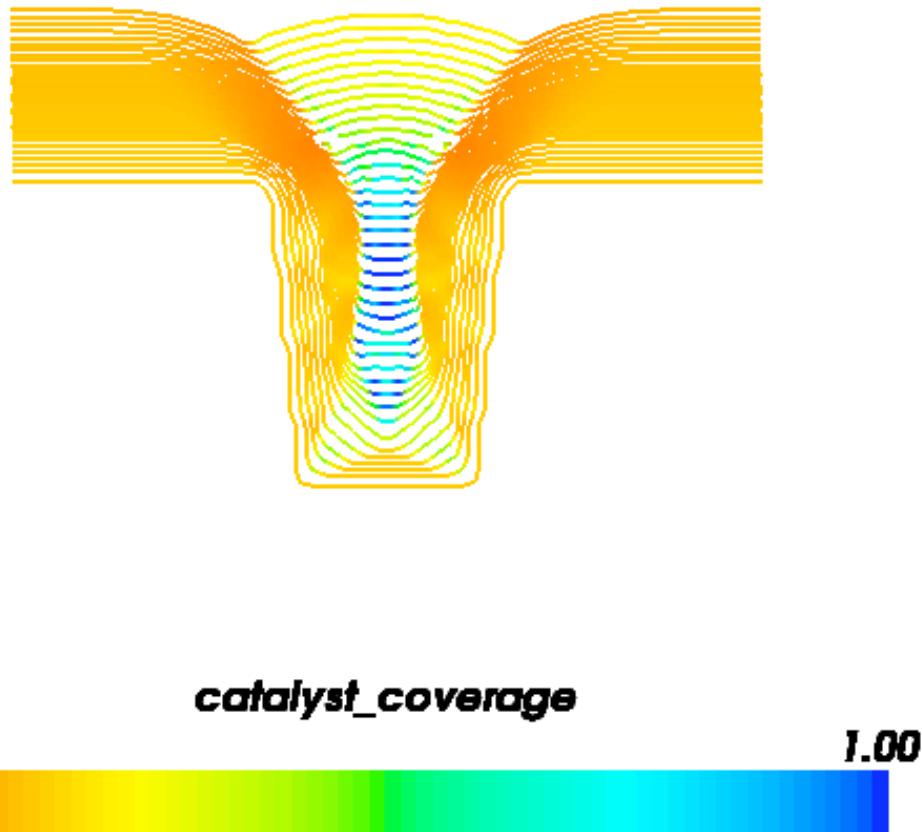
$$i = \frac{c_m}{c_m^\infty} (b_0 + b_1 \theta).$$

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.



Functions

```
runGold([faradaysConstant, ...])
```

10.8 examples.levelSet.electroChem.leveler

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 [NIST:leveler:2005].

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. In order to alter the number of timesteps, the python function that encapsulates the system of equations must first be imported (at the python command line),

```
>>> from examples.levelSet.electroChem.leveler import runLeveler
```

and then the function can be executed with a different number of time steps by changing 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 absorb 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 i_j represents S for suppressor, A for accelerator, L for leveler and V for vacant. This model assumes a linear interpolation between the three cases of complete coverage for each additive or vacant substrate. The governing equations for the surfactants are given by,

$$\begin{aligned}\dot{\theta}_L &= \kappa v \theta_L + k_l^+ c_L (1 - \theta_L) - k_l^- v \theta_L, \\ \dot{\theta}_a &= \kappa v \theta_A + k_A^+ c_A (1 - \theta_A - \theta_L) - k_L c_L \theta_A - k_A^- \theta_A^{q-1}, \\ \theta_S &= 1 - \theta_A - \theta_L \\ \theta_V &= 0.\end{aligned}$$

It has been found experimentally that $i_L = i_S$.

If the surface reaches full coverage, the equations do not naturally prevent the coverage rising above full coverage due to the curvature terms. Thus, when $\theta_L + \theta_A = 1$ then the equation for accelerator becomes $\dot{\theta}_A = -\dot{\theta}_L$ and when $\theta_L = 1$, the equation for leveler becomes $\dot{\theta}_L = -k_L^\alpha \theta_L^{\alpha-1} \nabla \theta_L$.

The parameters k_A^+ , k_A^- and q are both functions of η given by,

$$k_A^+ = k_{A0}^+ \exp \frac{-\alpha_k F \eta}{RT},$$

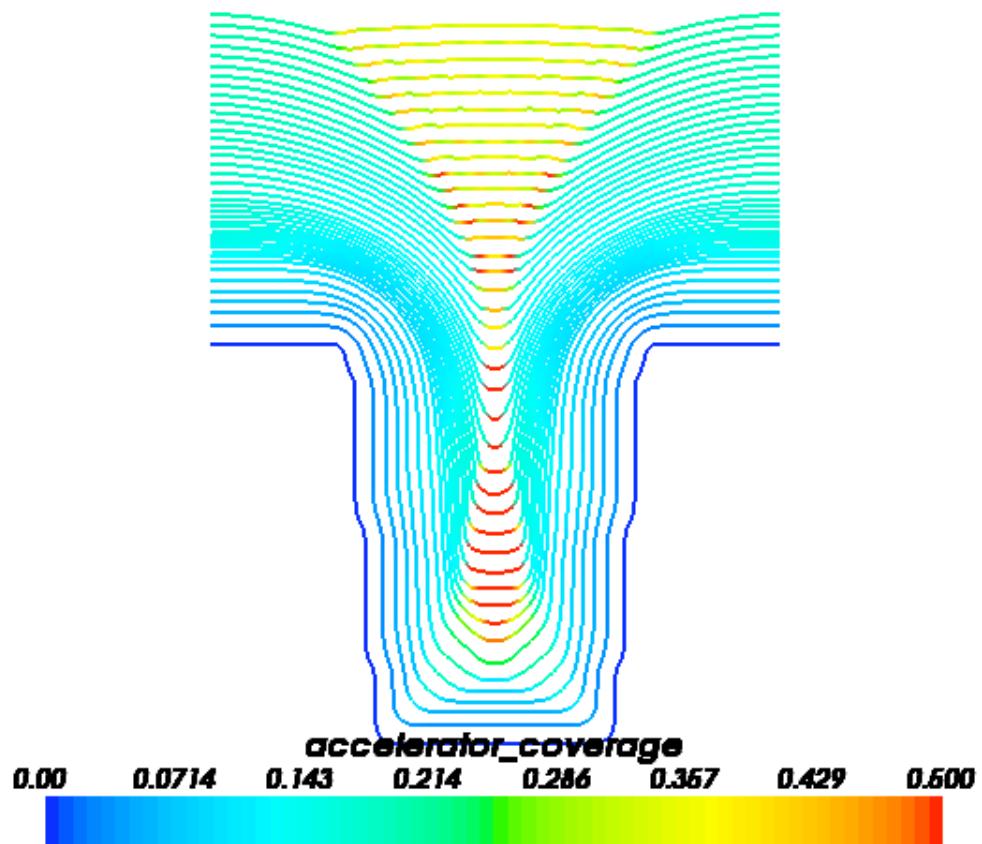
$$k_A^- = B_d + \frac{A}{\exp(B_a(\eta + V_d))} + \exp(B_b(\eta + V_d))$$

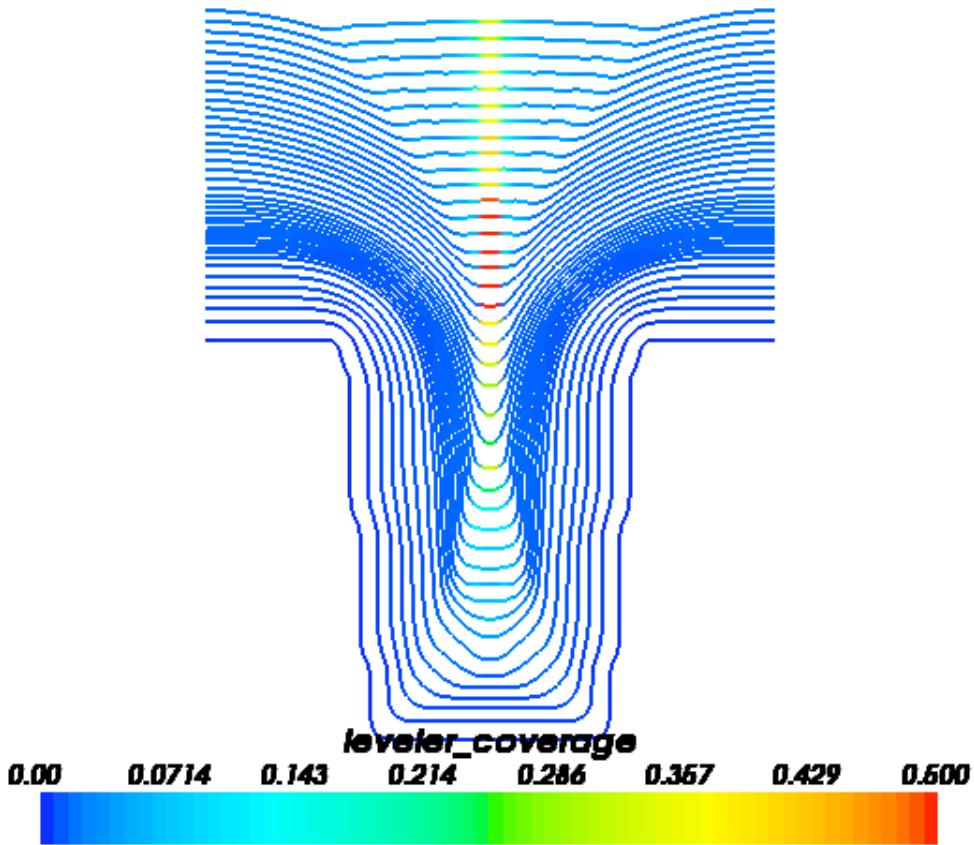
$$q = m * \eta + b.$$

The following table shows the symbols used in the governing equations and their corresponding arguments for the `runLeveler()` function.

Symbol	Description	Keyword Argument	Value	Unit
Deposition Rate Parameters				
v	deposition rate			m s^{-1}
i_A	accelerator current density	<code>i0Accelerator</code>		A m^{-2}
i_L	leveler current density	<code>i0Leveler</code>		A m^{-2}
Ω	molar volume	<code>molarVolume</code>	7.1×10^{-6}	$\text{m}^3 \text{ mol}^{-1}$
n	ion charge	<code>charge</code>	2	
F	Faraday's constant	<code>faradaysConstant</code>	9.6×10^{-4}	C mol^{-1}
i_0	exchange current density			A m^{-2}
α_A	accelerator transfer coefficient	<code>alphaAccelerator</code>	0.4	
α_S	leveler transfer coefficient	<code>alphaLeveler</code>	0.5	
η	overpotential	<code>overpotential</code>	-0.3	V
R	gas constant	<code>gasConstant</code>	8.314	J K mol^{-1}
T	temperature	<code>temperature</code>	298.0	K
Ion Parameters				
c_I	ion concentration	<code>ionConcentration</code>	250.0	mol m^{-3}
c_I^∞	far field ion concentration	<code>ionConcentration</code>	250.0	mol m^{-3}
D_I	ion diffusion coefficient	<code>ionDiffusion</code>	5.6×10^{-10}	$\text{m}^2 \text{ s}^{-1}$
Accelerator Parameters				
θ_A	accelerator coverage	<code>acceleratorCoverage</code>	0.0	
c_A	accelerator concentration	<code>acceleratorConcentration</code>	5.0×10^{-3}	mol m^{-3}
c_A^∞	far field accelerator concentration	<code>acceleratorConcentration</code>	5.0×10^{-3}	mol m^{-3}
D_A	catalyst diffusion coefficient	<code>catalystDiffusion</code>	1.0×10^{-9}	$\text{m}^2 \text{ s}^{-1}$
Γ_A	accelerator site density	<code>siteDensity</code>	9.8×10^{-6}	mol m^{-2}
k_A^+	accelerator adsorption			$\text{m}^3 \text{ mol}^{-1} \text{ s}^{-1}$
k_{A0}^+	accelerator adsorption coeff	<code>kAccelerator0</code>	2.6×10^{-4}	$\text{m}^3 \text{ mol}^{-1} \text{ s}^{-1}$
α_k	accelerator adsorption coeff	<code>alphaAdsorption</code>	0.62	
k_A^-	accelerator consumption coeff			
B_a	experimental parameter	<code>Bd</code>	-40.0	
B_b	experimental parameter	<code>Bd</code>	60.0	
V_d	experimental parameter	<code>Bd</code>	9.8×10^{-2}	
B_d	experimental parameter	<code>Bd</code>	8.0×10^{-4}	
Geometry Parameters				
D	trench depth	<code>trenchDepth</code>	0.5×10^{-6}	m
D/W	trench aspect ratio	<code>aspectRatio</code>	2.0	
S	trench spacing	<code>trenchSpacing</code>	0.6×10^{-6}	m
δ	boundary layer depth	<code>boundaryLayerDepth</code>	0.3×10^{-6}	m
Simulation Control Parameters				
computational cell size	<code>cellSize</code>		0.1×10^{-7}	m
number of time steps	<code>numberOfSteps</code>		5	
whether to display the viewers	<code>displayViewers</code>		True	

The following images show accelerator and leveler contour plots that can be obtained by running this example.





Functions

`runLeveler([kLeveler, ...])`

10.9 examples.levelSet.electroChem.howToWriteAScript

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=
```

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
```

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
>>> number_of_Cells_in_Narrow_Band = 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=5)

>>> if numberofElements != -1:
...     pos = trenchSpacing * cellsBelowTrench / 4 / numberofElements
...     sqr = trenchSpacing * (trenchDepth + boundaryLayerDepth) \
...         / (2 * numberofElements)
...     cellSize = pos + sqrt(pos**2 + sqr)
... else:
...     cellSize = 0.1e-7

>>> yCells = cellsBelowTrench \
...     + int((trenchDepth + boundaryLayerDepth) / cellSize)
>>> xCells = int(trenchSpacing / 2 / cellSize)

>>> from fipy import *
>>> from fipy import serial
>>> mesh = Grid2D(dx=cellSize,
...                 dy=cellSize,
...                 nx=xCells,
...                 ny=yCells,
...                 parallelModule=serial)

```

A `distanceVariable` object, ϕ , is required to store the position of the interface.

The `distanceVariable` calculates its value so that it is a distance function (*i.e.* holds the distance at any point in the mesh from the electrolyte/metal interface at $\phi = 0$) and $|\nabla\phi| = 1$.

First, create the ϕ variable, which is initially set to -1 everywhere. Create an initial variable,

```

>>> narrowBandWidth = numberofCellsInNarrowBand * cellSize
>>> distanceVar = DistanceVariable(
...     name='distance variable',
...     mesh= mesh,
...     value=-1.,
...     narrowBandWidth=narrowBandWidth,
...     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.getCellCenters()
>>> distanceVar.setValue(1., where=(y > trenchHeight)
...                         | ((y > bottomHeight)
...                           & (x < xCells * cellSize - sideWidth)))
...                         )

>>> distanceVar.calcDistanceFunction(narrowBandWidth=1e10)

```

The `distanceVariable` has now been created to mark the interface. Some other variables need to be created that govern the concentrations of various species.

Create the catalyst surfactant coverage, θ , variable. This variable influences the deposition rate.

```
>>> catalystVar = SurfactantVariable(
...     name="catalyst variable",
...     value=catalystCoverage,
...     distanceVar=distanceVar)
```

Create the bulk catalyst concentration, c_θ , in the electrolyte,

```
>>> bulkCatalystVar = CellVariable(
...     name='bulk catalyst variable',
...     mesh=mesh,
...     value=catalystConcentration)
```

Create the bulk metal ion concentration, c_m , in the electrolyte.

```
>>> metalVar = CellVariable(
...     name='metal variable',
...     mesh=mesh,
...     value=bulkMetalConcentration)
```

The following commands build the `depositionRateVariable`, v . The `depositionRateVariable` is given by the following equation.

$$v = \frac{i\Omega}{nF}$$

where Ω is the metal molar volume, n is the metal ion charge and F is Faraday's constant. The current density is given by

$$i = i_0 \frac{c_m^i}{c_m^\infty} \exp\left(\frac{-\alpha F}{RT}\eta\right)$$

where c_m^i is the metal ion concentration in the bulk at the interface, c_m^∞ is the far-field bulk concentration of metal ions, α is the transfer coefficient, R is the gas constant, T is the temperature and η is the overpotential. The exchange current density is an empirical function of catalyst coverage,

$$i_0(\theta) = b_0 + b_1\theta$$

The commands needed to build this equation are,

```
>>> expoConstant = -transferCoefficient * faradaysConstant \
...             / (gasConstant * temperature)
>>> tmp = currentDensity1 \
...         * catalystVar.getInterfaceVar()
>>> exchangeCurrentDensity = currentDensity0 + tmp
>>> expo = exp(expoConstant * overpotential)
>>> currentDensity = expo * exchangeCurrentDensity * metalVar \
...             / bulkMetalConcentration
>>> depositionRateVariable = currentDensity * molarVolume \
...             / (charge * faradaysConstant)
```

Build the extension velocity variable v_{ext} . The extension velocity uses the `extensionEquation` to spread the velocity at the interface to the rest of the domain.

```
>>> extensionVelocityVariable = CellVariable(
...     name='extension velocity',
...     mesh=mesh,
...     value=depositionRateVariable)
```

Using the variables created above the governing equations will be built. The governing equation for surfactant conservation is given by,

$$\dot{\theta} = Jv\theta + kc_\theta^i(1 - \theta)$$

where θ is the coverage of catalyst at the interface, J is the curvature of the interface, v is the normal velocity of the interface, c_θ^i is the concentration of catalyst in the bulk at the interface. The value k is given by an empirical function of overpotential,

$$k = k_0 + k_3\eta^3$$

The above equation is represented by the [AdsorbingSurfactantEquation](#) in *FiPy*:

```
>>> surfactantEquation = AdsorbingSurfactantEquation(
...     surfactantVar=catalystVar,
...     distanceVar=distanceVar,
...     bulkVar=bulkCatalystVar,
...     rateConstant=rateConstant0 \
...                 + rateConstant3 * overpotential**3)
```

The variable ϕ is advected by the [advectionEquation](#) given by,

$$\frac{\partial \phi}{\partial t} + v_{\text{ext}}|\nabla \phi| = 0$$

and is set up with the following commands:

```
>>> advectionEquation = buildHigherOrderAdvectionEquation(
...     advectionCoeff=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 [MetalIonDiffusionEquation](#) is set up with the following commands.

```
>>> metalEquation = buildMetalIonDiffusionEquation(
...     ionVar=metalVar,
...     distanceVar=distanceVar,
...     depositionRate=depositionRateVariable,
...     diffusionCoeff=metalDiffusionCoefficient,
...     metalIonMolarVolume=molarVolume,
... )
```

```
>>> metalEquationBCs = FixedValue(faces=mesh.getFacesTop(), value=bulkMetalConcentration)
```

The SurfactantBulkDiffusionEquation 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 SurfactantBulkDiffusionEquation is set up with the following commands.

```
>>> bulkCatalystEquation = buildSurfactantBulkDiffusionEquation(
...     bulkVar=bulkCatalystVar,
...     distanceVar=distanceVar,
...     surfactantVar=catalystVar,
...     diffusionCoeff=catalystDiffusion,
...     rateConstant=rateConstant0 * siteDensity
... )
```

```
>>> catalystBCs = FixedValue(faces=mesh.getFacesTop(), value=catalystConcentration)
```

If running interactively, create viewers.

```
>>> if __name__ == '__main__':
...     try:
...         viewer = MayaviSurfactantViewer(distanceVar,
...                                         catalystVar.getInterfaceVar(),
...                                         zoomFactor=1e6,
...                                         datamax=1.0,
...                                         datamin=0.0,
...                                         smooth=1)
...     except:
...         viewer = MultiViewer(viewers=(
...             Viewer(distanceVar, datamin=-1e-9, datamax=1e-9),
...             Viewer(catalystVar.getInterfaceVar())))
... else:
...     viewer = None
```

The levelSetUpdateFrequency defines how often to call the distanceEquation to reinitialize the distanceVariable to a distance function.

```
>>> levelSetUpdateFrequency = int(0.8 * narrowBandWidth \
... / (cellSize * cflNumber * 2))
```

The following loop runs for numberOfSteps time steps. The time step is calculated with the CFL number and the maximum extension velocity, v to v_{ext} throughout the whole domain using $\nabla \phi \cdot \nabla v_{ext} = 0$.

```

>>> for step in range(numberOfSteps):
...
...     if viewer is not None:
...         viewer.plot()
...
...     if step % levelSetUpdateFrequency == 0:
...         distanceVar.calcDistanceFunction()
...
...     extensionVelocityVariable.setValue(depositionRateVariable())
...
...     distanceVar.updateOld()
...     catalystVar.updateOld()
...     metalVar.updateOld()
...     bulkCatalystVar.updateOld()
...     distanceVar.extendVariable(extensionVelocityVariable)
...     dt = cflNumber * cellSize / extensionVelocityVariable.max()
...     advectionEquation.solve(distanceVar, dt=dt)
...     surfactantEquation.solve(catalystVar, dt=dt)
...     metalEquation.solve(var=metalVar, dt=dt,
...                         boundaryConditions=metalEquationBCs)
...     bulkCatalystEquation.solve(var=bulkCatalystVar, dt=dt,
...                               boundaryConditions=catalystBCs)
...

```

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")
...
>>> print catalystVar.allclose(loadtxt(filepath), rtol=1e-4)
1
...
>>> if __name__ == '__main__':
...     raw_input('finished')

```


Cahn Hilliard Examples

<code>examples.cahnHilliard.mesh2D</code>	The spinodal decomposition phenomenon is a spontaneous separation of
<code>examples.cahnHilliard.sphere</code>	Solves the Cahn-Hilliard problem on the surface of a sphere, such as

11.1 examples.cahnHilliard.mesh2D

The spinodal decomposition phenomenon is a spontaneous separation of an initially homogenous mixture into two distinct regions of different properties (spin-up/spin-down, component A/component B). It is a “barrierless” phase separation process, such that under the right thermodynamic conditions, any fluctuation, no matter how small, will tend to grow. This is in contrast to nucleation, where a fluctuation must exceed some critical magnitude before it will survive and grow. Spinodal decomposition can be described by the “Cahn-Hilliard” equation (also known as “conserved Ginsberg-Landau” or “model B” of Hohenberg & Halperin)

$$\frac{\partial \phi}{\partial t} = \nabla \cdot D \nabla \left(\frac{\partial f}{\partial \phi} - \epsilon^2 \nabla^2 \phi \right).$$

where ϕ is a conserved order parameter, possibly representing alloy composition or spin. The double-well free energy function $f = (a^2/2)\phi^2(1-\phi)^2$ penalizes states with intermediate values of ϕ between 0 and 1. The gradient energy term $\epsilon^2 \nabla^2 \phi$, on the other hand, penalizes sharp changes of ϕ . These two competing effects result in the segregation of ϕ into domains of 0 and 1, separated by abrupt, but smooth, transitions. The parameters a and ϵ determine the relative weighting of the two effects and D is a rate constant.

We can simulate this process in *FiPy* with a simple script:

```
>>> from fipy import *
```

(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 = 10
>>> mesh = Grid2D(nx=nx, ny=ny, dx=0.25, dy=0.25)
>>> phi = CellVariable(name=r"\phi", mesh=mesh)
```

We start the problem with random fluctuations about $\phi = 1/2$

```
>>> phi.setValue(GaussianNoiseVariable(mesh=mesh,
...                                     mean=0.5,
...                                     variance=0.01))
```

FiPy doesn't plot or output anything unless you tell it to:

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

For *FiPy*, we need to perform the partial derivative $\partial f / \partial \phi$ manually and then put the equation in the canonical form by decomposing the spatial derivatives so that each `Term` is of a single, even order:

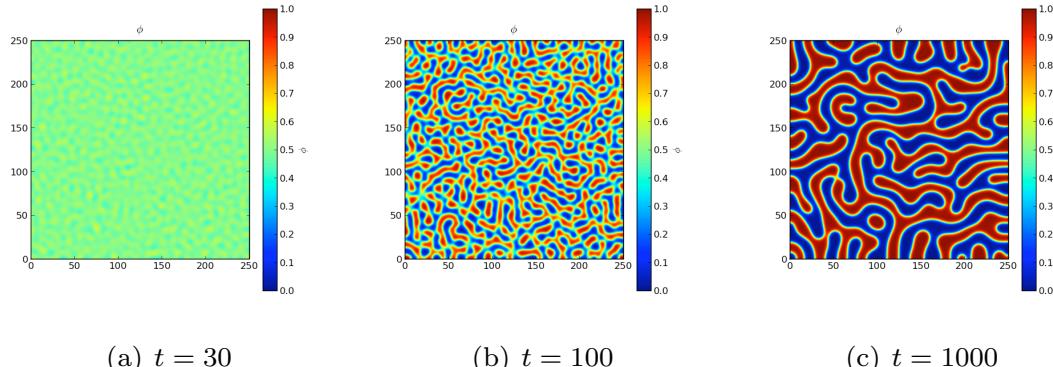
$$\frac{\partial \phi}{\partial t} = \nabla \cdot D a^2 [1 - 6\phi(1 - \phi)] \nabla \phi - \nabla \cdot D \nabla \epsilon^2 \nabla^2 \phi.$$

FiPy would automatically interpolate $D * a^{**2} * (1 - 6 * \phi * (1 - \phi))$ onto the `Faces`, where the diffusive flux is calculated, but we obtain somewhat more accurate results by performing a linear interpolation from `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.getArithmeticFaceValue()
>>> 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, exp(dexp))
...     elapsed += dt
...     dexp += 0.01
...     eq.solve(phi, dt=dt)
...     if __name__ == "__main__":
...         viewer.plot()
```



11.2 examples.cahnHilliard.sphere

Solves the Cahn-Hilliard problem on the surface of a sphere, such as may occur on vesicles (http://www.youtube.com/watch?v=kDsFP67_ZSE).

```
>>> from fipy import *
```

The only difference from `examples.cahnHilliard.mesh2D` is the declaration of `mesh`.

```
>>> mesh = GmshImporter2DIn3DSpace('''
...     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]};
... ''').extrude(extrudeFunc=lambda r: 1.1 * r)
>>> phi = CellVariable(name=r"\phi", mesh=mesh)
```

We start the problem with random fluctuations about $\phi = 1/2$

```
>>> phi.setValue(GaussianNoiseVariable(mesh=mesh,
...                                         mean=0.5,
...                                         variance=0.01))
```

FiPy doesn't plot or output anything unless you tell it to: If `MayaviClient` is available, we can customize the view with a subclass of `MayaviDaemon`.

```
>>> if __name__ == "__main__":
...     try:
...         viewer = MayaviClient(vars=phi,
...                               datamin=0., datamax=1.,
...                               daemon_file="examples/cahnHilliard/sphereDaemon.py")
...     except:
...         viewer = Viewer(vars=phi,
...                         datamin=0., datamax=1.,
...                         xmin=-2.5, 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:

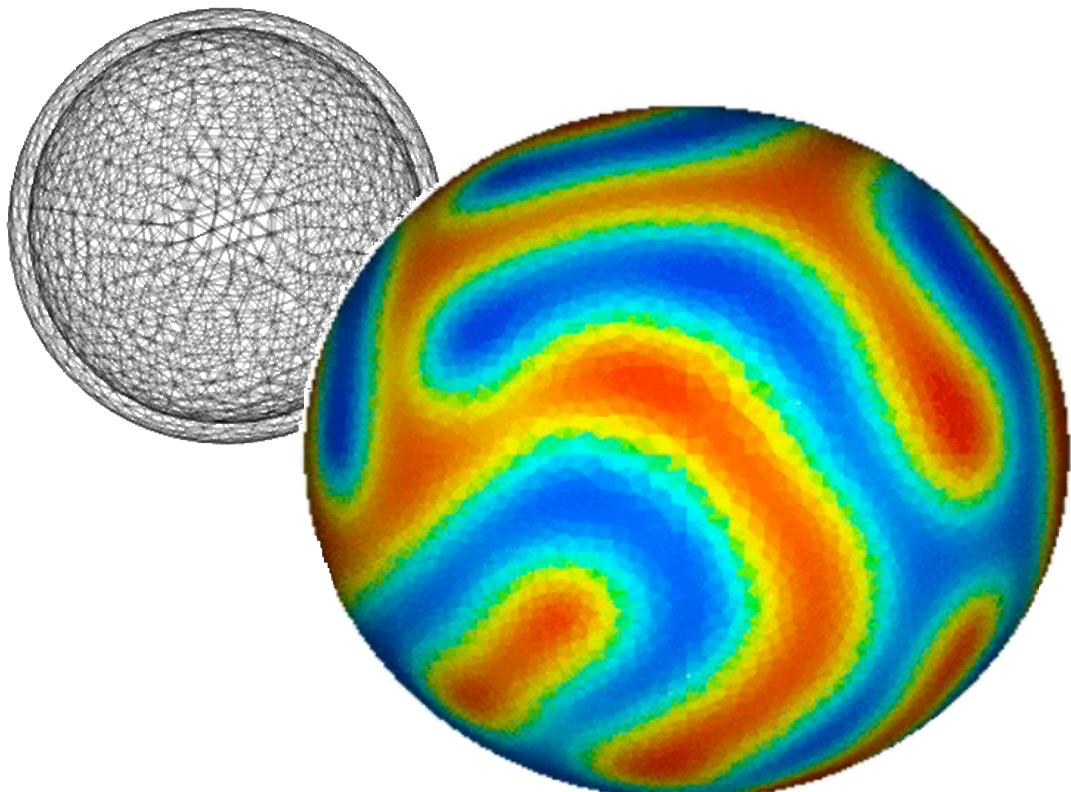
$$\frac{\partial \phi}{\partial t} = \nabla \cdot D a^2 [1 - 6\phi(1 - \phi)] \nabla \phi - \nabla \cdot D \nabla \epsilon^2 \nabla^2 \phi.$$

FiPy would automatically interpolate `D * a**2 * (1 - 6 * 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.getArithmetricFaceValue()
>>> 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, exp(dexp))
...     elapsed += dt
...     dexp += 0.01
...     eq.solve(phi, dt=dt)
...     if __name__ == "__main__":
...         viewer.plot()
```



Fluid Flow Examples

`examples.flow.stokesCavity` This example is an implementation of a rudimentary Stokes solver. It

12.1 examples.flow.stokesCavity

This example is an implementation of a rudimentary Stokes solver. It solves the Navier-Stokes equation in the viscous limit,

$$\nabla \mu \cdot \nabla \vec{u} = \nabla p$$

and the continuity equation,

$$\nabla \cdot \vec{u} = 0$$

where \vec{u} is the fluid velocity, p is the pressure and μ is the viscosity. The domain in this example is a square cavity of unit dimensions with a moving lid of unit speed. This example uses the SIMPLE algorithm with Rhie-Chow interpolation 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 [ferziger]. 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 *Dolfin* open source code on a 100 by 100 grid. The table shows the frequency of values that fall within the given error confidence bands. *Dolfin* has the added features described above. When these features are switched off the results of *Dolfin* 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 *
>>> L = 1.0
>>> N = 50
>>> dL = L / N
>>> viscosity = 1.
>>> pressureRelaxation = 0.2
```

```
>>> 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 a somewhat clumsy aspect of the `FiPy` interface that needs improvement.

```
>>> velocity = FaceVariable(mesh=mesh, rank=1)
```

Build the Stokes equations.

```
>>> xVelocityEq = DiffusionTerm(coeff=viscosity) - pressure.getGrad().dot([1, 0])
>>> yVelocityEq = DiffusionTerm(coeff=viscosity) - pressure.getGrad().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}^* . 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,

$$\nabla \mu \cdot \nabla \vec{u}' = \vec{p}'$$

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 is,

$$a_P \vec{u}'_P = \sum_f a_A \vec{u}'_A - V_P (\nabla p')_P$$

where notation from [Linear Equations](#) is used. The SIMPLE algorithm drops the second term in the above equation to leave,

$$\vec{u}'_P = -\frac{V_P (\nabla p')_P}{a_P}$$

By substituting the above expression into the continuity equations we obtain the pressure correction equation,

$$\nabla \frac{V_P}{a_P} \cdot \nabla p' = \nabla \cdot \vec{u}^*$$

In the discretized version of the above equation V_P/a_P is approximated at the face by $A_f d_{AP}/(a_P)_f$. In `FiPy` the pressure correction equation can be written as,

```
>>> ap = CellVariable(mesh=mesh)
>>> coeff = mesh._getFaceAreas() * mesh._getCellDistances() / ap.getArithmeticFaceValue()
>>> pressureCorrectionEq = DiffusionTerm(coeff=coeff) - velocity.getDivergence()
```

Set up the no-slip boundary conditions

```
>>> bcs = (FixedValue(faces=mesh.getFacesLeft(), value=0),
...          FixedValue(faces=mesh.getFacesRight(), value=0),
...          FixedValue(faces=mesh.getFacesBottom(), value=0),)
>>> bcsX = bcs + (FixedValue(faces=mesh.getFacesTop(), value=1),)
>>> bcsY = bcs + (FixedValue(faces=mesh.getFacesTop(), value=0),)
```

Set up the viewers,

```
>>> if __name__ == '__main__':
...     viewer = Viewer(vars=(pressure, xVelocity, yVelocity, velocity))
```

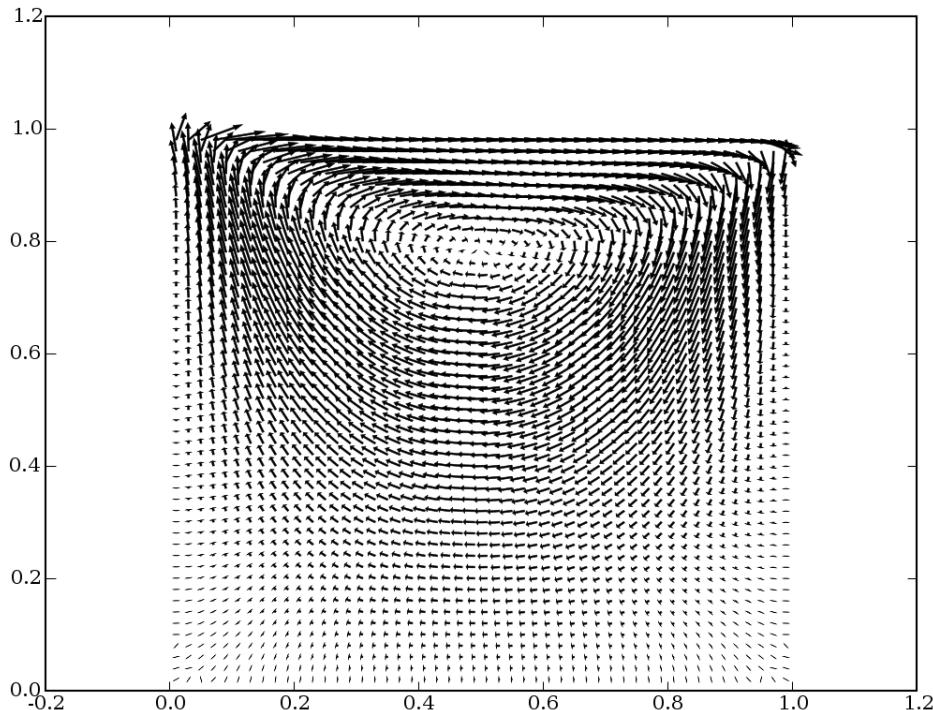
Below, we iterate for a set number of sweeps. We use the `sweep()` method instead of `solve()` because we require the residual for output. We also use the `cacheMatrix()`, `getMatrix()`, `cacheRHSvector()` and `getRHSvector()` because both the matrix and RHS vector are required by the SIMPLE algorithm. Additionally, the `sweep()` method is passed an underRelaxation factor to relax the solution. This argument cannot be passed to `solve()`.

```
>>> for sweep in range(sweeps):
...
...     ## solve the Stokes equations to get starred values
...     xVelocityEq.cacheMatrix()
...     xres = xVelocityEq.sweep(var=xVelocity,
...                               boundaryConditions=bcsX,
...                               underRelaxation=velocityRelaxation)
...     xmat = xVelocityEq.getMatrix()
...
...     yres = yVelocityEq.sweep(var=yVelocity,
...                               boundaryConditions=bcsY,
...                               underRelaxation=velocityRelaxation)
...
...     ## update the ap coefficient from the matrix diagonal
...     ap[:] = -xmat.takeDiagonal()
...
...     ## update the face velocities based on starred values
...     velocity[0] = xVelocity.getArithmeticFaceValue()
...     velocity[1] = yVelocity.getArithmeticFaceValue()
...     velocity[..., mesh.getExteriorFaces().getValue()] = 0.
...
...     ## solve the pressure correction equation
...     pressureCorrectionEq.cacheRHSvector()
...     pres = pressureCorrectionEq.sweep(var=pressureCorrection)
...     rhs = pressureCorrectionEq.getRHSvector()
...
...     ## update the pressure using the corrected value but hold one cell fixed
...     pressure.setValue(pressure + pressureRelaxation * \
...                      (pressureCorrection - pressureCorrection.getGlobalValue()))
...
...     ## update the velocity using the corrected pressure
...     xVelocity.setValue(xVelocity - pressureCorrection.getGrad()[0] / \
...                       ap * mesh.getCellVolumes())
...
...     yVelocity.setValue(yVelocity - pressureCorrection.getGrad()[1] / \
```

```
...
ap * mesh.getCellVolumes()))

...
if __name__ == '__main__':
    if sweep%1 == 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.getGlobalValue() [....,-1], 145.233883763)
1
>>> print numerix.allclose(xVelocity.getGlobalValue() [....,-1], 0.24964673696)
1
>>> print numerix.allclose(yVelocity.getGlobalValue() [....,-1], -0.164498041783)
1
```

Updating FiPy

[examples.updating.update0_1to1](#) It seems unlikely that many users are still running *FiPy* 0.1, but for those
[examples.updating.update1_0to2](#) How to update scripts from version 1.0 to 2.0.

13.1 examples.updating.update0_1to1_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
...
ImportError: No module named equations.stdyConvDiffScEquation
>>> eq = SteadyConvectionDiffusionScEquation(
...     var = var,
...     diffusionCoeff = diffCoeff,
...     convectionCoeff = convCoeff,
...     solver = LinearLUSSolver(tolerance = 1.e-15, steps = 2000),
...     convectionScheme = ExponentialConvectionTerm,
...     boundaryConditions = boundaryConditions
... )
...
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,))
...
NameError: name 'eq' is not defined
>>> it.timestep()
...
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
...
ImportError: No module named grid2DGistViewer

...     viewer = Grid2DGistViewer(var)
...     viewer.plot()
```

We see that a number of errors are thrown:

- ImportError: No module named equations.stdyConvDiffScEquation
- NameError: name 'SteadyConvectionDiffusionScEquation' is not defined
- NameError: name 'eq' is not defined
- NameError: name 'it' is not defined
- ImportError: No module named grid2DGistViewer

As is usually the case with computer programming, many of these errors are caused by earlier errors. Let us update the script, section by section:

Although no error was generated by the use of `Grid2D`, *FiPy* 1.0 supports a true 1D mesh class, so we instantiate the mesh as

```
>>> L = 10.
>>> nx = 1000
>>> from fipy.meshes.grid1D import Grid1D
>>> mesh = Grid1D(dx = L / nx, nx = nx)
```

The `Grid2D` class with $ny = 1$ still works perfectly well for 1D problems, but the `Grid1D` class is slightly more efficient, and it makes the code clearer when a 1D geometry is actually desired.

Because the mesh is now 1D, we must update the convection coefficient vector to be 1D as well

```
>>> diffCoeff = 1.
>>> convCoeff = (10.,)
```

The `FixedValue` boundary conditions at the left and right are unchanged, but a `Grid1D` mesh does not even have top and bottom faces:

```
>>> valueLeft = 0.
>>> valueRight = 1.
>>> from fipy.boundaryConditions.fixedValue import FixedValue
>>> boundaryConditions = (
```

```
...     FixedValue(mesh.getFacesLeft(), valueLeft),
...     FixedValue(mesh.getFacesRight(), valueRight))
```

The creation of the solution variable is unchanged:

```
>>> from fipy.variables.cellVariable import CellVariable
>>> var = CellVariable(name = "concentration",
...                      mesh = mesh,
...                      value = valueLeft)
```

The biggest change between *FiPy* 0.1 and *FiPy* 1.0 is that *Equation* objects no longer exist at all. Instead, *Term* objects can be simply added, subtracted, and equated to assemble an equation. Where before the assembly of the equation occurred in the black-box of *SteadyConvectionDiffusionScEquation*, we now assemble it directly:

```
>>> from fipy.terms.implicitDiffusionTerm import ImplicitDiffusionTerm
>>> diffTerm = ImplicitDiffusionTerm(coeff = diffCoeff)

>>> from fipy.terms.exponentialConvectionTerm import ExponentialConvectionTerm
>>> eq = diffTerm + ExponentialConvectionTerm(coeff = convCoeff,
...                                              diffusionTerm = diffTerm)
... 
```

One thing that *SteadyConvectionDiffusionScEquation* took care of automatically was that a *ConvectionTerm* must know about any *DiffusionTerm* in the equation in order to calculate a Peclet 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.

13.2 examples.updating.update1_0to2_0

How to update scripts from version 1.0 to 2.0.

FiPy 2.0 introduces several syntax changes from *FiPy* 1.0. We appreciate that this is very inconvenient for our users, but we hope you’ll agree that the new syntax is easier to read and easier to use. We assure you that this is not something we do casually; it has been over three years since our last incompatible change (when *FiPy* 1.0 superceded *FiPy* 0.1).

All examples included with version 2.0 have been updated to use the new syntax, but any scripts you have written for *FiPy* 1.0 will need to be updated. A complete listing of the changes needed to take the *FiPy* examples scripts from version 1.0 to version 2.0 can be found at

http://www.matforge.org/fipy/wiki/upgrade1_0examplesTo2_0

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.getArithmetricFaceValue()
...             < sin(gamma.getArithmetricFaceValue())))), 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.getCellCenters()
>>> initialArray[(x < dx) | (x > (Lx - dx)) | (y < dy) | (y > (Ly - dy))] = 1.
```

instead of the *much* slower

```
>>> def cellFilter(cell):
...     return ((cell.getCenter()[0] < dx)
...             or (cell.getCenter()[0] > (Lx - dx))
...             or (cell.getCenter()[1] < dy)
...             or (cell.getCenter()[1] > (Ly - dy)))
...
>>> positiveCells = mesh.getCells(filter=cellFilter)
>>> for cell in positiveCells:
...     initialArray[cell.getID()] = 1.
```

Although they still exist, we find very little cause to ever call `getCells()` or `fipy.meshes.numMesh.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 Peclet 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`.

Part III

fipy Package Documentation

Chapter 14

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.

boundaryConditions Package Documentation

This page contains the boundaryConditions Package documentation.

15.1 The boundaryCondition Module

class BoundaryCondition (faces, value)
Generic boundary condition base class.

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

Parameters

- *faces*: A *list* or *tuple* of *Face* objects to which this condition applies.
- *value*: The value to impose.

The *BoundaryCondition* class should raise an error when invoked with internal faces. Don't use the *BoundaryCondition* class in this manner. This is merely a test.

```
>>> from fipy.meshes.grid1D import Grid1D
>>> mesh = Grid1D(nx = 2)
>>> from fipy.tools import parallel
>>> if parallel.procID == 0:
...     bc = __BoundaryCondition(mesh.getInteriorFaces(), 0)
... else:
...     raise IndexError("Face list has interior faces")
Traceback (most recent call last):
...
IndexError: Face list has interior faces
```

15.2 The fixedFlux Module

class FixedFlux (faces, value)
Bases: `fipy.boundaryConditions.boundaryCondition.BoundaryCondition`

The *FixedFlux* boundary condition adds a contribution, equivalent to a fixed flux (Neumann condition), to the equation's RHS vector. The contribution, given by *value*, is only added to entries corresponding to the specified *faces*, and is weighted by the face areas.

Creates a *FixedFlux* object.

Parameters

- *faces*: A *list* or *tuple* of *Face* objects to which this condition applies.
- *value*: The value to impose.

15.3 The `fixedValue` Module

```
class FixedValue(faces, value)
```

Bases: `fipy.boundaryConditions.boundaryCondition.BoundaryCondition`

The *FixedValue* boundary condition adds a contribution, equivalent to a fixed value (Dirichlet condition), to the equation's RHS vector and coefficient matrix. The contributions are given by $-value \times G_{face}$ for the RHS vector and G_{face} for the coefficient matrix. The parameter G_{face} represents the term's geometric coefficient, which depends on the type of term and the mesh geometry.

Contributions are only added to entries corresponding to the specified faces.

Parameters

- *faces*: A *list* or *tuple* of *Face* objects to which this condition applies.
- *value*: The value to impose.

The *BoundaryCondition* class should raise an error when invoked with internal faces. Don't use the *BoundaryCondition* class in this manner. This is merely a test.

```
>>> from fipy.meshes.grid1D import Grid1D
>>> mesh = Grid1D(nx = 2)
>>> from fipy.tools import parallel
>>> if parallel.procID == 0:
...     bc = __BoundaryCondition(mesh.getInteriorFaces(), 0)
... else:
...     raise IndexError("Face list has interior faces")
Traceback (most recent call last):
...
IndexError: Face list has interior faces
```

15.4 The `nthOrderBoundaryCondition` Module

```
class NthOrderBoundaryCondition(faces, value, order)
```

Bases: `fipy.boundaryConditions.boundaryCondition.BoundaryCondition`

This boundary condition is generally used in conjunction with a *ImplicitDiffusionTerm* that has multiple coefficients. It does not have any direct effect on the solution matrices, but its derivatives do.

Creates an *NthOrderBoundaryCondition*.

Parameters

- *faces*: A *list* or *tuple* of *Face* objects to which this condition applies.
- *value*: The value to impose.
- *order*: The 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.

15.5 The test Module

Test numeric implementation of the mesh

meshes Package Documentation

This page contains the meshes Package documentation.

16.1 common Package Documentation

This page contains the common Package documentation.

16.1.1 The mesh Module

class Mesh()

Generic mesh class defining implementation-agnostic behavior.

Make changes to mesh here first, then implement specific implementations in *pyMesh* and *numMesh*.

Meshes contain cells, faces, and vertices.

getCellCenters()

getCellVolumes()

getDim()

getExteriorFaces()

getFacesBack()

Return list of faces on back boundary of Grid3D 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.)
>>> from fipy.tools import parallel
>>> print parallel.procID > 0 or numerix.allequal((6, 7, 8, 9, 10, 11),
...                                                 numerix.nonzero(mesh.getFacesBack())[0])
True
```

getFacesBottom()

Return list of faces on bottom boundary of Grid3D with the y-axis running from bottom to top.

```
>>> from fipy import Grid2D, Grid3D, numerix
>>> mesh = Grid3D(nx = 3, ny = 2, nz = 1, dx = 0.5, dy = 2., dz = 4.)
>>> from fipy.tools import parallel
>>> print parallel.procID > 0 or numerix.allequal((12, 13, 14),
...                                                 numerix.nonzero(mesh.getFacesBottom())[0])
1
>>> x, y, z = mesh.getFaceCenters()
>>> print parallel.procID > 0 or numerix.allequal((12, 13),
```

```
... numerix.nonzero(mesh.getFacesBottom() & (x < 1))[0])  
1
```

getFacesDown()

Return list of faces on bottom boundary of Grid3D with the y-axis running from bottom to top.

```
>>> from fipy import Grid2D, Grid3D, numerix  
>>> mesh = Grid3D(nx = 3, ny = 2, nz = 1, dx = 0.5, dy = 2., dz = 4.)  
>>> from fipy.tools import parallel  
>>> print parallel.procID > 0 or numerix.allequal((12, 13, 14),  
... numerix.nonzero(mesh.getFacesBottom())[0])  
1  
>>> x, y, z = mesh.getFaceCenters()  
>>> print parallel.procID > 0 or numerix.allequal((12, 13),  
... numerix.nonzero(mesh.getFacesBottom() & (x < 1))[0])  
1
```

getFacesFront()

Return list of faces on front boundary of Grid3D 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.)  
>>> from fipy.tools import parallel  
>>> print parallel.procID > 0 or numerix.allequal((0, 1, 2, 3, 4, 5),  
... numerix.nonzero(mesh.getFacesFront())[0])  
True
```

getFacesLeft()

Return face on left boundary of Grid1D 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.)  
>>> from fipy.tools import parallel  
>>> print parallel.procID > 0 or numerix.allequal((21, 25),  
... numerix.nonzero(mesh.getFacesLeft())[0])  
True  
>>> mesh = Grid2D(nx = 3, ny = 2, dx = 0.5, dy = 2.)  
>>> print parallel.procID > 0 or numerix.allequal((9, 13),  
... numerix.nonzero(mesh.getFacesLeft())[0])  
True
```

getFacesRight()

Return list of faces on right boundary of Grid3D 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.)  
>>> from fipy.tools import parallel  
>>> print parallel.procID > 0 or numerix.allequal((24, 28),  
... numerix.nonzero(mesh.getFacesRight())[0])  
True  
>>> mesh = Grid2D(nx = 3, ny = 2, dx = 0.5, dy = 2.)  
>>> print parallel.procID > 0 or numerix.allequal((12, 16),  
... numerix.nonzero(mesh.getFacesRight())[0])  
True
```

getFacesTop()

Return list of faces on top boundary of Grid3D with the y-axis running from bottom to top.

```

>>> from fipy import Grid2D, Grid3D, numerix
>>> mesh = Grid3D(nx = 3, ny = 2, nz = 1, dx = 0.5, dy = 2., dz = 4.)
>>> from fipy.tools import parallel
>>> print parallel.procID > 0 or numerix.allequal((18, 19, 20),
...                                                 numerix.nonzero(mesh.getFacesTop())[0])
True
>>> mesh = Grid2D(nx = 3, ny = 2, dx = 0.5, dy = 2.)
>>> print parallel.procID > 0 or numerix.allequal((6, 7, 8),
...                                                 numerix.nonzero(mesh.getFacesTop())[0])
...
True

getFacesUp()
Return list of faces on top boundary of Grid3D with the y-axis running from bottom to top.

>>> from fipy import Grid2D, Grid3D, numerix
>>> mesh = Grid3D(nx = 3, ny = 2, nz = 1, dx = 0.5, dy = 2., dz = 4.)
>>> from fipy.tools import parallel
>>> print parallel.procID > 0 or numerix.allequal((18, 19, 20),
...                                                 numerix.nonzero(mesh.getFacesTop())[0])
True
>>> mesh = Grid2D(nx = 3, ny = 2, dx = 0.5, dy = 2.)
>>> print parallel.procID > 0 or numerix.allequal((6, 7, 8),
...                                                 numerix.nonzero(mesh.getFacesTop())[0])
...
True

getInteriorFaces()
getNearestCell(point)
getNumberOfCells()
setScale(value=1.0)

```

16.2 numMesh Package Documentation

This page contains the numMesh Package documentation.

16.2.1 The `cell` Module

`class Cell(mesh, id)`

```

getCenter()
getID()
getMesh()
getNormal(index)

```

16.2.2 The `cylindricalGrid1D` Module

1D Mesh

class CylindricalGrid1D (dx=1.0, nx=None, origin=(0,), overlap=2)

Bases: `fipy.meshes.numMesh.grid1D.Grid1D`

Creates a 1D cylindrical grid mesh.

```
>>> mesh = CylindricalGrid1D(nx = 3)
>>> print mesh.getCellCenters()
[[ 0.5  1.5  2.5]]

>>> mesh = CylindricalGrid1D(dx = (1, 2, 3))
>>> print mesh.getCellCenters()
[[ 0.5  2.   4.5]]

>>> mesh = CylindricalGrid1D(nx = 2, dx = (1, 2, 3))
...
IndexError: nx != len(dx)

getCellCenters()
getFaceCenters()
getVertexCoords()
```

16.2.3 The `cylindricalGrid2D` Module

2D rectangular Mesh

class CylindricalGrid2D (dx=1.0, dy=1.0, nx=None, ny=None, origin=((0.0,), (0.0,)), overlap=2, parallelModule=<fipy.tools.Parallel object at 0x1745c30>)

Bases: `fipy.meshes.numMesh.grid2D.Grid2D`

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

```
getCellCenters()
getCellVolumes()
getFaceCenters()
getVertexCoords()
```

16.2.4 The `cylindricalUniformGrid1D` Module

1D Mesh

class CylindricalUniformGrid1D (dx=1.0, nx=1, origin=(0,), overlap=2)

Bases: `fipy.meshes.numMesh.uniformGrid1D.UniformGrid1D`

Creates a 1D cylindrical grid mesh.

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

getCellVolumes()
```

16.2.5 The cylindricalUniformGrid2D Module

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

```
class CylindricalUniformGrid2D (dx=1.0, dy=1.0, nx=1, ny=1, origin=((0, ), (0, )), overlap=2, parallelMode=<fipy.tools.Parallel object at 0x1745c30>)
Bases: fipy.meshes.numMesh.uniformGrid2D.UniformGrid2D
```

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

```
getCellVolumes()
```

16.2.6 The face Module

```
class Face (mesh, id)
```

Face within a *Mesh*

Face objects are bounded by *Vertex* objects. *Face* objects separate *Cell* objects.

Face is initialized by *Mesh*

Parameters

- *mesh*: the *Mesh* that contains this *Face*
- *id*: a unique identifier

```
getArea()
```

```
getCellID (index=0)
```

Return the *id* of the specified *Cell* on one side of this *Face*.

```
getCenter()
```

Return the coordinates of the *Face* center.

```
getID()
```

```
getMesh()
```

16.2.7 The gmshExport Module

This module takes a FiPy mesh and creates a mesh file that can be opened in Gmsh.

```
exception MeshExportError
```

Bases: exceptions.Exception

```
exportAsMesh (mesh, filename)
```

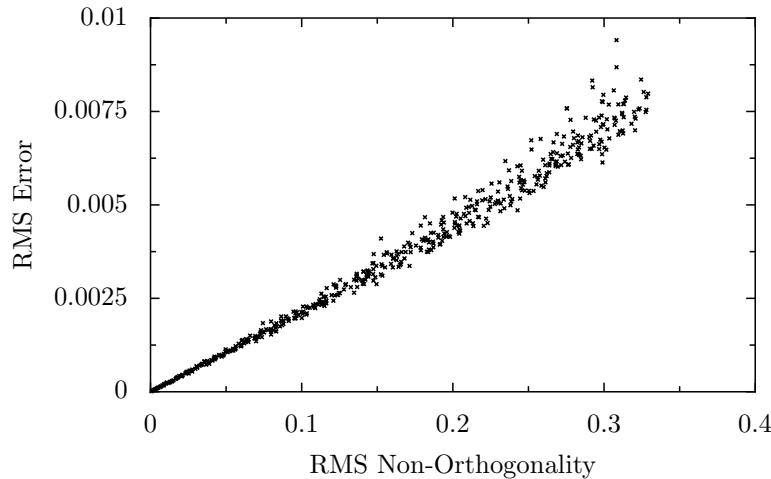
16.2.8 The gmshImport Module

This module takes a Gmsh output file (*.msh*) and converts it into a FiPy mesh. This currently supports triangular and tetrahedral meshes only.

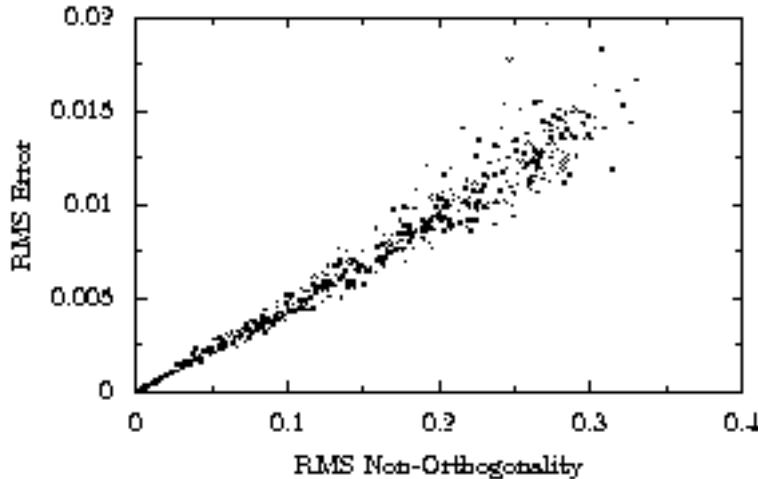
Gmsh generates unstructured meshes, which may contain a significant amount of non-orthogonality and it is very difficult to directly control the amount of non-orthogonality simply by manipulating Gmsh parameters. Therefore, it is necessary to take into account the possibility of errors arising due to the non-orthogonality of the mesh. To test the degree of error, an experiment was conducted using a simple 1D diffusion problem with constant diffusion coefficients and boundary conditions as follows: fixed value of 0 on the left side, fixed value of 1 on the right side, and a fixed flux of 0 on the top and bottom sides. The analytical solution is clearly a uniform gradient going from left to right.

this problem was implemented using a Cartesian Grid2D mesh with each interior vertex displaced a short distance in a random direction to create non-orthogonality. Then the root-mean-square error was plotted against the root-mean-square non-orthogonality. The error in each cell was calculated by simply subtracting the analytical solution at each cell center from the calculated value for that cell. The non-orthogonality in each cell is the average, weighted by face area, of the sines of the angles between the face normals and the line segments joining the cells. Thus, the non-orthogonality of a cell can range from 0 (every face is orthogonal to its corresponding cell-to-cell line segment) to 1 (only possible in a degenerate case). This test was run using 500 separate 20x20 meshes and 500 separate 10x10 meshes, each with the interior vertices moved different amounts so as to created different levels of non-orthogonality. The results are shown below.

Results for 20x20 mesh:



Results for 10x10 mesh:



It is clear from the graphs that finer meshes decrease the error due to non-orthogonality, and that even with a reasonably coarse mesh the error is quite low. However, note that this test is only for a simple 1D diffusion problem with a constant diffusion coefficient, and it is unknown whether the results will be significantly different with more complicated problems.

Test cases:

```
>>> newmesh = GmshImporter3D('fipy/meshes/numMesh/testgmsh.msh')
>>> print newmesh.getVertexCoords()
```

```

[[ 0.    0.5   1.    0.5   0.5]
 [ 0.    0.5   0.    1.    0.5]
 [ 0.    1.    0.    0.    0.5]]]

>>> print newmesh._getFaceVertexIDs()
[[2 4 4 4 3 4 4 3 4 3]
 [1 1 2 2 1 3 3 2 3 2]
 [0 0 0 1 0 0 1 0 2 1]]

>>> print newmesh._getCellFaceIDs()
[[0 4 7 9]
 [1 1 2 3]
 [2 5 5 6]
 [3 6 8 8]]

>>> mesh = GmshImporter2DIn3DSpace('fipy/meshes/numMesh/GmshTest2D.msh')
>>> print mesh.getVertexCoords()
[[ 0.    1.    0.5   0.    1.    0.5   0.    1. ]
 [ 0.    0.    0.5   1.    1.    1.5   2.    2. ]
 [ 0.    0.    0.    0.    0.    0.    0.    0. ]]

>>> mesh = GmshImporter2D('fipy/meshes/numMesh/GmshTest2D.msh')
>>> print mesh.getVertexCoords()
[[ 0.    1.    0.5   0.    1.    0.5   0.    1. ]
 [ 0.    0.    0.5   1.    1.    1.5   2.    2. ]]

>>> print mesh._getFaceVertexIDs()
[[2 0 1 0 3 1 4 4 3 5 3 6 5 7 7]
 [0 1 2 3 2 4 2 3 5 4 6 5 7 4 6]]

>>> print (mesh._getCellFaceIDs() == [[0, 0, 2, 7, 7, 8, 12, 14],
...                                     [1, 3, 5, 4, 8, 10, 13, 11],
...                                     [2, 4, 6, 6, 9, 11, 9, 12]]).flatten().all()
True

```

The following test case is to test the handedness of the mesh to check it does not return negative volumes. Firstly we set up a list with tuples of strings to be read by gmsh. The list provide instructions to gmsh to form a circular mesh.

```

>>> cellSize = 0.7
>>> radius = 1.
>>> lines = ['cellSize = ' + str(cellSize) + ';\n',
...            'radius = ' + str(radius) + ';\n',
...            'Point(1) = {0, 0, 0, cellSize};\n',
...            'Point(2) = {-radius, 0, 0, cellSize};\n',
...            'Point(3) = {0, radius, 0, cellSize};\n',
...            'Point(4) = {radius, 0, 0, cellSize};\n',
...            'Point(5) = {0, -radius, 0, cellSize};\n',
...            'Circle(6) = {2, 1, 3};\n',
...            'Circle(7) = {3, 1, 4};\n',
...            'Circle(8) = {4, 1, 5};\n',
...            'Circle(9) = {5, 1, 2};\n',
...            'Line Loop(10) = {6, 7, 8, 9} ;\n',
...            'Plane Surface(11) = {10};\n'

```

Check that the sign of the mesh volumes is correct

```
>>> mesh = GmshImporter2D(lines)
>>> print mesh.getCellVolumes()[0] > 0
1
```

Reverse the handedness of the mesh and check the sign

```
>>> lines[7:12] = ['Circle(6) = {3, 1, 2};\n',
...                 'Circle(7) = {4, 1, 3};\n',
...                 'Circle(8) = {5, 1, 4};\n',
...                 'Circle(9) = {2, 1, 5};\n',
...                 'Line Loop(10) = {9, 8, 7, 6};\n',]

>>> mesh = GmshImporter2D(lines)
>>> print mesh.getCellVolumes()[0] > 0
1

class GmshImporter2D(arg, coordDimensions=2)
    Bases: fipy.meshes.numMesh.mesh2D.Mesh2D
    getCellVolumes()

class GmshImporter2DIn3DSpace(arg)
    Bases: fipy.meshes.numMesh.gmshImport.GmshImporter2D

class GmshImporter3D(arg)
    Bases: fipy.meshes.numMesh.mesh.Mesh

    >>> mesh = GmshImporter3D('fipy/meshes/numMesh/testgmsh.msh')

    getCellVolumes()

exception MeshImportError
    Bases: exceptions.Exception

class MshFile(arg)

    getFilename()
    remove()
```

16.2.9 The grid1D Module

1D Mesh

```
class Grid1D(dx=1.0, nx=None, overlap=2, parallelModule=<fipy.tools.Parallel object at 0x1745c30>)
    Bases: fipy.meshes.numMesh.mesh1D.Mesh1D
```

Creates a 1D grid mesh.

```
>>> mesh = Grid1D(nx = 3)
>>> print mesh.getCellCenters()
[[ 0.5  1.5  2.5]]
```

```

>>> mesh = Grid1D(dx = (1, 2, 3))
>>> print mesh.getCellCenters()
[[ 0.5  2.   4.5]]

>>> mesh = Grid1D(nx = 2, dx = (1, 2, 3))
...
IndexError: nx != len(dx)

getDim()
getPhysicalShape()
    Return physical dimensions of Grid1D.

getScale()
getShape()

```

16.2.10 The grid2D Module

2D rectangular Mesh

```

class Grid2D(dx=1.0, dy=1.0, nx=None, ny=None, overlap=2, parallelModule=<fipy.tools.Parallel object at 0x1745c30>)
Bases: fipy.meshes.numMesh.mesh2D.Mesh2D

```

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

```

getPhysicalShape()
    Return physical dimensions of Grid2D.

getScale()
getShape()

```

16.2.11 The grid3D Module

```

class Grid3D(dx=1.0, dy=1.0, dz=1.0, nx=None, ny=None, nz=None, overlap=2, parallelModule=<fipy.tools.Parallel object at 0x1745c30>)
Bases: fipy.meshes.numMesh.mesh.Mesh

```

3D rectangular-prism Mesh

X axis runs from left to right. Y axis runs from bottom to top. Z axis runs from front to back.

Numbering System:

Vertices: Numbered in the usual way. X coordinate changes most quickly, then Y, then Z.

Cells: Same numbering system as vertices.

Faces: XY faces numbered first, then XZ faces, then YZ faces. Within each subcategory, it is numbered in the usual way.

```

getPhysicalShape()
    Return physical dimensions of Grid3D.

getScale()
getShape()

```

16.2.12 The mesh Module

```
class Mesh (vertexCoords, faceVertexIDs, cellFaceIDs)
    Bases: fipy.meshes.common.mesh.Mesh

    Generic mesh class using numerix to do the calculations

    Meshes contain cells, faces, and vertices.

    This is built for a non-mixed element mesh.

    faceVertexIDs and cellFacesIDs must be padded with minus ones.

    getExteriorFaces ()
        Return only the faces that have one neighboring cell.

    getFaceCellIDs ()

    getFaceCenters ()

    getInteriorFaces ()
        Return only the faces that have two neighboring cells.

    getVTKCellDataSet ()
        Returns a TVTK DataSet representing the cells of this mesh

    getVTKFaceDataSet ()
        Returns a TVTK DataSet representing the face centers of this mesh

    getVertexCoords ()

exception MeshAdditionError
    Bases: exceptions.Exception
```

16.2.13 The mesh1D Module

Generic mesh class using numerix to do the calculations
Meshes contain cells, faces, and vertices.
This is built for a non-mixed element mesh.

```
class Mesh1D (vertexCoords, faceVertexIDs, cellFaceIDs)
    Bases: fipy.meshes.numMesh.mesh.Mesh

    faceVertexIDs and cellFacesIDs must be padded with minus ones.
```

16.2.14 The mesh2D Module

Generic mesh class using numerix to do the calculations
Meshes contain cells, faces, and vertices.
This is built for a non-mixed element mesh.

```
class Mesh2D (vertexCoords, faceVertexIDs, cellFaceIDs)
    Bases: fipy.meshes.numMesh.mesh.Mesh

    faceVertexIDs and cellFacesIDs must be padded with minus ones.

extrude (extrudeFunc=<function <lambda> at 0x250f7f0>, layers=1)
    This function returns a new 3D mesh. The 2D mesh is extruded using the extrudeFunc and the number of
    layers.
```

Parameters

- *extrudeFunc*: function that takes the vertex coordinates and returns the displaced values
- *layers*: the number of layers in the extruded mesh (number of times *extrudeFunc* will be called)

```
>>> from fipy.meshes.grid2D import Grid2D
>>> print Grid2D(nx=2,ny=2).extrude(layers=2).getCellCenters()
[[ 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).getCellCenters()
[[ 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           ]]]
```

16.2.15 The periodicGrid1D Module

Peridoic 1D Mesh

```
class PeriodicGrid1D(dx=1.0, nx=None)
Bases: fipy.meshes.numMesh.grid1D.Grid1D

>>> from fipy import numerix
>>> from fipy.tools import parallel
```

Creates a Periodic grid mesh.

```
>>> mesh = PeriodicGrid1D(dx = (1, 2, 3))

>>> print (parallel.procID > 0
...       or numerix.allclose(numerix.nonzero(mesh.getExteriorFaces())[0],
...                               [3]))
True

>>> print (parallel.procID > 0
...       or numerix.allclose(mesh.getFaceCellIDs().filled(-999),
...                           [[2, 0, 1, 2],
...                            [0, 1, 2, -999]]))
True

>>> print (parallel.procID > 0
...       or numerix.allclose(mesh._getCellDistances(),
...                           [ 2., 1.5, 2.5, 1.5]))
```

```
>>> print (parallel.procID > 0
...       or numerix.allclose(mesh._getCellToCellDistances(),
...                           [[ 2.,    1.5,   2.5],
...                            [ 1.5,   2.5,   2. ]]))
True

>>> print (parallel.procID > 0
...       or numerix.allclose(mesh._getFaceNormals(),
...                           [[ 1.,   1.,   1.,   1.])))
True

>>> print (parallel.procID > 0
...       or numerix.allclose(mesh._getCellVertexIDs(),
...                           [[1, 2, 2],
...                            [0, 1, 0]]))
True

getCellCenters()
```

16.2.16 The `periodicGrid2D` Module

2D periodic rectangular Mesh

```
class PeriodicGrid2D(dx=1.0, dy=1.0, nx=None, ny=None)
Bases: fipy.meshes.numMesh.grid2D.Grid2D
```

Creates a periodic2D grid mesh with horizontal faces numbered first and then vertical faces. Vertices and cells are numbered in the usual way.

```
>>> from fipy import numerix
>>> from fipy.tools import parallel

>>> mesh = PeriodicGrid2D(dx = 1., dy = 0.5, nx = 2, ny = 2)

>>> print (parallel.procID > 0 or
...       numerix.allclose(numerix.nonzero(mesh.getExteriorFaces())[0],
...                         [ 4,  5,  8, 11]))
True

>>> print (parallel.procID > 0 or
...       numerix.allclose(mesh.getFaceCellIDs().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 (parallel.procID > 0 or
...       numerix.allclose(mesh._getCellDistances(),
...                         [ 0.5, 0.5, 0.5, 0.5, 0.25, 0.25, 1., 1., 0.5, 1., 1., 0.5]))
True
```

```

>>> print (parallel.procID > 0
...     or numerix.allclose(mesh._getCellFaceIDs(),
...                         [[0, 1, 2, 3],
...                          [7, 6, 10, 9],
...                          [2, 3, 0, 1],
...                          [6, 7, 9, 10]]))
True

>>> print (parallel.procID > 0 or
...     numerix.allclose(mesh._getCellToCellDistances(),
...                       [[ 0.5, 0.5, 0.5, 0.5],
...                        [ 1., 1., 1., 1. ],
...                        [ 0.5, 0.5, 0.5, 0.5],
...                        [ 1., 1., 1., 1. ]]))
True

>>> normals = [[0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1],
...               [1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0]]
... 

>>> print (parallel.procID > 0
...     or numerix.allclose(mesh._getFaceNormals(), normals))
True

>>> print (parallel.procID > 0
...     or numerix.allclose(mesh._getCellVertexIDs(),
...                         [[4, 5, 7, 8],
...                          [3, 4, 6, 7],
...                          [1, 2, 4, 5],
...                          [0, 1, 3, 4]]))
True

class PeriodicGrid2DLeftRight (dx=1.0, dy=1.0, nx=None, ny=None)
    Bases: fipy.meshes.numMesh.periodicGrid2D.PeriodicGrid2D

class PeriodicGrid2DTopBottom (dx=1.0, dy=1.0, nx=None, ny=None)
    Bases: fipy.meshes.numMesh.periodicGrid2D.PeriodicGrid2D

```

16.2.17 The `skewedGrid2D` Module

```

class SkewedGrid2D (dx=1.0, dy=1.0, nx=None, ny=1, rand=0)
    Bases: fipy.meshes.numMesh.mesh2D.Mesh2D

```

Creates a 2D grid mesh with horizontal faces numbered first and then vertical faces. The points are skewed by a random amount (between `rand` and `-rand`) in the X and Y directions.

```

getPhysicalShape()
    Return physical dimensions of Grid2D.

getScale()
getShape()

```

16.2.18 The `test` Module

Test numeric implementation of the mesh

16.2.19 The `tri2D` Module

```
class Tri2D (dx=1.0, dy=1.0, nx=1, ny=1)
Bases: fipy.meshes.numMesh.mesh2D.Mesh2D
```

This class creates a mesh made out of triangles. It does this by starting with a standard Cartesian mesh (*Grid2D*) and dividing each cell in that mesh (hereafter referred to as a ‘box’) into four equal parts with the dividing lines being the diagonals.

Creates a 2D triangular mesh with horizontal faces numbered first then vertical faces, then diagonal faces. Vertices are numbered starting with the vertices at the corners of boxes and then the vertices at the centers of boxes. Cells on the right of boxes are numbered first, then cells on the top of boxes, then cells on the left of boxes, then cells on the bottom of boxes. Within each of the ‘sub-categories’ in the above, the vertices, cells and faces are numbered in the usual way.

Parameters

- *dx, dy*: The X and Y dimensions of each ‘box’. If *dx* \neq *dy*, the line segments connecting the cell centers will not be orthogonal to the faces.
- *nx, ny*: 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*.

`getPhysicalShape()`

Return physical dimensions of Grid2D.

`getScale()`

`getShape()`

16.2.20 The `uniformGrid1D` Module

1D Mesh

```
class UniformGrid1D (dx=1.0, nx=1, origin=(0, ), overlap=2, parallelModule=<fipy.tools.Parallel object at
0x1745c30>)
Bases: fipy.meshes.numMesh.grid1D.Grid1D
```

Creates a 1D grid mesh.

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

`getCellVolumes()`

`getFaceCellIDs()`

`getFaceCenters()`

`getInteriorFaces()`

`getVertexCoords()`

16.2.21 The `uniformGrid2D` Module

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

```
class UniformGrid2D(dx=1.0, dy=1.0, nx=1, ny=1, origin=((0, ), (0, )), overlap=2, parallelModule=<fipy.tools.Parallel object at 0x1745c30>)
Bases: fipy.meshes.numMesh.grid2D.Grid2D

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

getCellVolumes()
getExteriorFaces()
    Return only the faces that have one neighboring cell.
getFaceCellIDs()
getFaceCenters()
getInteriorFaces()
    Return only the faces that have two neighboring cells.
getVertexCoords()
```

16.2.22 The uniformGrid3D Module

```
class UniformGrid3D(dx=1.0, dy=1.0, dz=1.0, nx=1, ny=1, nz=1, origin=[[0], [0], [0]], overlap=2, parallelModule=<fipy.tools.Parallel object at 0x1745c30>)
Bases: fipy.meshes.numMesh.grid3D.Grid3D
```

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.

```
getCellVolumes()
getExteriorFaces()
    Return only the faces that have one neighboring cell.
getFaceCellIDs()
getFaceCenters()
getInteriorFaces()
    Return only the faces that have two neighboring cells
getVertexCoords()
```

16.3 pyMesh Package Documentation

This page contains the pyMesh Package documentation.

16.3.1 The `cell` Module

Cell within a mesh

class Cell (*faces*, *faceOrientations*, *id*)

Cell within a mesh

Cell objects are bounded by *Face* objects.

Cell is initialized by *Mesh*

Parameters

- *faces*: *list* or *tuple* of bounding faces that define the cell
- *faceOrientations*: *list*, *tuple*, or *numerix.array* of orientations (+/-1) to indicate whether a face points into this face or out of it. Can be calculated, but the mesh typically knows this information already.
- *id*: unique identifier

getBoundingCells ()

getCenter ()

Return the coordinates of the *Cell* center.

getFaceIDs ()

getFaceOrientations ()

getFaces ()

Return the faces bounding the *Cell*.

getID ()

Return the id of this *Cell*.

getVolume ()

Return the volume of the *Cell*.

16.3.2 The `face` Module

Face within a *Mesh*

class Face (*vertices*, *id*)

Face within a *Mesh*

Face objects are bounded by *Vertex* objects. *Face* objects separate *Cell* objects.

Face is initialized by *Mesh*

Parameters

- *vertices*: the *Vertex* points that bound the *Face*
- *id*: a unique identifier

addBoundingCell (*cell*, *orientation*)

Add *cell* to the list of *Cell* objects which lie on either side of this *Face*.

getArea ()

Return the area of the *Face*.

getCellDistance ()

Return the distance between adjacent *Cell* centers.

getCellID (*index=0*)
 Return the *id* of the specified *Cell* on one side of this *Face*.

getCells ()
 Return the *Cell* objects which lie on either side of this *Face*.

getCenter ()
 Return the coordinates of the *Face* center.

getID ()

getNormal ()
 Return the unit normal vector

16.3.3 The `face2D` Module

1D (edge) Face in a 2D Mesh

class Face2D (*vertices, id*)
 Bases: `fipy.meshes.pyMesh.face.Face`

1D (edge) Face in a 2D Mesh

Face2D is bounded by two Vertices.

Face is initialized by *Mesh*

Parameters

- *vertices*: the *Vertex* points that bound the *Face*
- *id*: a unique identifier

16.3.4 The `grid2D` Module

2D rectangular Mesh

class Grid2D (*dx, dy, nx, ny*)
 Bases: `fipy.meshes.pyMesh.mesh.Mesh`

2D rectangular Mesh

Numbering system

`nx=5`

`ny=3`

Cells:

```
*****
*   *   *   *   *
* 10  * 11  * 12  * 13  * 14  *
*****
*   *   *   *   *
* 5   * 6   * 7   * 8   * 9   *
*****
*   *   *   *   *
* 0   * 1   * 2   * 3   * 4   *
*****
```

Faces (before reordering):

```
***15*****16*****17*****18****19***
*      *      *      *      *      *
32      33      34      35      36      37
***10*****11*****12*****13*****14**
*      *      *      *      *      *
26      27      28      29      30      31
***5*****6*****7*****8*****9***
*      *      *      *      *      *
20      21      22      23      24      25
***0*****1*****2*****3*****4***
```

Faces (after reordering):

```
***27*****28*****29*****30****31***
*      *      *      *      *      *
34      18      19      20      21      37
***5*****6*****7*****8*****9***
*      *      *      *      *      *
33      14      15      16      17      36
***0*****1*****2*****3*****4***
*      *      *      *      *      *
32      10      11      12      13      35
***22*****23*****24*****25*****26**
```

Vertices:

```
18*****19*****20*****21*****22****23
*      *      *      *      *      *
*      *      *      *      *      *
12*****13*****14*****15*****16****17
*      *      *      *      *      *
*      *      *      *      *      *
6*****7*****8*****9*****10****11
*      *      *      *      *      *
*      *      *      *      *      *
0*****1*****2*****3*****4*****5
```

Grid2D is initialized by caller

Parameters

- *dx*: dimension of each cell in **x** direction
- *dy*: dimension of each cell in **y** direction
- *nx*: number of cells in **x** direction
- *ny*: number of cells in **y** direction

getCellCenters ()

getCellVolumes ()

getFacesBottom ()

Return list of faces on bottom boundary of Grid2D with the y-axis running from bottom to top.

getFacesLeft ()

Return list of faces on left boundary of Grid2D with the x-axis running from left to right.

getFacesRight ()

Return list of faces on right boundary of Grid2D with the x-axis running from left to right.

```
getFacesTop()
    Return list of faces on top boundary of Grid2D with the y-axis running from bottom to top.

getPhysicalShape()
    Return physical dimensions of Grid2D.

getShape()
    Return cell dimensions Grid2D.
```

16.3.5 The `mesh` Module

Generic mesh class

Meshes contain cells, faces, and vertices.

```
class Mesh (cells, faces, interiorFaces, vertices)
    Bases: fipy.meshes.common.Mesh

getExteriorFaces()
    Return only the faces that have one neighboring cell.

getFaceOrientations()

getPhysicalShape()
    Return physical dimensions of Mesh.

getScale()

setScale (scale)
```

16.3.6 The `test` Module

Test numeric implementation of the mesh

16.3.7 The `vertex` Module

Vertex within a Mesh

Vertices bound Faces.

```
class Vertex (coordinates)
    Vertex is initialized by Mesh with its coordinates.

getCoordinates()
    Return coordinates of Vertex.
```

16.4 The `cylindricalGrid1D` Module

`CylindricalGrid1D (dr=None, nr=None, dx=1.0, nx=None)`

16.5 The `cylindricalGrid2D` Module

`CylindricalGrid2D (dr=None, dz=None, nr=None, nz=None, dx=1.0, dy=1.0, nx=None, ny=None, parallelModule=<fipy.tools.Parallel object at 0x1745c30>)`

16.6 The `grid1D` Module

`Grid1D` ($dx=1.0$, $nx=None$, $overlap=2$, $parallelModule=<\text{fipy.tools.Parallel object at } 0x1745c30>$)

16.7 The `grid2D` Module

`Grid2D` ($dx=1.0$, $dy=1.0$, $nx=None$, $ny=None$, $overlap=2$, $parallelModule=<\text{fipy.tools.Parallel object at } 0x1745c30>$)

16.8 The `grid3D` Module

`Grid3D` ($dx=1.0$, $dy=1.0$, $dz=1.0$, $nx=None$, $ny=None$, $nz=None$, $overlap=2$, $parallelModule=<\text{fipy.tools.Parallel object at } 0x1745c30>$)

16.9 The `test` Module

Test implementation of the mesh

models Package Documentation

This page contains the models Package documentation.

17.1 levelSet Package Documentation

This page contains the levelSet Package documentation.

17.1.1 advection Package Documentation

This page contains the advection Package documentation.

The advectionEquation Module

buildAdvectionEquation (*advectionCoeff=None*, *advectionTerm=None*)

The *buildAdvectionEquation* function constructs and returns an advection equation. The advection equation is given by:

$$\frac{\partial \phi}{\partial t} + u|\nabla \phi| = 0.$$

This solution method for the *_AdvectionTerm* is set up specifically to evolve *var* while preserving *var* as a distance function. This equation is used in conjunction with the *DistanceFunction* object. Further details of the numerical method can be found in “Level Set Methods and Fast Marching Methods” by J.A. Sethian, Cambridge University Press, 1999. Testing for the advection equation is in `examples.levelSet.advection`

Parameters

- *advectionCoeff*: The coeff to pass to the *advectionTerm*.
- *advectionTerm*: An advection term class.

The advectionTerm Module

The higherOrderAdvectionEquation Module

buildHigherOrderAdvectionEquation (*advectionCoeff=None*)

The *buildHigherOrderAdvectionEquation* function returns an advection equation that uses the *_HigherOrderAdvectionTerm*. The advection equation is given by,

$$\frac{\partial \phi}{\partial t} + u|\nabla \phi| = 0.$$

Parameters

- *advectionCoeff*: The *coeff* to pass to the *_HigherOrderAdvectionTerm*

The `higherOrderAdvectionTerm` Module

17.1.2 `distanceFunction` Package Documentation

This page contains the `distanceFunction` Package documentation.

The `distanceVariable` Module

```
class DistanceVariable(mesh, name='', value=0.0, unit=None, hasOld=0, narrowBandWidth=10000000000.0)
Bases: fipy.variables.cellVariable.CellVariable
```

A *DistanceVariable* object calculates ϕ so it satisfies,

$$|\nabla\phi| = 1$$

using the fast marching method with an initial condition defined by the zero level set.

Currently the solution is first order, This suffices for initial conditions with straight edges (e.g. trenches in electrodeposition). The method should work for unstructured 2D grids but testing on unstructured grids is untested thus far. This is a 2D implementation as it stands. Extending to 3D should be relatively simple.

Here we will define a few test cases. Firstly a 1D test case

```
>>> from fipy.meshes.grid1D import Grid1D
>>> from fipy.tools import serial
>>> mesh = Grid1D(dx = .5, nx = 8, parallelModule=serial)
>>> 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, -.25, .25, .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, parallelModule=serial)
>>> 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.grid2D import Grid2D
>>> mesh = Grid2D(dx = dx, dy = dy, nx = 2, ny = 3)
>>> 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. Essentially the equation solves a fake distance function to march out the velocity from the interface.

```

>>> from fipy.variables.cellVariable import CellVariable
>>> mesh = Grid2D(dx = 1., dy = 1., nx = 2, ny = 2)
>>> 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)
>>> print extensionVar.allclose((1.25, .5, 2, 1.25))
1
>>> mesh = Grid2D(dx = 1., dy = 1., nx = 3, ny = 3)
>>> var = DistanceVariable(mesh = mesh, value = (-1, 1, 1,
...                                         1, 1, 1,
...                                         1, 1, 1))
>>> var.calcDistanceFunction()
>>> 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)
>>> 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)
>>> var = DistanceVariable(mesh = mesh, value = (-1, 1, -1))
>>> var.calcDistanceFunction()
>>> print var.allclose((-0.5, 0.5, -0.5))
1
```

For future reference, the minimum distance for the interface cells can be calculated with the following functions. The trial cell values will also be calculated with these functions. In essence it is not difficult to calculate the level set distance function on an unstructured 3D grid. However a lot of testing will be required. The minimum distance functions will take the following form.

$$X_{\min} = \frac{|\vec{s} \times \vec{t}|}{|\vec{s} - \vec{t}|}$$

and in 3D,

$$X_{\min} = \frac{1}{3!} |\vec{s} \cdot (\vec{t} \times \vec{u})|$$

where the vectors \vec{s} , \vec{t} and \vec{u} represent the vectors from the cell of interest to the neighboring cell.

Creates a *distanceVariable* object.

Parameters

- *mesh*: The mesh that defines the geometry of this variable.
- *name*: The name of the variable.
- *value*: The initial value.
- *unit*: the physical units of the variable
- *hasOld*: Whether the variable maintains an old value.
- *narrowBandWidth*: The width of the region about the zero level set within which the distance function is evaluated.

calcDistanceFunction (*narrowBandWidth=None*, *deleteIslands=False*)

Calculates the *distanceVariable* as a distance function.

Parameters

- *narrowBandWidth*: The width of the region about the zero level set within which the distance function is evaluated.
- *deleteIslands*: Sets the temporary level set value to zero in isolated cells.

extendVariable (*extensionVariable*, *deleteIslands=False*)

Takes a *cellVariable* and extends the variable from the zero to the region encapsulated by the *narrowBandWidth*.

Parameters

- *extensionVariable*: The variable to extend from the zero level set.
- *deleteIslands*: Sets the temporary level set value to zero in isolated cells.

getCellInterfaceAreas ()

Returns the length of the interface that crosses the cell

A simple 1D test:

```
>>> from fipy.meshes.grid1D 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.getCellInterfaceAreas(),
...                         answer)
...
True
```

A 2D test case:

```
>>> from fipy.meshes.grid2D 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.getCellInterfaceAreas(), 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.getCellInterfaceAreas(),
...                         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.getCellCenters()
>>> rad = numerix.sqrt((x - .5)**2 + (y - .5)**2) - r
>>> distanceVariable = DistanceVariable(mesh = mesh, value = rad)
>>> print distanceVariable.getCellInterfaceAreas().sum()
1.57984690073
```

The levelSetDiffusionEquation Module

The levelSetDiffusionVariable Module

17.1.3 electroChem Package Documentation

This page contains the electroChem Package documentation.

The gapFillMesh Module

The *GapFillMesh* object 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 *GmshImporter2D* 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.

```
class GapFillMesh (cellSize=None, desiredDomainWidth=None, desiredDomainHeight=None, desiredFineRegionHeight=None, transitionRegionHeight=None)
Bases: fipy.meshes.numMesh.mesh2D.Mesh2D
```

The following test case tests for diffusion across the domain.

```
>>> domainHeight = 5.
>>> mesh = GapFillMesh(transitionRegionHeight = 2.,
...                     cellSize = 0.1,
...                     desiredFineRegionHeight = 1.,
...                     desiredDomainHeight = domainHeight,
...                     desiredDomainWidth = 1.)

>>> import fipy.tools.dump as dump
>>> (f, filename) = dump.write(mesh)
>>> mesh = dump.read(filename, f)
>>> mesh.getNumberOfCells() - len(mesh.getCellIDsAboveFineRegion())
90

>>> from fipy.variables.cellVariable import CellVariable
>>> var = CellVariable(mesh = mesh)

>>> from fipy.terms.diffusionTerm import DiffusionTerm
>>> eq = DiffusionTerm()

>>> from fipy.boundaryConditions.fixedValue import FixedValue
>>> eq.solve(var, boundaryConditions = (FixedValue(mesh.getFacesBottom(), 0.),
...                                         FixedValue(mesh.getFacesTop(), domainHeight)))
```

Evaluate the result:

```
>>> centers = mesh.getCellCenters()[1].copy() ## the copy makes the array contiguous for inlining
>>> localErrors = (centers - var)**2 / centers**2
>>> globalError = numerix.sqrt(numerix.sum(localErrors) / mesh.getNumberOfCells())
>>> argmax = numerix.argmax(localErrors)
>>> print numerix.sqrt(localErrors[argmax]) < 0.1
1
>>> print globalError < 0.05
1
```

Arguments:

cellSize - The cell size in the fine grid around the trench.

desiredDomainWidth - The desired domain width.

desiredDomainHeight - The total desired height of the domain.

desiredFineRegionHeight - The desired height of the in the fine region around the trench.

transitionRegionHeight - The height of the transition region.

```
buildTransitionMesh(nx, height, cellSize)
getCellIDsAboveFineRegion()
getFineMesh()

class TrenchMesh(trenchDepth=None, trenchSpacing=None, boundaryLayerDepth=None, cellSize=None, aspectRatio=None, angle=0.0, bowWidth=0.0, overBumpRadius=0.0, overBumpWidth=0.0)
Bases: fipy.models.levelSet.electroChem.gapFillMesh.GapFillMesh
```

The trench mesh takes the parameters generally used to define a trench region and recasts them for the general *GapFillMesh*.

The following test case tests for diffusion across the domain.

```
>>> cellSize = 0.05e-6
>>> trenchDepth = 0.5e-6
>>> boundaryLayerDepth = 50e-6
>>> domainHeight = 10 * cellSize + trenchDepth + boundaryLayerDepth

>>> mesh = TrenchMesh(trenchSpacing = 1e-6,
...                      cellSize = cellSize,
...                      trenchDepth = trenchDepth,
...                      boundaryLayerDepth = boundaryLayerDepth,
...                      aspectRatio = 1.)

>>> import fipy.tools.dump as dump
>>> (f, filename) = dump.write(mesh)
>>> mesh = dump.read(filename, f)
>>> mesh.getNumberOfCells() - len(numerix.nonzero(mesh.getElectrolyteMask())[0])
150

>>> from fipy.variables.cellVariable import CellVariable
>>> var = CellVariable(mesh = mesh, value = 0.)

>>> from fipy.terms.diffusionTerm import DiffusionTerm
>>> eq = DiffusionTerm()

>>> from fipy.boundaryConditions.fixedValue import FixedValue

>>> eq.solve(var, boundaryConditions = (FixedValue(mesh.getFacesBottom(), 0.),
...                                         FixedValue(mesh.getFacesTop(), domainHeight)))
```

Evaluate the result:

```
>>> centers = mesh.getCellCenters()[1].copy() ## ensure contiguous array for inlining
>>> localErrors = (centers - var)**2 / centers**2
>>> globalError = numerix.sqrt(numerix.sum(localErrors) / mesh.getNumberOfCells())
>>> argmax = numerix.argmax(localErrors)
>>> print numerix.sqrt(localErrors[argmax]) < 0.051
1
>>> print globalError < 0.02
1
```

trenchDepth - Depth of the trench.
trenchSpacing - The distance between the trenches.
boundaryLayerDepth - The depth of the hydrodynamic boundary layer.
cellSize - The cell Size.
aspectRatio - *trenchDepth* / *trenchWidth*
angle - The angle for the taper of the trench.
bowWidth - The maximum displacement for any bow in the trench shape.
overBumpWidth - The width of the over bump.
overBumpRadius - The radius of the over bump.

getBottomFaces ()
Included to not break the interface

getElectrolyteMask ()

getTopFaces ()
Included to not break the interface

The `metalIonDiffusionEquation` Module

`buildMetalIonDiffusionEquation (ionVar=None, distanceVar=None, depositionRate=1, transientCo-eff=1, diffusionCoeff=1, metalIonMolarVolume=1)`

The *MetalIonDiffusionEquation* solves the diffusion of the metal species with a source term at the electrolyte interface. The governing equation is given by,

$$\frac{\partial c}{\partial t} = \nabla \cdot D \nabla c$$

where,

$$D = \begin{cases} D_c & \text{when } \phi > 0 \\ 0 & \text{when } \phi \leq 0 \end{cases}$$

The velocity of the interface generally has a linear dependence on ion concentration. The following boundary condition applies at the zero level set,

$$D \hat{n} \cdot \nabla c = \frac{v(c)}{\Omega} \quad \text{at } phi = 0$$

where

$$v(c) = c V_0$$

The test case below is for a 1D steady state problem. The solution is given by:

$$c(x) = \frac{c^\infty}{\Omega D/V_0 + L} (x - L) + c^\infty$$

This is the test case,

```
>>> from fipy.meshes.grid1D import Grid1D
>>> nx = 11
>>> dx = 1.
>>> from fipy.tools import serial
```

```

>>> mesh = Grid1D(nx = nx, dx = dx, parallelModule=serial)
>>> x, = mesh.getCellCenters()
>>> from fipy.variables.cellVariable import CellVariable
>>> ionVar = CellVariable(mesh = mesh, value = 1.)
>>> from fipy.models.levelSet.distanceFunction.distanceVariable \
...     import DistanceVariable
>>> disVar = DistanceVariable(mesh = mesh,
...                             value = (x - 0.5) - 0.99,
...                             hasOld = 1)

>>> v = 1.
>>> diffusion = 1.
>>> omega = 1.
>>> cinf = 1.
>>> from fipy.boundaryConditions.fixedValue import FixedValue
>>> eqn = buildMetalIonDiffusionEquation(ionVar = ionVar,
...                                         distanceVar = disVar,
...                                         depositionRate = v * ionVar,
...                                         diffusionCoeff = diffusion,
...                                         metalIonMolarVolume = omega)
>>> bc = (FixedValue(mesh.getFacesRight(), cinf),)
>>> for i in range(10):
...     eqn.solve(ionVar, dt = 1000, boundaryConditions = bc)
>>> L = (nx - 1) * dx - dx / 2
>>> gradient = cinf / (omega * diffusion / v + L)
>>> answer = gradient * (x - L - dx * 3 / 2) + cinf
>>> answer[x < dx] = 1
>>> print ionVar.allclose(answer)
1

```

Parameters

- *ionVar*: The metal ion concentration variable.
- *distanceVar*: A *DistanceVariable* object.
- *depositionRate*: A float or a *CellVariable* representing the interface deposition rate.
- *transientCoeff*: The transient coefficient.
- *diffusionCoeff*: The diffusion coefficient
- *metalIonMolarVolume*: Molar volume of the metal ions.

The `metalIonSourceVariable` Module

The `test` Module

17.1.4 surfactant Package Documentation

This page contains the surfactant Package documentation.

The adsorbingSurfactantEquation Module

```
class AdsorbingSurfactantEquation(surfactantVar=None, distanceVar=None, bulkVar=None, rateConstant=None, otherVar=None, otherBulkVar=None, otherRateConstant=None, consumptionCoeff=None)
```

Bases: `fipy.models.levelSet.surfactant.surfactantEquation.SurfactantEquation`

The `AdsorbingSurfactantEquation` object solves the `SurfactantEquation` but with an adsorbing species from some bulk value. The equation that describes the surfactant adsorbing is given by,

$$\dot{\theta} = Jv\theta + kc(1 - \theta - \theta_{\text{other}}) - \theta c_{\text{other}} k_{\text{other}} - k^- \theta$$

where θ , J , v , k , c , k^- and n represent the surfactant coverage, the curvature, the interface normal velocity, the adsorption rate, the concentration in the bulk at the interface, the consumption rate and an exponent of consumption, respectively. The other subscript refers to another surfactant with greater surface affinity.

The terms on the RHS of the above equation represent conservation of surfactant on a non-uniform surface, Langmuir adsorption, removal of surfactant due to adsorption of the other surfactant onto non-vacant sites and consumption of the surfactant respectively. The adsorption term is added to the source by setting :math:`S_c = k c (1 - \theta_{\text{other}})^n` and $S_p = -kc$. The other terms are added to the source in a similar way.

The following is a test case:

```
>>> from fipy.models.levelSet.distanceFunction.distanceVariable \
...     import DistanceVariable
>>> from fipy.models.levelSet.surfactant.surfactantVariable \
...     import SurfactantVariable
>>> from fipy.meshes.grid2D import Grid2D
>>> dx = .5
>>> dy = 2.3
>>> dt = 0.25
>>> k = 0.56
>>> initialValue = 0.1
>>> c = 0.2

>>> from fipy.meshes.grid2D import Grid2D
>>> mesh = Grid2D(dx = dx, dy = dy, nx = 5, ny = 1)
>>> distanceVar = DistanceVariable(mesh = mesh,
...                                     value = (-dx*3/2, -dx/2, dx/2,
...                                               3*dx/2, 5*dx/2),
...                                     hasOld = 1)
>>> surfactantVar = SurfactantVariable(value = (0, 0, initialValue, 0, 0),
...                                         distanceVar = distanceVar)
>>> bulkVar = CellVariable(mesh = mesh, value = (c, c, c, c, c))
>>> eqn = AdsorbingSurfactantEquation(surfactantVar = surfactantVar,
...                                       distanceVar = distanceVar,
...                                       bulkVar = bulkVar,
...                                       rateConstant = k)
>>> eqn.solve(surfactantVar, dt = dt)
>>> answer = (initialValue + dt * k * c) / (1 + dt * k * c)
>>> print numerix.allclose(surfactantVar.getInterfaceVar(),
...                           numerix.array((0, 0, answer, 0, 0)))
1
```

The following test case is for two surfactant variables. One has more surface affinity than the other.

```
>>> from fipy.models.levelSet.distanceFunction.distanceVariable \
...     import DistanceVariable
```

```

>>> from fipy.models.levelSet.surfactant.surfactantVariable \
...     import SurfactantVariable
>>> from fipy.meshes.grid2D import Grid2D
>>> dx = 0.5
>>> dy = 2.73
>>> dt = 0.001
>>> k0 = 1.
>>> k1 = 10.
>>> theta0 = 0.
>>> thetal = 0.
>>> c0 = 1.
>>> c1 = 1.
>>> totalSteps = 100
>>> mesh = Grid2D(dx = dx, dy = dy, nx = 5, ny = 1)
>>> distanceVar = DistanceVariable(mesh = mesh,
...                                     value = dx * (numerix.arange(5) - 1.5),
...                                     hasOld = 1)
>>> var0 = SurfactantVariable(value = (0, 0, theta0, 0, 0),
...                             distanceVar = distanceVar)
>>> var1 = SurfactantVariable(value = (0, 0, thetal, 0, 0),
...                             distanceVar = distanceVar)
>>> bulkVar0 = CellVariable(mesh = mesh, value = (c0, c0, c0, c0, c0))
>>> bulkVar1 = CellVariable(mesh = mesh, value = (c1, c1, c1, c1, c1))

>>> eqn0 = AdsorbingSurfactantEquation(surfactantVar = var0,
...                                         distanceVar = distanceVar,
...                                         bulkVar = bulkVar0,
...                                         rateConstant = k0)

>>> eqn1 = AdsorbingSurfactantEquation(surfactantVar = var1,
...                                         distanceVar = distanceVar,
...                                         bulkVar = bulkVar1,
...                                         rateConstant = k1,
...                                         otherVar = var0,
...                                         otherBulkVar = bulkVar0,
...                                         otherRateConstant = k0)

>>> for step in range(totalSteps):
...     eqn0.solve(var0, dt = dt)
...     eqn1.solve(var1, dt = dt)
>>> answer0 = 1 - numerix.exp(-k0 * c0 * dt * totalSteps)
>>> answer1 = (1 - numerix.exp(-k1 * c1 * dt * totalSteps)) * (1 - answer0)
>>> print numerix.allclose(var0.getInterfaceVar(),
...                           numerix.array((0, 0, answer0, 0, 0)), rtol = 1e-2)
1
>>> print numerix.allclose(var1.getInterfaceVar(),
...                           numerix.array((0, 0, answer1, 0, 0)), rtol = 1e-2)
1
>>> dt = 0.1
>>> for step in range(10):
...     eqn0.solve(var0, dt = dt)
...     eqn1.solve(var1, dt = dt)

>>> x, y = mesh.getCellCenters()
>>> check = var0.getInterfaceVar() + var1.getInterfaceVar()

```

```
>>> answer = CellVariable(mesh=mesh, value=check)
>>> answer[x==1.25] = 1.
>>> print check.allegual(answer)
True

The following test case is to fix a bug where setting the adsorption coefficient to zero leads to the solver not converging and an eventual failure.

>>> var0 = SurfactantVariable(value = (0, 0, theta0, 0 ,0),
...                               distanceVar = distanceVar)
>>> bulkVar0 = CellVariable(mesh = mesh, value = (c0, c0, c0, c0, c0))

>>> eqn0 = AdsorbingSurfactantEquation(surfactantVar = var0,
...                                       distanceVar = distanceVar,
...                                       bulkVar = bulkVar0,
...                                       rateConstant = 0)

>>> eqn0.solve(var0, dt = dt)
>>> eqn0.solve(var0, dt = dt)
>>> answer = CellVariable(mesh=mesh, value=var0.getInterfaceVar())
>>> answer[x==1.25] = 0.
>>> print var0.getInterfaceVar().allclose(answer)
True

The following test case is to fix a bug that allows the accelerator to become negative.

>>> nx = 5
>>> ny = 5
>>> mesh = Grid2D(dx = 1., dy = 1., nx = nx, ny = ny)
>>> x, y = mesh.getCellCenters()
>>> disVar = DistanceVariable(mesh=mesh, value=1., hasOld=True)
>>> disVar[y < dy] = -1
>>> disVar[x < dx] = -1
>>> disVar.calcDistanceFunction()

>>> levVar = SurfactantVariable(value = 0.5, distanceVar = disVar)
>>> accVar = SurfactantVariable(value = 0.5, distanceVar = disVar)

>>> levEq = AdsorbingSurfactantEquation(levVar,
...                                         distanceVar = disVar,
...                                         bulkVar = 0,
...                                         rateConstant = 0)

>>> accEq = AdsorbingSurfactantEquation(accVar,
...                                         distanceVar = disVar,
...                                         bulkVar = 0,
...                                         rateConstant = 0,
...                                         otherVar = levVar,
...                                         otherBulkVar = 0,
...                                         otherRateConstant = 0)

>>> extVar = CellVariable(mesh = mesh, value = accVar.getInterfaceVar())
```

```
>>> from fipy.models.levelSet.advection.higherOrderAdvectionEquation \
...      import buildHigherOrderAdvectionEquation
>>> advEq = buildHigherOrderAdvectionEquation(advectionCoeff = extVar)

>>> dt = 0.1

>>> for i in range(50):
...     disVar.calcDistanceFunction()
...     extVar.setValue(numerix.array(accVar.getInterfaceVar()))
...     disVar.extendVariable(extVar)
...     disVar.updateOld()
...     advEq.solve(disVar, dt = dt)
...     levEq.solve(levVar, dt = dt)
...     accEq.solve(accVar, dt = dt)

>>> print (accVar >= -1e-10).all()
True
```

Create a *AdsorbingSurfactantEquation* object.

Parameters

- *surfactantVar*: The *SurfactantVariable* to be solved for.
- *distanceVar*: The *DistanceVariable* that marks the interface.
- *bulkVar*: The value of the *surfactantVar* in the bulk.
- *rateConstant*: The adsorption rate of the *surfactantVar*.
- *otherVar*: Another *SurfactantVariable* with more surface affinity.
- *otherBulkVar*: The value of the *otherVar* in the bulk.
- *otherRateConstant*: The adsorption rate of the *otherVar*.
- *consumptionCoeff*: The rate that the *surfactantVar* is consumed during deposition.

solve (*var*, *boundaryConditions*=(), *solver*=*LinearPCGSolver(tolerance=1e-10, iterations=1000)*, *dt*=1.0)
Builds and solves the *AdsorbingSurfactantEquation*'s linear system once.

Parameters

- *var*: A *SurfactantVariable* to be solved for. Provides the initial condition, the old value and holds the solution on completion.
- *solver*: The iterative solver to be used to solve the linear system of equations.
- *boundaryConditions*: A tuple of *boundaryConditions*.
- *dt*: The time step size.

sweep (*var*, *solver*=*LinearLUSolver(tolerance=1e-10, iterations=10)*, *boundaryConditions*=(), *dt*=1.0, *underRelaxation*=None, *residualFn*=None)
Builds and solves the *AdsorbingSurfactantEquation*'s linear system once. This method also recalculates and returns the residual as well as applying under-relaxation.

Parameters

- *var*: The variable to be solved for. Provides the initial condition, the old value and holds the solution on completion.

- *solver*: The iterative solver to be used to solve the linear system of equations.
- *boundaryConditions*: A tuple of boundaryConditions.
- *dt*: The time step size.
- *underRelaxation*: Usually a value between 0 and 1 or *None* in the case of no under-relaxation

The convectionCoeff Module

The lines Module

The matplotlibSurfactantViewer Module

```
class MatplotlibSurfactantViewer(distanceVar, surfactantVar=None, levelSetValue=0.0, title=None, smooth=0, zoomFactor=1.0, animate=False, limits={}, **kwlimits)
Bases: fipy.viewers.matplotlibViewer.matplotlibViewer._MatplotlibViewer
```

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.getCellCenters()
>>> v = CellVariable(mesh=m, value=x**2 + y**2 - 10**2)
>>> s = CellVariable(mesh=m, value=sin(x / 10) * cos(y / 30))
>>> viewer = MatplotlibSurfactantViewer(distanceVar=v, surfactantVar=s)
>>> for r in range(1,200):
...     v.setValue(x**2 + y**2 - r**2)
...     viewer.plot()

>>> from fipy import *
>>> dx = 1.
>>> dy = 1.
>>> nx = 11
>>> ny = 11
>>> Lx = ny * dy
>>> Ly = nx * dx
>>> mesh = Grid2D(dx = dx, dy = dy, nx = nx, ny = ny)
>>> # from fipy.models.levelSet.distanceFunction import DistanceVariable
>>> var = DistanceVariable(mesh = mesh, value = -1)

>>> x, y = mesh.getCellCenters()

>>> 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 / 3.) & (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*: a *DistanceVariable* object.
- *levelSetValue*: the value of the contour to be displayed
- *title*: displayed at the top of the *Viewer* window
- *animate*: whether to show only the initial condition and the
- *limits*: a dictionary with possible keys *xmin*, *xmax*, *ymin*, *ymax*, *zmin*, *zmax*, *datamin*, *datamax*. A 1D *Viewer* will only use *xmin* and *xmax*, a 2D viewer will also use *ymin* and *ymax*, and so on. All viewers will use *datamin* and *datamax*. Any limit set to a (default) value of *None* will autoscale, moving top boundary or to show all contours (Default)

The `mayaviSurfactantViewer` Module

```
class MayaviSurfactantViewer(distanceVar, surfactantVar=None, levelSetValue=0.0, title=None, smooth=0,
                               zoomFactor=1.0, animate=False, limits={}, **kwlimits)
Bases: fipy.viewers.viewer._Viewer
```

The *MayaviSurfactantViewer* creates a viewer with the `Mayavi` python plotting package that displays a *DistanceVariable*.

Create a *MayaviSurfactantViewer*.

```

>>> from fipy import *
>>> dx = 1.
>>> dy = 1.
>>> nx = 11
>>> ny = 11
>>> Lx = ny * dy
>>> Ly = nx * dx
>>> mesh = Grid2D(dx = dx, dy = dy, nx = nx, ny = ny)
>>> # from fipy.models.levelSet.distanceFunction.distanceVariable import DistanceVariable
>>> var = DistanceVariable(mesh = mesh, value = -1)

>>> x, y = mesh.getCellCenters()

>>> 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 / 3.)) | ((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*: a *DistanceVariable* object.
- *levelSetValue*: the value of the contour to be displayed
- *title*: displayed at the top of the *Viewer* window
- *animate*: whether to show only the initial condition and the
- *limits*: a dictionary with possible keys *xmin*, *xmax*, *ymin*, *ymax*, *zmin*, *zmax*, *datamin*, *datamax*. A 1D *Viewer* will only use *xmin* and *xmax*, a 2D viewer will also use *ymin* and *ymax*, and so on. All viewers will use *datamin* and *datamax*. Any limit set to a (default) value of *None* will autoscale. moving top boundary or to show all contours (Default)

plot (*filename=None*)

The surfactantBulkDiffusionEquation Module

buildSurfactantBulkDiffusionEquation (*bulkVar=None*, *distanceVar=None*, *surfactantVar=None*, *otherSurfactantVar=None*, *diffusionCoeff=None*, *transientCoeff=1.0*, *rateConstant=None*)

The *buildSurfactantBulkDiffusionEquation* function returns a 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,

$$D = \begin{cases} D_c & \text{when } \phi > 0 \\ 0 & \text{when } \phi \leq 0 \end{cases}$$

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 the interface is given by

$$D\hat{n} \cdot \nabla c = -kc(1 - \theta) \quad \text{at } \phi = 0.$$

Parameters

- *bulkVar*: The bulk surfactant concentration variable.
- *distanceVar*: A *DistanceVariable* object
- *surfactantVar*: A *SurfactantVariable* object
- *otherSurfactantVar*: Any other surfactants that may remove this one.
- *diffusionCoeff*: A float or a *FaceVariable*.
- *transientCoeff*: In general 1 is used.
- *rateConstant*: The adsorption coefficient.

The `surfactantEquation` Module

`class SurfactantEquation (distanceVar=None)`

A *SurfactantEquation* aims to evolve a surfactant on an interface defined by the zero level set of the *distanceVar*. The method should completely conserve the total coverage of surfactant. The surfactant is only in the cells immediately in front of the advancing interface. The method only works for a positive velocity as it stands.

Creates a *SurfactantEquation* object.

Parameters

- *distanceVar*: The *DistanceVariable* that marks the interface.

`solve (var, boundaryConditions=(), solver=LinearLUSolver(tolerance=1e-10, iterations=10), dt=1.0)`

Builds and solves the *SurfactantEquation*'s linear system once.

Parameters

- *var*: A *SurfactantVariable* to be solved for. Provides the initial condition, the old value and holds the solution on completion.
- *solver*: The iterative solver to be used to solve the linear system of equations.
- *boundaryConditions*: A tuple of *boundaryConditions*.
- *dt*: The time step size.

`sweep (var, solver=LinearLUSolver(tolerance=1e-10, iterations=10), boundaryConditions=(), dt=1.0, underRelaxation=None, residualFn=None)`

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*: The variable to be solved for. Provides the initial condition, the old value and holds the solution on completion.
- *solver*: The iterative solver to be used to solve the linear system of equations.
- *boundaryConditions*: A tuple of *boundaryConditions*.
- *dt*: The time step size.
- *underRelaxation*: Usually a value between 0 and 1 or *None* in the case of no under-relaxation

The `SurfactantVariable` Module

```
class SurfactantVariable(value=0.0, distanceVar=None, name='surfactant variable', hasOld=False)
    Bases: fipy.variables.cellVariable.CellVariable
```

The `SurfactantVariable` maintains a conserved volumetric concentration on cells adjacent to, but in front of, the interface. The `value` argument corresponds to the initial concentration of surfactant on the interface (moles divided by area). The value held by the `SurfactantVariable` is actually a volume density (moles divided by volume).

A simple 1D test:

```
>>> from fipy.meshes.grid1D import Grid1D
>>> mesh = Grid1D(dx = 1., nx = 4)
>>> from fipy.models.levelSet.distanceFunction.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.grid2D 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`: The initial value.
- `distanceVar`: A `DistanceVariable` object.
- `name`: The name of the variable.

`copy()`

getInterfaceVar()

Returns the *SurfactantVariable* rendered as an *_InterfaceSurfactantVariable* which evaluates the surfactant concentration as an area concentration the interface rather than a volumetric concentration.

17.1.5 The `test` Module

17.2 The `test` Module

solvers Package Documentation

This page contains the solvers Package documentation.

18.1 pysparse Package Documentation

This page contains the pysparse Package documentation.

18.1.1 The linearCGSSolver Module

```
class LinearCGSSolver (*args, **kwargs)
Bases: fipy.solvers.pysparse.pysparseSolver.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.

18.1.2 The linearGMRESSolver Module

```
class LinearGMRESSolver (*args, **kwargs)
Bases: fipy.solvers.pysparse.pysparseSolver.PysparseSolver
```

The *LinearGMRESSolver* solves a linear system of equations using the generalised 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.

18.1.3 The linearJORSolver Module

```
class LinearJORSolver (tolerance=1e-10, iterations=1000, steps=None, relaxation=1.0, precon=None)
Bases: fipy.solvers.pysparse.pysparseSolver.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*: The required error tolerance.

- *iterations*: The maximum number of iterative steps to perform.
- *steps*: A deprecated name for *iterations*.
- *relaxation*: The relaxation.

18.1.4 The `linearLUSolver` Module

```
class LinearLUSolver (tolerance=1e-10, iterations=10, steps=None, precon=None, maxIterations=10)
Bases: fipy.solvers.pysparse.pysparseSolver.PysparseSolver
```

The *LinearLUSolver* solves a linear system of equations using LU-factorisation. 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*: The required error tolerance.
- *iterations*: The number of LU decompositions to perform.
- *steps*: A deprecated name for *iterations*. For large systems a number of iterations is generally required.

18.1.5 The `linearPCGSolver` Module

```
class LinearPCGSolver (*args, **kwargs)
Bases: fipy.solvers.pysparse.pysparseSolver.PysparseSolver
```

The *LinearPCGSolver* solves a linear system of equations using the preconditioned conjugate gradient method (PCG) with symmetric successive over-relaxation (SSOR) preconditioning. 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.

18.1.6 The `pysparseSolver` Module

```
class PysparseSolver (*args, **kwargs)
Bases: fipy.solvers.solver.Solver
```

The base *pysparseSolver* class.

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

18.2 trilinos Package Documentation

This page contains the trilinos Package documentation.

18.2.1 preconditioners Package Documentation

This page contains the preconditioners Package documentation.

The `domDecompPreconditioner` Module

```
class DomDecompPreconditioner()
    Bases: fipy.solvers.trilinos.preconditioners.preconditioner.Preconditioner
    Domain Decomposition preconditioner for Trilinos solvers.
    Create a Preconditioner object.
```

The `icPreconditioner` Module

```
class ICPreconditioner()
    Bases: fipy.solvers.trilinos.preconditioners.preconditioner.Preconditioner
    Incomplete Cholesky Preconditioner from IFPACK for Trilinos Solvers.
    Create a Preconditioner object.
```

The `jacobiPreconditioner` Module

```
class JacobiPreconditioner()
    Bases: fipy.solvers.trilinos.preconditioners.preconditioner.Preconditioner
    Jacobi Preconditioner for Trilinos solvers.
    Create a Preconditioner object.
```

The `multilevelDDMLPreconditioner` Module

```
class MultilevelDDMLPreconditioner()
    Bases: fipy.solvers.trilinos.preconditioners.preconditioner.Preconditioner
    Multilevel preconditioner for Trilinos solvers. 3-level algebraic domain decomposition.
    Create a Preconditioner object.
```

The `multilevelDDPPreconditioner` Module

```
class MultilevelDDPPreconditioner()
    Bases: fipy.solvers.trilinos.preconditioners.preconditioner.Preconditioner
    Multilevel preconditioner for Trilinos solvers. A classical smoothed aggregation-based 2-level domain decomposition.
    Create a Preconditioner object.
```

The `multilevelNSSAPreconditioner` Module

```
class MultilevelNSSAPreconditioner()
    Bases: fipy.solvers.trilinos.preconditioners.preconditioner.Preconditioner
    Energy-based minimizing smoothed aggregation suitable for highly convective non-symmetric fluid flow problems.
    Create a Preconditioner object.
```

The `multilevelSAPreconditioner` Module

```
class MultilevelSAPreconditioner()
    Bases: fipy.solvers.trilinos.preconditioners.preconditioner.Preconditioner

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

Create a *Preconditioner* object.

The `preconditioner` Module

```
class Preconditioner()
    The base Preconditioner class.
```

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

Create a *Preconditioner* object.

18.2.2 The `linearBicgstabSolver` Module

```
class LinearBicgstabSolver(tolerance=1e-10,           iterations=1000,           steps=None,           pre-
                                con=<fipy.solvers.trilinos.preconditioners.jacobiPreconditioner.JacobiPreconditioner
                                instance at 0x43d1e40>)
    Bases: fipy.solvers.trilinos.trilinosAztecOOSolver.TrilinosAztecOOSolver
```

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

Parameters

- *tolerance*: The required error tolerance.
- *iterations*: The maximum number of iterative steps to perform.
- *steps*: A deprecated name for *iterations*.
- *precon*: Preconditioner to use.

18.2.3 The `linearCGSSolver` Module

```
class LinearCGSSolver(tolerance=1e-10,           iterations=1000,           steps=None,           pre-
                                con=<fipy.solvers.trilinos.preconditioners.multilevelDDPreconditioner.MultilevelDDPreconditioner
                                instance at 0x43d2d78>)
    Bases: fipy.solvers.trilinos.trilinosAztecOOSolver.TrilinosAztecOOSolver
```

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

Parameters

- *tolerance*: The required error tolerance.
- *iterations*: The maximum number of iterative steps to perform.
- *steps*: A deprecated name for *iterations*.
- *precon*: Preconditioner to use.

18.2.4 The `linearGMRESSolver` Module

```
class LinearGMRESSolver (tolerance=1e-10,           iterations=1000,           steps=None,           pre-
                           con=<fipy.solvers.trilinos.preconditioners.multilevelDDPreconditioner.MultilevelDDPreconditioner
                           instance at 0x43d2648>)
Bases: fipy.solvers.trilinos.trilinosAztecOOSolver.TrilinosAztecOOSolver
```

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

Parameters

- *tolerance*: The required error tolerance.
- *iterations*: The maximum number of iterative steps to perform.
- *steps*: A deprecated name for *iterations*.
- *precon*: Preconditioner to use.

18.2.5 The `linearLUSolver` Module

```
class LinearLUSolver (tolerance=1e-10, iterations=10, steps=None, precon=None, maxIterations=10)
Bases: fipy.solvers.trilinos.trilinosSolver.TrilinosSolver
```

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

Parameters

- *tolerance*: The required error tolerance.
- *iterations*: The maximum number of iterative steps to perform.
- *steps*: A deprecated name for *iterations*.

18.2.6 The `linearPCGSolver` Module

```
class LinearPCGSolver (tolerance=1e-10,           iterations=1000,           steps=None,           pre-
                           con=<fipy.solvers.trilinos.preconditioners.multilevelDDPreconditioner.MultilevelDDPreconditioner
                           instance at 0x43d2be8>)
Bases: fipy.solvers.trilinos.trilinosAztecOOSolver.TrilinosAztecOOSolver
```

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

Parameters

- *tolerance*: The required error tolerance.
- *iterations*: The maximum number of iterative steps to perform.
- *steps*: A deprecated name for *iterations*.
- *precon*: Preconditioner to use.

18.2.7 The `trilinosAztecOOSolver` Module

```
class TrilinosAztecOOSolver (tolerance=1e-10,           iterations=1000,           steps=None,           pre-
                           con=<fipy.solvers.trilinos.preconditioners.jacobiPreconditioner.JacobiPreconditioner
                           instance at 0x43d2dc8>)
Bases: fipy.solvers.trilinos.trilinosSolver.TrilinosSolver
```

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

Parameters

- *tolerance*: The required error tolerance.
- *iterations*: The maximum number of iterative steps to perform.
- *steps*: A deprecated name for *iterations*.
- *precon*: Preconditioner object to use.

18.2.8 The `trilinosMLTest` Module

```
class TrilinosMLTest (tolerance=1e-10, iterations=5, steps=None, MLOptions={}, testUnsupported=False)
Bases: fipy.solvers.trilinos.trilinosSolver.TrilinosSolver
```

This solver class does not actually solve the system, but outputs information about what ML preconditioner settings will work best.

Parameters

- *tolerance*: The required error tolerance.
- *iterations*: The maximum number of iterations to perform per test.
- *steps*: A deprecated name for *iterations*.
- *MLOptions*: Options to pass to ML. A dictionary of {option:value} pairs. This will be passed to ML.SetParameterList.
- *testUnsupported*: test smoothers that are not currently implemented in preconditioner objects.

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.

18.2.9 The `trilinosSolver` Module

```
class TrilinosSolver (*args, **kwargs)
Bases: fipy.solvers.solver.Solver
```

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

18.3 The `solver` Module

The iterative solvers may output warnings if the solution is considered unsatisfactory. If you are not interested in these warnings, you can invoke python with a warning filter such as:

```
$ python -W ignore::fipy.SolverConvergenceWarning myscript.py
```

If you are extremely concerned about your preconditioner for some reason, you can abort whenever it has problems with:

```
$ python -Werror::fipy.PreconditionerWarning myscript.py

exception IllConditionedPreconditionerWarning
    Bases: fipy.solvers.solver.PreconditionerWarning

exception MatrixIllConditionedWarning
    Bases: fipy.solvers.solver.SolverConvergenceWarning

exception MaximumIterationWarning
    Bases: fipy.solvers.solver.SolverConvergenceWarning

exception PreconditionerNotPositiveDefiniteWarning
    Bases: fipy.solvers.solver.PreconditionerWarning

exception PreconditionerWarning
    Bases: fipy.solvers.solver.SolverConvergenceWarning

exception ScalarQuantityOutOfRangeWarning
    Bases: fipy.solvers.solver.SolverConvergenceWarning

class Solver (tolerance=1e-10, iterations=1000, steps=None, precon=None)
    The base LinearXSolver class.
```

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

Create a *Solver* object.

Parameters

- *tolerance*: The required error tolerance.
- *iterations*: The maximum number of iterative steps to perform.
- *steps*: A deprecated name for *iterations*.
- *precon*: Preconditioner to use. This parameter is only available for Trilinos solvers.

```
exception SolverConvergenceWarning
    Bases: exceptions.Warning

exception StagnatedSolverWarning
    Bases: fipy.solvers.solver.SolverConvergenceWarning
```

18.4 The test Module

steppers Package Documentation

This page contains the steppers Package documentation.

19.1 The `steppers` Package

L1error (*var, matrix, RHSvector*)

Parameters

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

Returns

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

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

L2error (*var, matrix, RHSvector*)

Parameters

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

Returns

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

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

LINFerror (*var, matrix, RHSvector*)

Parameters

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

Returns

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

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

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

Parameters

- *var*: The *CellVariable* in question.
- *matrix*: (*ignored*)
- *RHSvector*: (*ignored*)
- *norm*: A function that will normalize its *array* argument and return a single number

Returns

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

where $\|\vec{x}\|_{?}$ is the normalization of \vec{x} provided by `norm()`.

residual (*var, matrix, RHSvector*)

Determines the residual for the current solution matrix and variable.

Parameters

- *var*: The *CellVariable* in question, *prior* to solution.
- *matrix*: The coefficient matrix at this step/sweep
- *RHSvector*: The

Returns

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

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

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

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

Parameters

- *fn*: The function to call
- *args*: The unnamed function argument *list*
- *kwargs*: The named function argument *dict*

Returns the final residual

19.2 The pidStepper Module

```
class PIDStepper (vardata=(), proportional=0.07499999999999997, integral=0.1749999999999999, derivative=0.01)
```

Bases: `fipy.steppers stepper.Stepper`

Adaptive stepper using a PID controller, based on:

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

19.3 The `pseudoRKQSStepper` Module

```
class PseudoRKQSStepper (vardata=(), safety=0.9000000000000002, pgrow=-0.2000000000000001,
                           pshrink=-0.25, errcon=0.00018900000000000001)
Bases: fipy.steppers.stepper.Stepper
```

Adaptive stepper based on the rkqs (Runge-Kutta “quality-controlled” stepper) algorithm of numerixal 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.

19.4 The `stepper` Module

```
class Stepper (vardata=())

static failFn (vardata, dt, *args, **kwargs)
step (dt, dtTry=None, dtMin=None, dtPrev=None, sweepFn=None, successFn=None, failFn=None, *args,
       **kwargs)
static successFn (vardata, dt, dtPrev, elapsed, *args, **kwargs)
static sweepFn (vardata, dt, *args, **kwargs)
```


Chapter 20

terms Package Documentation

This page contains the terms Package documentation.

20.1 The `cellTerm` Module

```
class CellTerm(coeff=1.0)
Bases: fipy.terms.term.Term
```

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

20.2 The `centralDiffConvectionTerm` Module

```
class CentralDifferenceConvectionTerm(coeff=1.0, diffusionTerm=None)
Bases: fipy.terms.convectionTerm.ConvectionTerm
```

This `Term` represents

$$\int_V \nabla \cdot (\vec{u}\phi) dV \simeq \sum_f (\vec{n} \cdot \vec{u})_f \phi_f A_f$$

where $\phi_f = \alpha_f \phi_P + (1 - \alpha_f) \phi_A$ and α_f is calculated using the central differencing scheme. For further details see *Numerical Schemes*.

Create a `ConvectionTerm` object.

```
>>> from fipy.meshes.grid1D import Grid1D
>>> from fipy.variables.cellVariable import CellVariable
>>> from fipy.variables.faceVariable import FaceVariable
>>> 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):
...
TypeError: The coefficient must be a vector value.
>>> __ConvectionTerm(coeff = fv)
Traceback (most recent call last):
...
TypeError: The coefficient must be a vector value.
```

```
>>> __ConvectionTerm(coeff = vcv)
__ConvectionTerm(coeff=_ArithmeticCellToFaceVariable(value=array([[ 0.,  0.,  0.]]), mesh=UniformGrid1D(dx=1.0, ny=1))
>>> __ConvectionTerm(coeff = vfv)
__ConvectionTerm(coeff=FaceVariable(value=array([[ 0.,  0.,  0.]]), mesh=UniformGrid1D(dx=1.0, ny=1))
>>> __ConvectionTerm(coeff = (1,))
__ConvectionTerm(coef=(1,))
>>> from fipy.terms.explicitUpwindConvectionTerm import ExplicitUpwindConvectionTerm
>>> ExplicitUpwindConvectionTerm(coeff = (0,)).solve(var = cv)
>>> ExplicitUpwindConvectionTerm(coeff = 1).solve(var = cv)
Traceback (most recent call last):
...
TypeError: The coefficient must be a vector value.
>>> from fipy.meshes.grid2D import Grid2D
>>> m2 = Grid2D(nx=2, ny=1)
>>> cv2 = CellVariable(mesh=m2)
>>> vcv2 = CellVariable(mesh=m2, rank=1)
>>> vfv2 = FaceVariable(mesh=m2, rank=1)
>>> __ConvectionTerm(coef=vcv2)
__ConvectionTerm(coeff=_ArithmeticCellToFaceVariable(value=array([[ 0.,  0.,  0.,  0.,  0.,  0.,
[ 0.,  0.,  0.,  0.,  0.,  0.]]), mesh=UniformGrid2D(dx=1.0, dy=1.0, nx=2, ny=1)))
>>> __ConvectionTerm(coef=vfv2)
__ConvectionTerm(coeff=FaceVariable(value=array([[ 0.,  0.,  0.,  0.,  0.,  0.,
[ 0.,  0.,  0.,  0.,  0.,  0.]]), mesh=UniformGrid2D(dx=1.0, dy=1.0, nx=2, ny=1)))
>>> ExplicitUpwindConvectionTerm(coef = ((0),(0))).solve(var=cv2)
>>> ExplicitUpwindConvectionTerm(coef = (0,0)).solve(var=cv2)
```

Parameters

- *coeff* : The *Term*'s coefficient value.
- *diffusionTerm* : **deprecated**. The Peclet number is calculated automatically.

20.3 The collectedDiffusionTerm Module

20.4 The convectionTerm Module

```
class ConvectionTerm(coeff=1.0, diffusionTerm=None)
Bases: fipy.terms.faceTerm.FaceTerm
```

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

Create a *ConvectionTerm* object.

```
>>> from fipy.meshes.grid1D import Grid1D
>>> from fipy.variables.cellVariable import CellVariable
>>> from fipy.variables.faceVariable import FaceVariable
>>> 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(coef = cv)
Traceback (most recent call last):
...
...
```

```

TypeError: The coefficient must be a vector value.
>>> __ConvectionTerm(coeff = fv)
Traceback (most recent call last):
...
TypeError: The coefficient must be a vector value.
>>> __ConvectionTerm(coeff = vcv)
__ConvectionTerm(coeff=_ArithmeticCellToFaceVariable(value=array([[ 0.,  0.,  0.]]), mesh=UniformGrid1D(dx=1.0, ny=1))
>>> __ConvectionTerm(coeff = vfv)
__ConvectionTerm(coeff=FaceVariable(value=array([[ 0.,  0.,  0.])), mesh=UniformGrid1D(dx=1.0, ny=1))
>>> __ConvectionTerm(coeff = (1,))
__ConvectionTerm(coeff=(1,))
>>> from fipy.terms.explicitUpwindConvectionTerm import ExplicitUpwindConvectionTerm
>>> ExplicitUpwindConvectionTerm(coeff = (0,)).solve(var = cv)
>>> ExplicitUpwindConvectionTerm(coeff = 1).solve(var = cv)
Traceback (most recent call last):
...
TypeError: The coefficient must be a vector value.
>>> from fipy.meshes.grid2D import Grid2D
>>> 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, dy=1.0, nx=2, 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, dy=1.0, nx=2, ny=1)))
>>> ExplicitUpwindConvectionTerm(coeff = ((0,), (0,))).solve(var=cv2)
>>> ExplicitUpwindConvectionTerm(coeff = (0,0)).solve(var=cv2)

```

Parameters

- *coeff* : The *Term*'s coefficient value.
- *diffusionTerm* : **deprecated**. The Peclet number is calculated automatically.

20.5 The diffusionTerm Module

```

class DiffusionTerm(coeff=(1.0, ))
Bases: fipy.terms.term.Term

```

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 *DiffusionTerm*.

Parameters

- *coeff*: *Tuple* or *list* of *FaceVariables* or numbers.

```
class DiffusionTermNoCorrection(coeff=(1.0, ))  
Bases: fipy.terms.diffusionTerm.DiffusionTerm
```

Create a *DiffusionTerm*.

Parameters

- *coeff*: *Tuple* or *list* of *FaceVariables* or numbers.

20.6 The equation Module

20.7 The explicitDiffusionTerm Module

```
class ExplicitDiffusionTerm(coeff=(1.0, ))  
Bases: fipy.terms.diffusionTerm.DiffusionTerm
```

The discretization for the *ExplicitDiffusionTerm* is given by

$$\int_V \nabla \cdot (\Gamma \nabla \phi) dV \simeq \sum_f \Gamma_f \frac{\phi_A^{\text{old}} - \phi_P^{\text{old}}}{d_{AP}} A_f$$

where ϕ_A^{old} and ϕ_P^{old} are the old values of the variable. The term is added to the RHS vector and makes no contribution to the solution matrix.

Create a *DiffusionTerm*.

Parameters

- *coeff*: *Tuple* or *list* of *FaceVariables* or numbers.

20.8 The explicitSourceTerm Module

20.9 The explicitUpwindConvectionTerm Module

```
class ExplicitUpwindConvectionTerm(coeff=1.0, diffusionTerm=None)  
Bases: fipy.terms.upwindConvectionTerm.UpwindConvectionTerm
```

The discretization for this *Term* is given by

$$\int_V \nabla \cdot (\vec{u} \phi) dV \simeq \sum_f (\vec{n} \cdot \vec{u})_f \phi_f A_f$$

where $\phi_f = \alpha_f \phi_P^{\text{old}} + (1 - \alpha_f) \phi_A^{\text{old}}$ and α_f is calculated using the upwind scheme. For further details see *Numerical Schemes*.

Create a *ConvectionTerm* object.

```

>>> from fipy.meshes.grid1D import Grid1D
>>> from fipy.variables.cellVariable import CellVariable
>>> from fipy.variables.faceVariable import FaceVariable
>>> 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):
...
TypeError: The coefficient must be a vector value.
>>> __ConvectionTerm(coeff = fv)
Traceback (most recent call last):
...
TypeError: The coefficient must be a vector value.
>>> __ConvectionTerm(coeff = vcv)
__ConvectionTerm(coeff=_ArithmeticCellToFaceVariable(value=array([[ 0.,  0.,  0.])), mesh=UniformGrid1D(dx=1.0, ny=1))
>>> __ConvectionTerm(coeff = vfv)
__ConvectionTerm(coeff=FaceVariable(value=array([[ 0.,  0.,  0.])), mesh=UniformGrid1D(dx=1.0, ny=1))
>>> __ConvectionTerm(coeff = (1,))
__ConvectionTerm(coeff=(1,))
>>> from fipy.terms.explicitUpwindConvectionTerm import ExplicitUpwindConvectionTerm
>>> ExplicitUpwindConvectionTerm(coeff = (0,)).solve(var = cv)
>>> ExplicitUpwindConvectionTerm(coeff = 1).solve(var = cv)
Traceback (most recent call last):
...
TypeError: The coefficient must be a vector value.
>>> from fipy.meshes.grid2D import Grid2D
>>> 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, dy=1.0, nx=2, 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, dy=1.0, nx=2, ny=1)))
>>> ExplicitUpwindConvectionTerm(coeff = ((0,), (0,))).solve(var=cv2)
>>> ExplicitUpwindConvectionTerm(coeff = (0,0)).solve(var=cv2)

```

Parameters

- *coeff* : The *Term*'s coefficient value.
- *diffusionTerm* : **deprecated**. The Peclet number is calculated automatically.

20.10 The exponentialConvectionTerm Module

```

class ExponentialConvectionTerm(coeff=1.0, diffusionTerm=None)
Bases: fipy.terms.convectionTerm.ConvectionTerm

```

The discretization for this `Term` is given by

$$\int_V \nabla \cdot (\vec{u}\phi) dV \simeq \sum_f (\vec{n} \cdot \vec{u})_f \phi_f A_f$$

where $\phi_f = \alpha_f \phi_P + (1 - \alpha_f) \phi_A$ and α_f is calculated using the exponential scheme. For further details see [Numerical Schemes](#).

Create a `ConvectionTerm` object.

```
>>> from fipy.meshes.grid1D import Grid1D
>>> from fipy.variables.cellVariable import CellVariable
>>> from fipy.variables.faceVariable import FaceVariable
>>> 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):
...
TypeError: The coefficient must be a vector value.
>>> __ConvectionTerm(coeff = fv)
Traceback (most recent call last):
...
TypeError: The coefficient must be a vector value.
>>> __ConvectionTerm(coeff = vcv)
__ConvectionTerm(coeff=_ArithmeticCellToFaceVariable(value=array([[ 0.,  0.,  0.]]), mesh=UniformGrid1D(dx=1.0, ny=1))
>>> __ConvectionTerm(coeff = vfv)
__ConvectionTerm(coeff=FaceVariable(value=array([[ 0.,  0.,  0.]]), mesh=UniformGrid1D(dx=1.0, ny=1))
>>> __ConvectionTerm(coeff = (1,))
__ConvectionTerm(coeff=(1,))
>>> from fipy.terms.explicitUpwindConvectionTerm import ExplicitUpwindConvectionTerm
>>> ExplicitUpwindConvectionTerm(coeff = (0,)).solve(var = cv)
>>> ExplicitUpwindConvectionTerm(coeff = 1).solve(var = cv)
Traceback (most recent call last):
...
TypeError: The coefficient must be a vector value.
>>> from fipy.meshes.grid2D import Grid2D
>>> 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, dy=1.0, nx=2, ny=1)))
>>> __ConvectionTerm(coeff=vfv2)
__ConvectionTerm(coeff=FaceVariable(value=array([[ 0.,  0.,  0.,  0.,  0.,  0.,  0.,
       [ 0.,  0.,  0.,  0.,  0.,  0.]]), mesh=UniformGrid2D(dx=1.0, dy=1.0, nx=2, ny=1)))
>>> ExplicitUpwindConvectionTerm(coeff = ((0,), (0,))).solve(var=cv2)
>>> ExplicitUpwindConvectionTerm(coeff = (0,0)).solve(var=cv2)
```

Parameters

- `coeff` : The `Term`'s coefficient value.
- `diffusionTerm` : **deprecated**. The Peclet number is calculated automatically.

20.11 The faceTerm Module

```
class FaceTerm(coeff=1.0)
Bases: fipy.terms.Term
```

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

20.12 The hybridConvectionTerm Module

```
class HybridConvectionTerm(coeff=1.0, diffusionTerm=None)
Bases: fipy.terms.convectionTerm.ConvectionTerm
```

The discretization for this `Term` is given by

$$\int_V \nabla \cdot (\vec{u}\phi) dV \simeq \sum_f (\vec{n} \cdot \vec{u})_f \phi_f A_f$$

where $\phi_f = \alpha_f \phi_P + (1 - \alpha_f) \phi_A$ and α_f is calculated using the hybrid scheme. For further details see [Numerical Schemes](#).

Create a `ConvectionTerm` object.

```
>>> from fipy.meshes.grid1D import Grid1D
>>> from fipy.variables.cellVariable import CellVariable
>>> from fipy.variables.faceVariable import FaceVariable
>>> 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):
...
TypeError: The coefficient must be a vector value.
>>> __ConvectionTerm(coeff = fv)
Traceback (most recent call last):
...
TypeError: 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,))
>>> from fipy.terms.explicitUpwindConvectionTerm import ExplicitUpwindConvectionTerm
>>> ExplicitUpwindConvectionTerm(coeff = (0,)).solve(var = cv)
>>> ExplicitUpwindConvectionTerm(coeff = 1).solve(var = cv)
Traceback (most recent call last):
...
TypeError: The coefficient must be a vector value.
>>> from fipy.meshes.grid2D import Grid2D
>>> 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=_ArithmetricCellToFaceVariable(value=array([[ 0.,  0.,  0.,  0.,  0.,  0.,
[ 0.,  0.,  0.,  0.,  0.,  0.]]), mesh=UniformGrid2D(dx=1.0, dy=1.0, nx=2, 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, dy=1.0, nx=2, ny=1)))
>>> ExplicitUpwindConvectionTerm(coef = ((0,),(0,))).solve(var=cv2)
>>> ExplicitUpwindConvectionTerm(coef = (0,0)).solve(var=cv2)
```

Parameters

- *coeff* : The *Term*'s coefficient value.
- *diffusionTerm* : **deprecated**. The Peclet number is calculated automatically.

20.13 The implicitSourceTerm Module

```
class ImplicitSourceTerm(coeff=0.0)
Bases: fipy.terms.sourceTerm.SourceTerm
```

The *ImplicitSourceTerm* represents

$$\int_V \phi S dV \simeq \phi_P S_P V_P$$

where *S* is the *coeff* value.

20.14 The mulTerm Module

20.15 The nthOrderDiffusionTerm Module

```
class ExplicitNthOrderDiffusionTerm(coeff)
Bases: fipy.terms.explicitDiffusionTerm.ExplicitDiffusionTerm
```

Attention: This class is deprecated. Use *ExplicitDiffusionTerm* instead.

```
class NthOrderDiffusionTerm(coeff)
Bases: fipy.terms.diffusionTerm.DiffusionTerm
```

Attention: This class is deprecated. Use *ImplicitDiffusionTerm* instead.

20.16 The powerLawConvectionTerm Module

```
class PowerLawConvectionTerm(coeff=1.0, diffusionTerm=None)
Bases: fipy.terms.convectionTerm.ConvectionTerm
```

The discretization for this *Term* is given by

$$\int_V \nabla \cdot (\vec{u} \phi) dV \simeq \sum_f (\vec{n} \cdot \vec{u})_f \phi_f A_f$$

where $\phi_f = \alpha_f \phi_P + (1 - \alpha_f) \phi_A$ and α_f is calculated using the power law scheme. For further details see [Numerical Schemes](#).

Create a *ConvectionTerm* object.

```
>>> from fipy.meshes.grid1D import Grid1D
>>> from fipy.variables.cellVariable import CellVariable
>>> from fipy.variables.faceVariable import FaceVariable
>>> 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):
...
TypeError: The coefficient must be a vector value.
>>> __ConvectionTerm(coeff = fv)
Traceback (most recent call last):
...
TypeError: The coefficient must be a vector value.
>>> __ConvectionTerm(coeff = vcv)
__ConvectionTerm(coeff=_ArithmeticCellToFaceVariable(value=array([[ 0.,  0.,  0.]]), mesh=UniformGrid1D(dx=1.0, ny=1))
>>> __ConvectionTerm(coeff = vfv)
__ConvectionTerm(coeff=FaceVariable(value=array([[ 0.,  0.,  0.]]), mesh=UniformGrid1D(dx=1.0, ny=1))
>>> __ConvectionTerm(coeff = (1,))
__ConvectionTerm(coeff=(1,))
>>> from fipy.terms.explicitUpwindConvectionTerm import ExplicitUpwindConvectionTerm
>>> ExplicitUpwindConvectionTerm(coeff = (0,)).solve(var = cv)
>>> ExplicitUpwindConvectionTerm(coeff = 1).solve(var = cv)
Traceback (most recent call last):
...
TypeError: The coefficient must be a vector value.
>>> from fipy.meshes.grid2D import Grid2D
>>> 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, dy=1.0, nx=2, 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, dy=1.0, nx=2, ny=1)))
>>> ExplicitUpwindConvectionTerm(coeff = ((0,), (0,))).solve(var=cv2)
>>> ExplicitUpwindConvectionTerm(coeff = (0,0)).solve(var=cv2)
```

Parameters

- *coeff* : The *Term*'s coefficient value.
- *diffusionTerm* : **deprecated**. The Peclet number is calculated automatically.

20.17 The sourceTerm Module

```
class SourceTerm(coeff=0.0)
Bases: fipy.terms.cellTerm.CellTerm
```

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

20.18 The term Module

`class Term(coeff=1.0)`

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

Create a *Term*.

Parameters

- `coeff`: The coefficient for the term. A *CellVariable* or number. *FaceVariable* objects are also acceptable for diffusion or convection terms.

`cacheMatrix()`

Informs *solve()* and *sweep()* to cache their matrix so that *getMatrix()* can return the matrix.

`cacheRHSvector()`

Informs *solve()* and *sweep()* to cache their right hand side vector so that *getRHSvector()* can return it.

`copy()`

`getDefaultSolver(solver=None, *args, **kwargs)`

`getMatrix()`

Return the matrix calculated in *solve()* or *sweep()*. The *cacheMatrix()* method should be called before *solve()* or *sweep()* to cache the matrix.

`getRHSvector()`

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

`justResidualVector(var, solver=None, boundaryConditions=(), dt=1.0, 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`: The variable to be solved for. Provides the initial condition, the old value and holds the solution on completion.
- `solver`: The iterative solver to be used to solve the linear system of equations. Defaults to *LinearPCGSolver* for Pysparse and *LinearLUSolver* for Trilinos.
- `boundaryConditions`: A tuple of boundaryConditions.
- `dt`: The time step size.
- `underRelaxation`: Usually a value between 0 and 1 or *None* in the case of no under-relaxation
- `residualFn`: A function that takes var, matrix, and RHSvector arguments used to customize the residual calculation.

`residualVectorAndNorm(var, solver=None, boundaryConditions=(), dt=1.0, 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*: The variable to be solved for. Provides the initial condition, the old value and holds the solution on completion.
- *solver*: The iterative solver to be used to solve the linear system of equations. Defaults to *LinearPCGSolver* for Pysparse and *LinearLUSolver* for Trilinos.
- *boundaryConditions*: A tuple of boundaryConditions.
- *dt*: The time step size.
- *underRelaxation*: Usually a value between 0 and 1 or *None* in the case of no under-relaxation
- *residualFn*: A function that takes var, matrix, and RHSvector arguments used to customize the residual calculation.

solve (*var, solver=None, boundaryConditions=(), dt=1.0*)

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*: The variable to be solved for. Provides the initial condition, the old value and holds the solution on completion.
- *solver*: The iterative solver to be used to solve the linear system of equations. Defaults to *LinearPCGSolver* for Pysparse and *LinearLUSolver* for Trilinos.
- *boundaryConditions*: A tuple of boundaryConditions.
- *dt*: The time step size.

sweep (*var, solver=None, boundaryConditions=(), dt=1.0, underRelaxation=None, residualFn=None*)

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*: The variable to be solved for. Provides the initial condition, the old value and holds the solution on completion.
- *solver*: The iterative solver to be used to solve the linear system of equations. Defaults to *LinearPCGSolver* for Pysparse and *LinearLUSolver* for Trilinos.
- *boundaryConditions*: A tuple of boundaryConditions.
- *dt*: The time step size.
- *underRelaxation*: Usually a value between 0 and 1 or *None* in the case of no under-relaxation
- *residualFn*: A function that takes var, matrix, and RHSvector arguments, used to customize the residual calculation.

20.19 The test Module

20.20 The transientTerm Module

```
class TransientTerm(coeff=1.0)
Bases: fipy.terms.cellTerm.CellTerm
```

The *TransientTerm* represents

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

where ρ is the *coeff* value.

The following test case verifies that variable coefficients and old coefficient values work correctly. We will solve the following equation

$$\frac{\partial\phi^2}{\partial t} = k.$$

The analytic solution is given by

$$\phi = \sqrt{\phi_0^2 + kt},$$

where ϕ_0 is the initial value.

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

>>> from fipy.meshes.grid1D import Grid1D
>>> mesh = Grid1D(nx = 1)
>>> from fipy.variables.cellVariable import CellVariable
>>> var = CellVariable(mesh = mesh, value = phi0, hasOld = 1)
>>> from fipy.terms.transientTerm import TransientTerm
>>> from fipy.terms.implicitSourceTerm import ImplicitSourceTerm
```

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

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

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

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

Compare the final result with the analytical solution.

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

20.21 The upwindConvectionTerm Module

```
class UpwindConvectionTerm(coeff=1.0, diffusionTerm=None)
Bases: fipy.terms.convectionTerm.ConvectionTerm
```

The discretization for this `Term` is given by

$$\int_V \nabla \cdot (\vec{u}\phi) dV \simeq \sum_f (\vec{n} \cdot \vec{u})_f \phi_f A_f$$

where $\phi_f = \alpha_f \phi_P + (1 - \alpha_f) \phi_A$ and α_f is calculated using the upwind convection scheme. For further details see [Numerical Schemes](#).

Create a `ConvectionTerm` object.

```
>>> from fipy.meshes.grid1D import Grid1D
>>> from fipy.variables.cellVariable import CellVariable
>>> from fipy.variables.faceVariable import FaceVariable
>>> 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):
...
TypeError: The coefficient must be a vector value.
>>> __ConvectionTerm(coeff = fv)
Traceback (most recent call last):
...
TypeError: 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, ny=1))
>>> __ConvectionTerm(coeff = (1,))
__ConvectionTerm(coeff=(1,))
>>> from fipy.terms.explicitUpwindConvectionTerm import ExplicitUpwindConvectionTerm
>>> ExplicitUpwindConvectionTerm(coeff = (0,)).solve(var = cv)
>>> ExplicitUpwindConvectionTerm(coeff = 1).solve(var = cv)
Traceback (most recent call last):
...
TypeError: The coefficient must be a vector value.
>>> from fipy.meshes.grid2D import Grid2D
>>> 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, dy=1.0, nx=2, 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, dy=1.0, nx=2, ny=1)))
>>> ExplicitUpwindConvectionTerm(coeff = ((0,), (0,))).solve(var=cv2)
>>> ExplicitUpwindConvectionTerm(coeff = (0,0)).solve(var=cv2)
```

Parameters

- *coeff* : The *Term*'s coefficient value.
- *diffusionTerm* : **deprecated**. The Peclet number is calculated automatically.

20.22 The vanLeerConvectionTerm Module

```
class VanLeerConvectionTerm(coeff=1.0, diffusionTerm=None)
```

Bases: `fipy.terms.explicitUpwindConvectionTerm.ExplicitUpwindConvectionTerm`

Create a *ConvectionTerm* object.

```
>>> from fipy.meshes.grid1D import Grid1D
>>> from fipy.variables.cellVariable import CellVariable
>>> from fipy.variables.faceVariable import FaceVariable
>>> 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):
...
TypeError: The coefficient must be a vector value.
>>> __ConvectionTerm(coeff = fv)
Traceback (most recent call last):
...
TypeError: The coefficient must be a vector value.
>>> __ConvectionTerm(coeff = vcv)
__ConvectionTerm(coeff=_ArithmeticCellToFaceVariable(value=array([[ 0.,  0.,  0.])), mesh=UniformGrid1D(dx=1.0, ny=1))
>>> __ConvectionTerm(coeff = vfv)
__ConvectionTerm(coeff=FaceVariable(value=array([[ 0.,  0.,  0.]]), mesh=UniformGrid1D(dx=1.0, ny=1))
>>> __ConvectionTerm(coeff = (1,))
__ConvectionTerm(coeff=(1,))
>>> from fipy.terms.explicitUpwindConvectionTerm import ExplicitUpwindConvectionTerm
>>> ExplicitUpwindConvectionTerm(coeff = (0,)).solve(var = cv)
>>> ExplicitUpwindConvectionTerm(coeff = 1).solve(var = cv)
Traceback (most recent call last):
...
TypeError: The coefficient must be a vector value.
>>> from fipy.meshes.grid2D import Grid2D
>>> 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, dy=1.0, nx=2, 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, dy=1.0, nx=2, ny=1)))
>>> ExplicitUpwindConvectionTerm(coeff = ((0,), (0,))).solve(var=cv2)
>>> ExplicitUpwindConvectionTerm(coeff = (0,0)).solve(var=cv2)
```

Parameters

- *coeff* : The *Term*'s coefficient value.

- *diffusionTerm* : **deprecated**. The Peclet number is calculated automatically.

test Module Documentation

This page contains the test Module documentation.

21.1 The `test` Module

Chapter 22

tests Package Documentation

This page contains the tests Package documentation.

22.1 The `doctestPlus` Module

`execButNoTest (name='__main__')`

22.2 The `lateImportTest` Module

22.3 The `testBase` Module

22.4 The `testProgram` Module

`main`
alias of `_TestProgram`

Chapter 23

tools Package Documentation

This page contains the tools Package documentation.

23.1 dimensions Package Documentation

This page contains the dimensions Package documentation.

23.1.1 The DictWithDefault Module

23.1.2 The NumberDict Module

23.1.3 The physicalField Module

Physical quantities with units.

This module derives from Konrad Hinsen's PhysicalQuantity <<http://dirac.cnrs-orleans.fr/ScientificPython/ScientificPythonManual/Scientific.Physics.PhysicalQuantities-module.html>>.

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

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/s**2/m
1 b = 1e-28 m
1 bar = 100000.0 kg/s**2/m
1 Bohr = 5.29177208115e-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 = 1e-05 m**3
1 cup = 0.000236588256 m**3
1 d = 86400.0 s
```

```

1 deg = 0.0174532925199 rad
1 degF = 0.5555555555556 K
1 degR = 0.5555555555556 K
1 dl = 0.0001 m**3
1 dyn = 1e-05 m*kg/s**2
1 e = 1.60217653e-19 A*s
1 eps0 = 8.85418781762e-12 A**2*s**4/kg/m**3
1 erg = 1e-07 m**2*kg/s**2
1 floz = 2.9573532e-05 m**3
1 ft = 0.3048 m
1 g = 0.001 kg
1 galUK = 0.00454609 m**3
1 galUS = 0.003785412096 m**3
1 gn = 9.80665 m/s**2
1 Grav = 6.6742e-11 m**3/s**2/kg
1 h = 3600.0 s
1 ha = 10000.0 m**2
1 Hartree = 4.3597441768e-18 m**2*kg/s**2
1 hbar = 1.05457168236e-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 = 0.0254 m
1 invcm = 1.98644560233e-23 m**2*kg/s**2
1 kB = 1.3806505e-23 kg*m**2/s**2/K
1 kcal = 4184.0 m**2*kg/s**2
1 kcal = 4186.8 m**2*kg/s**2
1 Ken = 1.3806505e-23 m**2*kg/s**2
1 l = 0.001 m**3
1 lb = 0.45359237 kg
1 lyr = 9.46073047258e+15 m
1 me = 9.1093826e-31 kg
1 mi = 1609.344 m
1 min = 60.0 s
1 ml = 1e-06 m**3
1 mp = 1.67262171e-27 kg
1 mu0 = 1.25663706144e-06 kg*m/A**2/s**2
1 Nav = 6.0221415e+23 1/mol
1 nmi = 1852.0 m
1 oz = 0.028349523125 kg
1 psi = 6894.75729317 kg/s**2/m
1 pt = 0.000473176512 m**3
1 qt = 0.000946353024 m**3
1 tbsp = 1.4786766e-05 m**3
1 ton = 907.18474 kg
1 Torr = 133.322368421 kg/s**2/m
1 tsp = 4.928922e-06 m**3
1 wk = 604800.0 s
1 yd = 0.9144 m
1 yr = 31536000.0 s
1 yrJul = 31557600.0 s
1 yrSid = 31558152.96 s

```

class PhysicalField(*value*, *unit=None*, *array=None*)

Bases: `object`

Physical field or quantity with units

Physical Fields can be constructed in one of two ways:

- `PhysicalField(*value*, *unit*)`, where `*value*` is a number of arbitrary type and `*unit*` is a string containing the unit name

```
>>> print PhysicalField(value = 10., unit = 'm')
10.0 m
```

- `PhysicalField(*string*)`, where `*string*` contains both the value and the unit. This form is provided to make interactive use more convenient

```
>>> print PhysicalField(value = "10. m")
10.0 m
```

Dimensionless quantities, with a *unit* of 1, can be specified in several ways

```
>>> print PhysicalField(value = "1")
1.0 1
>>> print PhysicalField(value = 2., unit = " ")
2.0 1
>>> print PhysicalField(value = 2.)
2.0 1
```

Physical arrays are also possible (and are the reason this code was adapted from Konrad 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.

`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()
1.57079632679 rad
```

The input *PhysicalField* must be dimensionless

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

arccosh()

Return the inverse hyperbolic cosine of the *PhysicalField*

```
>>> print PhysicalField(2).arccosh()
1.31695789692
```

The input *PhysicalField* must be dimensionless

```
>>> print 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()
1.57079632679 rad
```

The input *PhysicalField* must be dimensionless

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

arctan()

Return the arctangent of the *PhysicalField* in radians

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

The input *PhysicalField* must be dimensionless

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

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

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

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

The units of the *PhysicalField* must be an angle

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

cosh()

Return the hyperbolic cosine of the *PhysicalField*

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

The units of the *PhysicalField* must be dimensionless

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

divide(*other*)

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

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

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

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

dot (*other*)

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

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

floor ()

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

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

getNumericValue ()

Return the *PhysicalField* without units, after conversion to base SI units.

```
>>> print round(PhysicalField("1 inch").getNumericValue(), 6)
0.0254
```

getShape ()**getUnit** ()

Return the unit object of *self*.

```
>>> PhysicalField("1 m").getUnit()
<PhysicalUnit m>
```

getsctype (*default=None*)

Returns the Numpy dtype 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.,
True
```

inBaseUnits ()

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

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

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()
7088849.01085 kg*m**2/s**2/mol
```

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')
32.0 degF
```

If several units are specified, the return value is a tuple of *PhysicalField* instances 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')
>>> [str(element) for element in t.inUnitsOf('d', 'h', 'min', 's')]
['3.0 d', '15.0 h', '15.0 min', '59.0 s']
```

isCompatible(unit)**itemset(value)**

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

```
>>> a = PhysicalField(4., "m")
>>> a.itemset(PhysicalField("6 ft"))
>>> print a
1.8288 m
>>> a = PhysicalField(((3., 4.), (5., 6.)), "m")
>>> a.itemset(PhysicalField("6 ft"))
Traceback (most recent call last):
...
ValueError: can only place a scalar for an array of size 1
>>> a.itemset(PhysicalField("2 min"))
Traceback (most recent call last):
...
TypeError: Incompatible units
```

log()

Return the natural logarithm of the *PhysicalField*

```
>>> print round(PhysicalField(10).log(), 6)
2.302585
```

The input *PhysicalField* must be dimensionless

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

log10()

Return the base-10 logarithm of the *PhysicalField*

```
>>> print round(PhysicalField(10.).log10(), 6)
1.0
```

The input *PhysicalField* must be dimensionless

```
>>> print 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')
100.0 m*N
```

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

```
>>> print (PhysicalField(10., 's') * PhysicalField(2., 'Hz'))
20.0
```

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
```

setUnit (*unit*)

Change the unit object of *self* to *unit*

```
>>> a = PhysicalField(value="1 m")
>>> a.setUnit("m**2/s")
>>> print a
1.0 m**2/s
```

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*

```
>>> round(PhysicalField(numerix.pi/4,"rad").tan(), 6)
1.0
>>> 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
```

class PhysicalUnit(*names*, *factor*, *powers*, *offset*=0)

A *PhysicalUnit* represents the units of a *PhysicalField*.

This class is not generally instantiated by users of this module, but rather it is created in the process of constructing a *PhysicalField*.

Parameters

- *names*: the name of the unit
- *factor*: the multiplier between the unit and the fundamental SI unit
- *powers*: a nine-element *list*, *tuple*, or *Numeric array* representing the fundamental SI units of [“m”, “kg”, “s”, “A”, “K”, “mol”, “cd”, “rad”, “sr”]
- *offset*: the displacement between the zero-point of the unit and the zero-point of the corresponding fundamental SI unit.

conversionFactorTo(*other*)

Return the multiplication factor between two physical units

```
>>> a = PhysicalField("1. mm")
>>> b = PhysicalField("1. inch")
>>> print round(b.getUnit().conversionFactorTo(a.getUnit()), 6)
25.4
```

Units must have the same fundamental SI units

```
>>> c = PhysicalField("1. K")
>>> c.getUnit().conversionFactorTo(a.getUnit())
Traceback (most recent call last):
...
TypeError: Incompatible units
```

If units have different offsets, they must have the same factor

```
>>> d = PhysicalField("1. degC")
>>> c.getUnit().conversionFactorTo(d.getUnit())
1.0
>>> e = PhysicalField("1. degF")
>>> c.getUnit().conversionFactorTo(e.getUnit())
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").getUnit()
>>> b = PhysicalField("1. degF").getUnit()
>>> [str(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").getUnit().isAngle()
1
>>> PhysicalField("1. rad").getUnit().isAngle()
1
>>> PhysicalField("1. inch").getUnit().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.getUnit().isCompatible(b.getUnit())),
...                                [True, True, True, True, True, True, True, True])
True
>>> c = PhysicalField("1. K")
>>> print numerix.allclose(a.getUnit().isCompatible(c.getUnit())),
...                                [False, True, True, False, True, True, True, True])
True
```

isDimensionless()

Returns *True* if the unit is dimensionless

```
>>> PhysicalField("1. m/m").getUnit().isDimensionless()
1
>>> PhysicalField("1. inch").getUnit().isDimensionless()
0
```

isDimensionlessOrAngle()

Returns *True* if the unit is dimensionless or an angle

```
>>> PhysicalField("1. m/m").getUnit().isDimensionlessOrAngle()
1
>>> PhysicalField("1. deg").getUnit().isDimensionlessOrAngle()
1
>>> PhysicalField("1. rad").getUnit().isDimensionlessOrAngle()
1
>>> PhysicalField("1. inch").getUnit().isDimensionlessOrAngle()
0
```

isInverseAngle()

Returns *True* if the 1 divided by the unit is an angle

```
>>> PhysicalField("1. deg**-1").getUnit().isInverseAngle()
1
>>> PhysicalField("1. 1/rad").getUnit().isInverseAngle()
1
>>> PhysicalField("1. inch").getUnit().isInverseAngle()
0
```

name()

Return the name of the unit

```
>>> PhysicalField("1. m").getUnit().name()
'm'
>>> (PhysicalField("1. m") / PhysicalField("1. s"))
... / PhysicalField("1. s")).getUnit().name()
'm/s**2'

setName(name)
Set the name of the unit to name

>>> a = PhysicalField("1. m/s").getUnit()
>>> a
<PhysicalUnit m/s>
>>> a.setName('meterpersecond')
>>> a
<PhysicalUnit meterpersecond>
```

23.2 The tools Package

```
class Parallel()
    Bases: object

class Serial()
    Bases: object
```

23.3 The debug Module

PRINT(*label*, **args*, ***kwargs*)

23.4 The dump Module

read(*filename*, *fileobject*=None)

Read a pickled object from a file. Returns the unpickled object. Wrapper for *cPickle.load()*.

Parameters

- *filename*: The name of the file to unpickle the object from.
- *fileobject*: Used to remove temporary files

write(*data*, *filename*=None, *extension*=”)

Pickle an object and write it to a file. Wrapper for *cPickle.dump()*.

Parameters

- *data*: The object to be pickled.
- *filename*: The name of the file to place the pickled object. If *filename* is *None* then a temporary file will be used and the file object and file name will be returned as a tuple
- *extension*: Used if filename is not given.

Test to check pickling and unpickling.

```
>>> from fipy.meshes.grid1D import Grid1D
>>> old = Grid1D(nx = 2)
>>> f, tempfile = write(old)
>>> new = read(tempfile, f)
>>> print old.getNumberOfCells() == new.getNumberOfCells()
True
```

23.5 The `inline` Module

23.6 The `memoryLeak` Module

This python script is ripped from <http://www.nightmare.com/medusa/memory-leaks.html>

It outputs the top 100 number of outstanding references for each object.

23.7 The `memoryLogger` Module

```
class MemoryHighWaterThread(pid, sampleTime=1)
    Bases: threading.Thread
        run()
        stop()

class MemoryLogger(sampleTime=1)

    start()
    stop()
```

23.8 The `memoryUsage` Module

This python script is ripped from http://aspn.activestate.com/ASPN/Cookbook/Python/Recipe/286222/index_txt

23.9 The `numerix` Module

The functions provided in this module replace the *Numeric* 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
numerix.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.]
```

Eventually, this module will be the only place in the code where *Numeric* (or *numarray* (or *scipy_core*)) is explicitly imported.

L1norm(arr)

Parameters

- *arr*: The *array* to evaluate.

Returns $\|arr\|_1 = \sum_{j=1}^n |arr_j|$ is the L^1 -norm of *arr*.

L2norm(arr)

Parameters

- *arr*: The *array* to evaluate.

Returns $\|arr\|_2 = \sqrt{\sum_{j=1}^n |arr_j|^2}$ is the L^2 -norm of *arr*.

LINFnorm(arr)

Parameters

- *arr*: The *array* to evaluate.

Returns $\frac{\|arr\|_\infty = [\sum_{j=1}^n |arr_j|^\infty]^\infty}{\max_j |arr_j|}$ is the L^∞ -norm of *arr*.

allclose(first, second, rtol=1.000000000000001e-05, atol=1e-08)

Tests whether or not *first* and *second* are equal, subject to the given relative and absolute tolerances, such that:

```
|first - second| < atol + rtol * |second|
```

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

allequal(first, second)

Returns *true* if every element of *first* is equal to the corresponding element of *second*.

arccos(arr)

Inverse cosine of *x*, $\cos^{-1} x$

```
>>> print tostring(arccos(0.0), precision=3)
1.571
```

```
>>> isnan(arccos(2.0))
True
```

```
>>> print tostring(arccos(array((0,0.5,1.0))), precision=3)
[ 1.571  1.047  0.    ]
>>> from fipy.variables.variable import Variable
>>> arccos(Variable(value=(0,0.5,1.0)))
numerix.arccos(Variable(value=array([ 0. ,  0.5,  1. ])))
```

Attention: the next should really return radians, but doesn't

```
>>> print tostring(arccos(Variable(value=(0,0.5,1.0))), precision=3)
[ 1.571  1.047  0.    ]
```

arccosh(*arr*)

Inverse hyperbolic cosine of x , $\cosh^{-1} x$

```
>>> print arccosh(1.0)
0.0
```

```
>>> isnan(arccosh(0.0))
True
```

```
>>> print tostring(arccosh(array((1,2,3))), precision=3)
[ 0.      1.317  1.763]
>>> from fipy.variables.variable import Variable
>>> arccosh(Variable(value=(1,2,3)))
numerix.arccosh(Variable(value=array([1, 2, 3])))
>>> print tostring(arccosh(Variable(value=(1,2,3))), precision=3)
[ 0.      1.317  1.763]
```

arcsin(*arr*)

Inverse sine of x , $\sin^{-1} x$

```
>>> print tostring(arcsin(1.0), precision=3)
1.571
```

```
>>> isnan(arcsin(2.0))
True
```

```
>>> print tostring(arcsin(array((0,0.5,1.0))), precision=3)
[ 0.      0.524   1.571]
>>> from fipy.variables.variable import Variable
>>> arcsin(Variable(value=(0,0.5,1.0)))
numerix.arcsin(Variable(value=array([ 0. ,  0.5,  1. ])))
```

Attention: the next should really return radians, but doesn't

```
>>> print tostring(arcsin(Variable(value=(0,0.5,1.0))), precision=3)
[ 0.      0.524   1.571]
```

arcsinh(*arr*)

Inverse hyperbolic sine of x , $\sinh^{-1} x$

```
>>> print tostring(arcsinh(1.0), precision=3)
0.881
>>> print tostring(arcsinh(array((1,2,3))), precision=3)
[ 0.881  1.444  1.818]
>>> from fipy.variables.variable import Variable
>>> arcsinh(Variable(value=(1,2,3)))
numerix.arcsinh(Variable(value=array([1, 2, 3])))
>>> print tostring(arcsinh(Variable(value=(1,2,3))), precision=3)
[ 0.881  1.444  1.818]
```

arctan(*arr*)

Inverse tangent of x , $\tan^{-1} x$

```
>>> print tostring(arctan(1.0), precision=3)
0.785
>>> print tostring(arctan(array((0,0.5,1.0))), precision=3)
[ 0.          0.464   0.785]
>>> from fipy.variables.variable import Variable
>>> arctan(Variable(value=(0,0.5,1.0)))
numerix.arctan(Variable(value=array([ 0. ,  0.5,  1. ])))
```

Attention: the next should really return radians, but doesn't

```
>>> print tostring(arctan(Variable(value=(0,0.5,1.0))), precision=3)
[ 0.          0.464   0.785]
```

arctan2(*arr, other*)

Inverse tangent of a ratio x/y , $\tan^{-1} \frac{x}{y}$

```
>>> print tostring(arctan2(3.0, 3.0), precision=3)
0.785
>>> print tostring(arctan2(array((0, 1, 2)), 2), precision=3)
[ 0.          0.464   0.785]
>>> from fipy.variables.variable import Variable
>>> arctan2(Variable(value=(0, 1, 2)), 2)
(numerix.arctan2(Variable(value=array([0, 1, 2])), 2))
```

Attention: the next should really return radians, but doesn't

```
>>> print tostring(arctan2(Variable(value=(0, 1, 2)), 2), precision=3)
[ 0.          0.464   0.785]
```

arctanh(*arr*)

Inverse hyperbolic tangent of x , $\tanh^{-1} x$

```
>>> print tostring(arctanh(0.5), precision=3)
0.549
>>> print tostring(arctanh(array((0,0.25,0.5))), precision=3)
[ 0.          0.255   0.549]
>>> from fipy.variables.variable import Variable
>>> arctanh(Variable(value=(0,0.25,0.5)))
numerix.arctanh(Variable(value=array([ 0. ,  0.25,  0.5 ])))
```

```
>>> print tostring(arctanh(Variable(value=(0, 0.25, 0.5))), precision=3)
[ 0.      0.255   0.549]
```

ceil (arr)

The largest integer $\geq x$, $\lceil x \rceil$

```
>>> print ceil(2.3)
3.0
>>> print ceil(array((-1.5, 2, 2.5)))
[-1.  2.  3.]
>>> from fipy.variables.variable import Variable
>>> ceil(Variable(value=(-1.5, 2, 2.5), unit="m**2"))
numerix.ceil(Variable(value=PhysicalField(array([-1.5, 2., 2.5]), 'm**2')))
>>> print ceil(Variable(value=(-1.5, 2, 2.5), unit="m**2"))
[-1.  2.  3.] m**2
```

conjugate (arr)

Complex conjugate of $z = x + iy$, $z^* = x - iy$

```
>>> print conjugate(3 + 4j) == 3 - 4j
True
>>> print allclose(conjugate(array((3 + 4j, -2j, 10))), (3 - 4j, 2j, 10))
1
>>> from fipy.variables.variable import Variable
>>> var = conjugate(Variable(value=(3 + 4j, -2j, 10), unit="ohm"))
>>> print var.getUnit()
<PhysicalUnit ohm>
>>> print allclose(var.getNumericValue(), (3 - 4j, 2j, 10))
1
```

cos (arr)

Cosine of x , $\cos x$

```
>>> print allclose(cos(2*pi/6), 0.5)
True
>>> print tostring(cos(array((0, 2*pi/6, pi/2))), precision=3, suppress_small=1)
[ 1.      0.5     0. ]
>>> from fipy.variables.variable import Variable
>>> cos(Variable(value=(0, 2*pi/6, pi/2), unit="rad"))
numerix.cos(Variable(value=PhysicalField(array([ 0.           , 1.04719755, 1.57079633]), 'rad')))
>>> print tostring(cos(Variable(value=(0, 2*pi/6, pi/2), unit="rad")), suppress_small=1)
[ 1.      0.5     0. ]
```

cosh (arr)

Hyperbolic cosine of x , $\cosh x$

```
>>> print cosh(0)
1.0
>>> print tostring(cosh(array((0, 1, 2))), precision=3)
[ 1.      1.543   3.762]
>>> from fipy.variables.variable import Variable
>>> cosh(Variable(value=(0, 1, 2)))
numerix.cosh(Variable(value=array([0, 1, 2])))
>>> print tostring(cosh(Variable(value=(0, 1, 2))), precision=3)
[ 1.      1.543   3.762]
```

dot (a1, a2, axis=0)

return array of vector dot-products of v1 and v2 for arrays a1 and a2 of vectors v1 and v2

We can't use `numpy.dot()` on an array of vectors

Test that Variables are returned as Variables.

```
>>> from fipy.meshes.grid2D 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)._getVariableClass()
<class 'fipy.variables.cellVariable.CellVariable'>
>>> dot(v2, v1)._getVariableClass()
<class 'fipy.variables.cellVariable.CellVariable'>
>>> print rank(dot(v2, v1))
0
>>> print dot(v1, v2)
[ 4 10]
>>> dot(v1, v1)._getVariableClass()
<class 'fipy.variables.cellVariable.CellVariable'>
>>> print dot(v1, v1)
[ 4 10]
>>> v3 = array(((0,1),(2,3)))
>>> type(dot(v3, v3))
<type 'numpy.ndarray'>
>>> print dot(v3, v3)
[ 4 10]
```

exp (arr)

Natural exponent of x , e^x

floor (arr)

The largest integer $\leq x$, $\lfloor x \rfloor$

```
>>> print floor(2.3)
2.0
>>> print floor(array((-1.5,2,2.5)))
[-2. 2. 2.]
>>> from fipy.variables.variable import Variable
>>> floor(Variable(value=(-1.5,2,2.5), unit="m**2"))
numerix.floor(Variable(value=PhysicalField(array([-1.5, 2., 2.5]),'m**2')))
>>> print floor(Variable(value=(-1.5,2,2.5), unit="m**2"))
[-2. 2. 2.] m**2
```

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"))
()

getUnit(arr)

indices(dimensions, typecode=None)
    indices(dimensions, typecode=None) returns an array representing a grid of indices with row-only, and column-only variation.

>>> NUMERIX.allclose(NUMERIX.array(indices((4, 6))), NUMERIX.indices((4, 6)))
1
>>> NUMERIX.allclose(NUMERIX.array(indices((4, 6, 2))), NUMERIX.indices((4, 6, 2)))
1
>>> NUMERIX.allclose(NUMERIX.array(indices((1,))), NUMERIX.indices((1,)))
1
>>> NUMERIX.allclose(NUMERIX.array(indices((5,))), NUMERIX.indices((5,)))
1

isFloat(arr)

isInt(arr)

isclose(first, second, rtol=1.000000000000001e-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.

log(arr)
    Natural logarithm of x,  $\ln x \equiv \log_e x$ 

>>> print tostring(log(10), precision=3)
2.303
>>> print tostring(log(array((0.1, 1, 10))), precision=3)
[-2.303  0.       2.303]
>>> from fipy.variables.variable import Variable
>>> log(Variable(value=(0.1, 1, 10)))
numerix.log(Variable(value=array([ 0.1,    1. ,   10. ])))
>>> print tostring(log(Variable(value=(0.1, 1, 10))), precision=3)
[-2.303  0.       2.303]

log10(arr)
    Base-10 logarithm of x,  $\log_{10} x$ 

>>> print log10(10)
1.0
>>> print log10(array((0.1, 1, 10)))
[-1.  0.  1.]
>>> from fipy.variables.variable import Variable
>>> log10(Variable(value=(0.1, 1, 10)))
numerix.log10(Variable(value=array([ 0.1,    1. ,   10. ])))
>>> print log10(Variable(value=(0.1, 1, 10)))
[-1.  0.  1.]

```

obj2sctype (*rep*, *default=None*)

ones (*a*, *t='l'*)

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]
```

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.

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).

sign (*arr*)

sin (*arr*)

Sine of *x*, sin *x*

```
>>> print sin(pi/6)
0.5
>>> print sin(array((0,pi/6,pi/2)))
[ 0.  0.5  1. ]
>>> from fipy.variables.variable import Variable
>>> sin(Variable(value=(0,pi/6,pi/2), unit="rad"))
numerix.sin(Variable(value=PhysicalField(array([ 0.           ,  0.52359878,  1.57079633]),'rad')))
>>> print sin(Variable(value=(0,pi/6,pi/2), unit="rad"))
[ 0.  0.5  1. ]
```

sinh(*arr*)

Hyperbolic sine of x , $\sinh x$

```
>>> print sinh(0)
0.0
>>> print tostring(sinh(array((0,1,2))), precision=3)
[ 0.      1.175   3.627]
>>> from fipy.variables.variable import Variable
>>> sinh(Variable(value=(0,1,2)))
numerix.sinh(Variable(value=array([0, 1, 2])))
>>> print tostring(sinh(Variable(value=(0,1,2))), precision=3)
[ 0.      1.175   3.627]
```

sqrt(*arr*)

Square root of x , \sqrt{x}

```
>>> print tostring(sqrt(2), precision=3)
1.414
>>> print tostring(sqrt(array((1,2,3))), precision=3)
[ 1.      1.414   1.732]
>>> from fipy.variables.variable import Variable
>>> sqrt(Variable(value=(1, 2, 3), unit="m**2"))
numerix.sqrt(Variable(value=PhysicalField(array([1, 2, 3]),'m**2')))
>>> print tostring(sqrt(Variable(value=(1, 2, 3), unit="m**2")), precision=3)
[ 1.      1.414   1.732] m
```

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*.

sum(*arr*, *axis*=0)

The sum of all the elements of *arr* along the specified axis.

take(*a*, *indices*, *axis*=0, *fill_value*=None)

Selects the elements of *a* corresponding to *indices*.

tan(*arr*)

Tangent of x , $\tan x$

```
>>> print tostring(tan(pi/3), precision=3)
1.732
>>> print tostring(tan(array((0,pi/3,2*pi/3))), precision=3)
[ 0.      1.732 -1.732]
>>> from fipy.variables.variable import Variable
>>> tan(Variable(value=(0,pi/3,2*pi/3), unit="rad"))
numerix.tan(Variable(value=PhysicalField(array([ 0.          , 1.04719755, 2.0943951 ]),'rad')))
>>> print tostring(tan(Variable(value=(0,pi/3,2*pi/3), unit="rad")), precision=3)
[ 0.      1.732 -1.732]
```

tanh(*arr*)

Hyperbolic tangent of x , $\tanh x$

```
>>> print tostring(tanh(1), precision=3)
0.762
>>> print tostring(tanh(array((0,1,2))), precision=3)
[ 0.      0.762   0.964]
>>> from fipy.variables.variable import Variable
```

```
>>> tanh(Variable(value=(0,1,2)))
numerix.tanh(Variable(value=array([0, 1, 2])))
>>> print tostring(tanh(Variable(value=(0,1,2))), precision=3)
[ 0.      0.762   0.964]
```

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.

Parameters

- *max_line_width*: the maximum number of characters used in a single line. Default is *sys.output_line_width* or 77.
- *precision*: the number of digits after the decimal point. Default is *sys.float_output_precision* or 8.
- *suppress_small*: whether small values should be suppressed (and output as 0). Default is *sys.float_output_suppress_small* or *false*.
- *separator*: what character string to place between two numbers.
- *array_output*: Format output for an *eval*. Only used if *arr* is a *Numeric array*.

```
>>> from fipy import Variable
>>> print tostring(Variable((1,0,11.2345)), precision=1)
[ 1.    0.    11.2]
>>> print tostring(array((1,2)), precision=5)
[1 2]
>>> print tostring(array((1.12345,2.79)), precision=2)
[ 1.12   2.79]
>>> print tostring(1)
1
>>> print tostring(array(1))
1
>>> print tostring(array([1.23345]), precision=2)
[ 1.23]
>>> print tostring(array([1]), precision=2)
[1]
>>> print tostring(1.123456, precision=2)
1.12
>>> print tostring(array(1.123456), precision=3)
1.123
```

zeros(*a*, *t='l'*)

23.10 The parser Module

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*: The argument to be parsed.

- *action*: *store* or *store_true* are possibilities
- *type*: Type of the argument. *int* or *float* are possibilities.
- *default*: Default value.

23.11 The `pysparseMatrix` Module

23.12 The `sparseMatrix` Module

23.13 The `test` Module

23.14 The `trilinosMatrix` Module

23.15 The `vector` Module

Vector utility functions that are inexplicably absent from Numeric

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])
```

putAdd (*vector, ids, additionVector*)

This is a temporary replacement for Numeric.put as it was not doing what we thought it was doing.

23.16 The `vitals` Module

class Vitals()

Bases: `xml.dom.minidom.Document`

Returns XML formatted information about current FiPy environment

appendChild (*child*)

appendInfo (*name, svnpath=None, **kwargs*)

append some additional information, possibly about a project under a separate svn repository

dictToXML (*d, name*)

save (*fname*)

svn (**args*)

svncmd (*cmd, *args*)

tupleToXML (*t, name, keys=None*)

variables Package Documentation

This page contains the variables Package documentation.

24.1 The addOverFacesVariable Module

24.2 The arithmeticCellToFaceVariable Module

24.3 The betaNoiseVariable Module

```
class BetaNoiseVariable(mesh, alpha, beta, name='', hasOld=0)
    Bases: fipy.variables.noiseVariable.NoiseVariable
```

Represents a beta distribution of random numbers with the probability distribution

$$x^{\alpha-1} \frac{\beta^\alpha e^{-\beta x}}{\Gamma(\alpha)}$$

with a shape parameter α , a rate parameter β , and $\Gamma(z) = \int_0^\infty t^{z-1} e^{-t} dt$.

We generate noise on a uniform cartesian mesh

```
>>> from fipy.variables.variable import Variable
>>> alpha = Variable()
>>> beta = Variable()
>>> from fipy.meshes.grid2D 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.getMesh())
>>> x = histogram.getMesh().getCellCenters()[0]
```

```

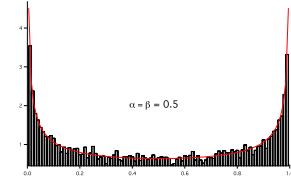
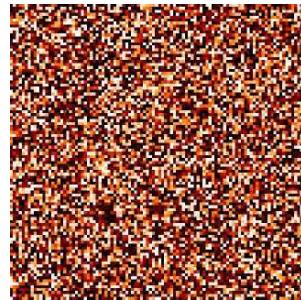
>>> if __name__ == '__main__':
...     from fipy import Viewer
...     viewer = Viewer(vars=noise, datamin=0, datamax=1)
...     histoplot = Viewer(vars=(histogram, betadist),
...                         datamin=0, datamax=1.5)

>>> from fipy.tools.numerix import arange, exp
>>> from scipy.special import gamma as Gamma

>>> for a in arange(0.5,5,0.5):
...     alpha.setValue(a)
...     for b in arange(0.5,5,0.5):
...         beta.setValue(b)
...         betadist.setValue((Gamma(alpha + beta) / (Gamma(alpha) * Gamma(beta)))
...                           * x** (alpha - 1) * (1 - x)** (beta - 1))
...     if __name__ == '__main__':
...         import sys
...         print >>sys.stderr, "alpha: %g, beta: %g" % (alpha, beta)
...         viewer.plot()
...         histoplot.plot()

>>> print abs(noise.getFaceGrad().getDivergence().getCellVolumeAverage()) < 5e-15
1

```



Parameters

- *mesh*: The mesh on which to define the noise.
- *alpha*: The parameter α .
- *beta*: The parameter β .

`random()`

24.4 The `binaryOperatorVariable` Module

24.5 The `cellToFaceVariable` Module

24.6 The `cellVariable` Module

```
class CellVariable(mesh, name='', value=0.0, rank=None, elementshape=None, unit=None, hasOld=0)
Bases: fipy.variables.meshVariable._MeshVariable
```

Represents the field of values of a variable on a *Mesh*.

A *CellVariable* can be pickled to persistent storage (disk) for later use:

```
>>> from fipy.meshes.grid2D 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.getCellCenters()
>>> var.setValue(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

copy()
```

`getArithmeticFaceValue()`

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.grid1D import Grid1D
>>> from fipy import numerix
>>> mesh = Grid1D(dx = (1., 1.))
>>> L = 1
>>> R = 2
>>> var = CellVariable(mesh = mesh, value = (L, R))
>>> faceValue = var.getArithmeticFaceValue() [mesh.getInteriorFaces().getValue()]
>>> 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.getArithmeticFaceValue() [mesh.getInteriorFaces().getValue()]
>>> 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.getArithmeticFaceValue() [mesh.getInteriorFaces().getValue()]
>>> answer = (R - L) * (5.0 / 55.0) + L
>>> print numerix.allclose(faceValue, answer, atol = 1e-10, rtol = 1e-10)
True
```

getCellVolumeAverage()

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.grid2D 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.getCellVolumeAverage()
3.0
```

getFaceGrad()

Return $\nabla\phi$ as a rank-1 *FaceVariable* using differencing for the normal direction(second-order gradient).

getFaceGradAverage()

Return $\nabla\phi$ as a rank-1 *FaceVariable* using averaging for the normal direction(second-order gradient)

getFaceValue()

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.grid1D import Grid1D
>>> from fipy import numerix
>>> mesh = Grid1D(dx = (1., 1.))
>>> L = 1
>>> R = 2
>>> var = CellVariable(mesh = mesh, value = (L, R))
>>> faceValue = var.getArithmeticFaceValue() [mesh.getInteriorFaces().getValue()]
>>> 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.getArithmeticFaceValue() [mesh.getInteriorFaces().getValue()]
>>> 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.getArithmeticFaceValue() [mesh.getInteriorFaces().getValue()]
>>> answer = (R - L) * (5.0 / 55.0) + L
>>> print numerix.allclose(faceValue, answer, atol = 1e-10, rtol = 1e-10)
True
```

getGaussGrad()
 Return $\frac{1}{V_P} \sum_f \vec{n} \phi_f A_f$ as a rank-1 *CellVariable* (first-order gradient).

getGlobalValue()
 Concatenate and return values from all processors
 When running on a single processor, the result is identical to `getValue()`.

getGrad()
 Return $\nabla \phi$ as a rank-1 *CellVariable* (first-order gradient).

getHarmonicFaceValue()
 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.grid1D import Grid1D
>>> from fipy import numerix
>>> mesh = Grid1D(dx = (1., 1.))
>>> L = 1
>>> R = 2
>>> var = CellVariable(mesh = mesh, value = (L, R))
>>> faceValue = var.getHarmonicFaceValue() [mesh.getInteriorFaces().getValue()]
>>> 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.getHarmonicFaceValue() [mesh.getInteriorFaces().getValue()]
>>> 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.getHarmonicFaceValue() [mesh.getInteriorFaces().getValue()]
>>> answer = L * R / ((R - L) * (5.0 / 55.0) + L)
>>> print numerix.allclose(faceValue, answer, atol = 1e-10, rtol = 1e-10)
True
```

getLeastSquaresGrad()
 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)).getLeastSquaresGrad().getGlobalValue(),
...                                [[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)).getLeastSquaresGrad().getGlobalValue(),
True
```

getMinmodFaceValue()

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)).getMinmodFaceValue()
[1 1 2]
>>> print CellVariable(mesh=Grid1D(nx=2), value=(-1, -2)).getMinmodFaceValue()
[-1 -1 -2]
>>> print CellVariable(mesh=Grid1D(nx=2), value=(-1, 2)).getMinmodFaceValue()
[-1 0 2]
```

getOld()

Return the values of the *CellVariable* from the previous solution sweep.

Combinations of *CellVariable*'s should also return old values.

```
>>> from fipy.meshes.grid1D 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.setValue((3,2))
>>> print v
[ 9  8]
>>> print v.getOld()
[ 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.getOld()
[ 6  9]
```

setValue(value, unit=None, where=None)

updateOld()

Set the values of the previous solution sweep to the current values.

24.7 The `cellVolumeAverageVariable` Module

24.8 The `constant` Module

24.9 The `exponentialNoiseVariable` Module

```
class ExponentialNoiseVariable(mesh, mean=0.0, name='', hasOld=0)
```

Bases: `fipy.variables.noiseVariable.NoiseVariable`

Represents an exponential distribution of random numbers with the probability distribution

$$\mu^{-1} e^{-\frac{x}{\mu}}$$

with a mean parameter μ .

We generate noise on a uniform cartesian mesh

```
>>> from fipy.variables.variable import Variable
>>> mean = Variable()
>>> from fipy.meshes.grid2D 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

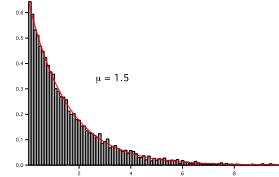
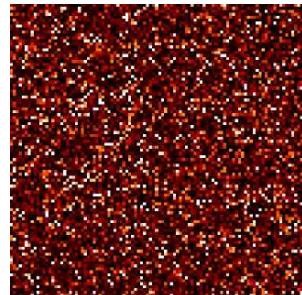
```
>>> from fipy.variables.cellVariable import CellVariable
>>> expdist = CellVariable(mesh = histogram.getMesh())
>>> x = histogram.getMesh().getCellCenters()[0]

>>> if __name__ == '__main__':
...     from fipy import Viewer
...     viewer = Viewer(vars=noise, datamin=0, datamax=5)
...     histoplot = Viewer(vars=(histogram, expdist),
...                         datamin=0, datamax=1.5)
...
...     import sys
...     print >>sys.stderr, "mean: %g" % mean
...     viewer.plot()
...     histoplot.plot()

>>> from fipy.tools.numerix import arange, exp

>>> for mu in arange(0.5, 3, 0.5):
...     mean.setValue(mu)
...     expdist.setValue((1/mean)*exp(-x/mean))
...     if __name__ == '__main__':
...         import sys
...         print >>sys.stderr, "mean: %g" % mean
...         viewer.plot()
...         histoplot.plot()
```

```
>>> print abs(noise.getFaceGrad().getDivergence().getCellVolumeAverage()) < 5e-15
1
```



Parameters

- *mesh*: The mesh on which to define the noise.
- *mean*: The mean of the distribution μ .

`random()`

24.10 The `faceGradContributionsVariable` Module

24.11 The `faceGradVariable` Module

24.12 The `faceVariable` Module

```
class FaceVariable(mesh, name='', value=0.0, rank=None, elementshape=None, unit=None, cached=1)
Bases: fipy.variables.meshVariable._MeshVariable
```

Parameters

- *mesh*: the mesh that defines the geometry of this *Variable*
- *name*: the user-readable name of the *Variable*
- *value*: the initial value
- *rank*: the rank (number of dimensions) of each element of this *Variable*. Default: 0
- ***elementshape***: the shape of each element of this variable Default: $rank \times mesh.getDim()$
- *unit*: the physical units of the *Variable*

`copy()`

`getDivergence()`

```

>>> from fipy.meshes.grid2D import Grid2D
>>> from fipy.variables.cellVariable import CellVariable
>>> mesh = Grid2D(nx=3, ny=2)
>>> var = CellVariable(mesh=mesh, value=range(3*2))
>>> print var.getFaceGrad().getDivergence()
[ 4.  3.  2. -2. -3. -4.]

getGlobalValue()
setValue(value, unit=None, where=None)

```

24.13 The fixedBCFaceGradVariable Module

24.14 The gammaNoiseVariable Module

`class GammaNoiseVariable(mesh, shape, rate, name=”, hasOld=0)`

Bases: `fipy.variables.noiseVariable.NoiseVariable`

Represents a gamma distribution of random numbers with the probability distribution

$$x^{\alpha-1} \frac{\beta^\alpha e^{-\beta x}}{\Gamma(\alpha)}$$

with a shape parameter α , a rate parameter β , and $\Gamma(z) = \int_0^\infty t^{z-1} e^{-t} dt$.

We generate noise on a uniform cartesian mesh

```

>>> from fipy.variables.variable import Variable
>>> alpha = Variable()
>>> beta = Variable()
>>> from fipy.meshes.grid2D 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
>>> gammadist = CellVariable(mesh = histogram.getMesh())
>>> x = histogram.getMesh().getCellCenters()[0]

>>> 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, exp
>>> from scipy.special import gamma as Gamma

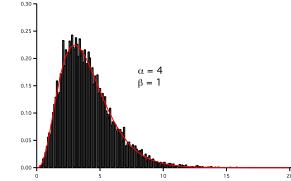
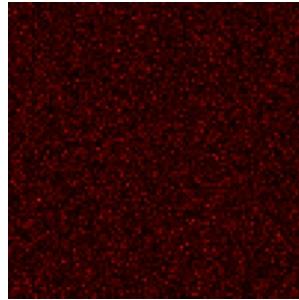
```

```

>>> for shape in arange(1,8,1):
...     alpha.setValue(shape)
...     for rate in arange(0.5,2.5,0.5):
...         beta.setValue(rate)
...         gammadist.setValue(x** (alpha - 1) * (beta**alpha * exp(-beta * x)) / Gamma(alpha))
...         if __name__ == '__main__':
...             import sys
...             print >>sys.stderr, "alpha: %g, beta: %g" % (alpha, beta)
...             viewer.plot()
...             histoplot.plot()

>>> print abs(noise.getFaceGrad().getDivergence().getCellVolumeAverage()) < 5e-15
1

```



Parameters

- *mesh*: The mesh on which to define the noise.
- *shape*: The shape parameter, α .
- *rate*: The rate or inverse scale parameter, β .

`random()`

24.15 The `gaussCellGradVariable` Module

24.16 The `gaussianNoiseVariable` Module

```
class GaussianNoiseVariable(mesh, name='', mean=0.0, variance=1.0, hasOld=0)
Bases: fipy.variables.noiseVariable.NoiseVariable
```

Represents a normal (Gaussian) distribution of random numbers with mean μ and variance $\langle \eta(\vec{r})\eta(\vec{r}') \rangle = \sigma^2$, which has the probability distribution

$$\frac{1}{\sigma\sqrt{2\pi}} \exp -\frac{(x-\mu)^2}{2\sigma^2}$$

For example, the variance of thermal noise that is uncorrelated in space and time is often expressed as

$$\langle \eta(\vec{r}, t)\eta(\vec{r}', t') \rangle = M k_B T \delta(\vec{r} - \vec{r}') \delta(t - t')$$

which can be obtained with:

```
sigmaSqr = Mobility * kBoltzmann * Temperature / (mesh.getCellVolumes() * timeStep)
GaussianNoiseVariable(mesh = mesh, variance = sigmaSqr)
```

Note: If the time step will change as the simulation progresses, either through use of an adaptive iterator or by making manual changes at different stages, remember to declare *timeStep* as a *Variable* and to change its value with its *setValue()* method.

```
>>> import sys
>>> from fipy.tools.numerix import *

>>> mean = 0.
>>> variance = 4.
```

We generate noise on a non-uniform cartesian mesh with cell dimensions of x^2 and y^3 .

```
>>> from fipy.meshes.grid2D 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.getCellVolumes())
>>> 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

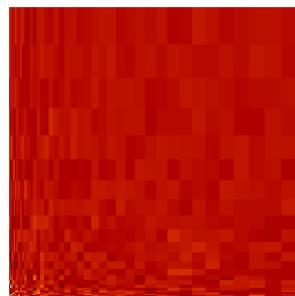
```
>>> gauss = CellVariable(mesh = histogram.getMesh())
>>> x = histogram.getMesh().getCellCenters()[0]
>>> gauss.setValue((1/(sqrt(variance * 2 * pi))) * exp(-(x - mean)**2 / (2 * variance)))

>>> if __name__ == '__main__':
...     from fipy import viewers
...     viewer = Viewer(vars=noise,
...                      datamin=-5, datamax=5)
...     histoplot = Viewer(vars=(histogram, gauss))

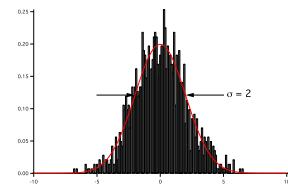
>>> for i in range(10):
...     noise.scramble()
...     if __name__ == '__main__':
...         viewer.plot()
...         histoplot.plot()

>>> print abs(noise.getFaceGrad().getDivergence().getCellVolumeAverage()) < 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*: The mesh on which to define the noise.
- *mean*: The mean of the noise distribution, μ .
- *variance*: The variance of the noise distribution, σ^2 .

`parallelRandom()`

24.17 The harmonicCellToFaceVariable Module

24.18 The histogramVariable Module

```
class HistogramVariable (distribution, dx=1.0, nx=None, offset=0.0)
    Bases: fipy.variables.cellVariable.CellVariable
```

Produces a histogram of the values of the supplied distribution.

Parameters

- *distribution*: The collection of values to sample.
- *dx*: the bin size
- *nx*: the number of bins
- *offset*: the position of the first bin

24.19 The leastSquaresCellGradVariable Module

24.20 The meshVariable Module

24.21 The minmodCellToFaceVariable Module

24.22 The modCellGradVariable Module

24.23 The modCellToFaceVariable Module

24.24 The modFaceGradVariable Module

24.25 The modPhysicalField Module

24.26 The modularVariable Module

```
class ModularVariable(mesh, name='', value=0.0, rank=None, elementshape=None, unit=None, hasOld=0)
    Bases: fipy.variables.cellVariable.CellVariable
```

The *ModularVariable* defines a variable that exists on the circle between $-\pi$ and π

The following examples show how *ModularVariable* works. When subtracting the answer wraps back around the circle.

```
>>> from fipy.meshes.grid1D 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.getArithmeticFaceValue(), (2*pi/3, pi, -2*pi/3))
1
```

Obtaining the gradient.

```
>>> print numerix.allclose(v1.getGrad(), ((pi/3, pi/3),))
1
```

Obtaining the gradient at the faces.

```
>>> print numerix.allclose(v1.getFaceGrad(), ((0, 2*pi/3, 0),))
1
```

Obtaining the gradient at the faces but without modular arithmetic.

```
>>> print numerix.allclose(v1.getFaceGradNoMod(), ((0, -4*pi/3, 0),))  
1
```

getArithmeticFaceValue()

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*

getFaceGrad()

Return $\nabla\phi$ as a rank-1 *FaceVariable* (second-order gradient). Adjusted for a *ModularVariable*

getFaceGradNoMod()

Return $\nabla\phi$ as a rank-1 *FaceVariable* (second-order gradient). Not adjusted for a *ModularVariable*

getGrad()

Return $\nabla\phi$ as a rank-1 *CellVariable* (first-order gradient). Adjusted for a *ModularVariable*

updateOld()

Set the values of the previous solution sweep to the current values. Test case due to bug.

```
>>> from fipy.meshes.grid1D 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.getOld().allclose(answer)  
True
```

24.27 The noiseVariable Module

```
class NoiseVariable(mesh, name='', hasOld=0)  
Bases: fipy.variables.cellVariable.CellVariable
```

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

A generic base class for sources of noise distributed over the cells of a mesh.

In the event that the noise should be conserved, use:

```
<Specific>NoiseVariable(...).getFaceGrad().getDivergence()
```

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.

copy()

Copy the value of the *NoiseVariable* to a static *CellVariable*.

parallelRandom()**random()****scramble()**

Generate a new random distribution.

24.28 The operatorVariable Module

24.29 The scharfetterGummelFaceVariable Module

```
class ScharfetterGummelFaceVariable(var, boundaryConditions=())
Bases: fipy.variables.cellToFaceVariable._CellToFaceVariable
```

24.30 The test Module

Test numeric implementation of the mesh

24.31 The unaryOperatorVariable Module

24.32 The uniformNoiseVariable Module

```
class UniformNoiseVariable(mesh, name='', minimum=0.0, maximum=1.0, hasOld=0)
Bases: fipy.variables.noiseVariable.NoiseVariable
```

Represents a uniform distribution of random numbers.

We generate noise on a uniform cartesian mesh

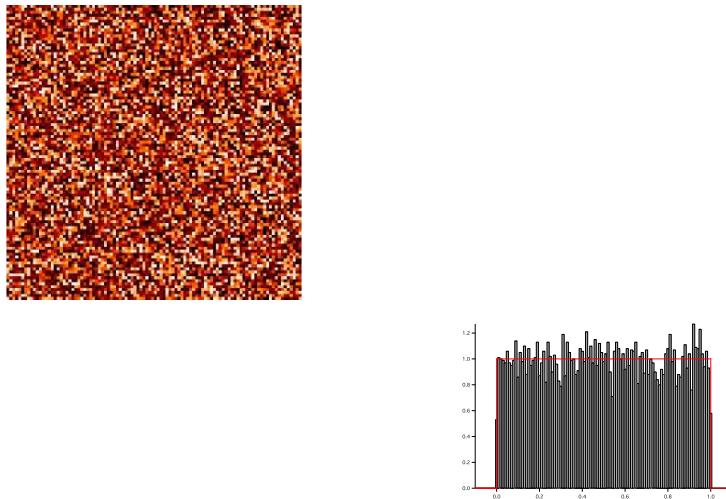
```
>>> from fipy.meshes.grid2D import Grid2D
>>> noise = UniformNoiseVariable(mesh=Grid2D(nx=100, ny=100))
```

and histogram the noise

```
>>> from fipy.variables.histogramVariable import HistogramVariable
>>> histogram = HistogramVariable(distribution=noise, dx=0.01, nx=120, offset=-.1)

>>> if __name__ == '__main__':
...     from fipy import Viewer
...     viewer = Viewer(vars=noise,
...                      datamin=0, datamax=1)
...     histoplot = Viewer(vars=histogram)

>>> for i in range(10):
...     noise.scramble()
...     if __name__ == '__main__':
...         viewer.plot()
...         histoplot.plot()
```



Parameters

- *mesh*: The mesh on which to define the noise.
- *minimum*: The minimum (not-inclusive) value of the distribution.
- *maximum*: The maximum (not-inclusive) value of the distribution.

`random()`

24.33 The `variable` Module

```
class Variable(value=0.0, unit=None, array=None, name='', cached=1)
Bases: object
```

Lazily evaluated quantity with units.

Using a `Variable` in a mathematical expression will create an automatic dependency `Variable`, e.g.,

```
>>> a = Variable(value=3)
>>> b = 4 * a
>>> b
(Variable(value=array(3)) * 4)
>>> b()
12
```

Changes to the value of a `Variable` will automatically trigger changes in any dependent `Variable` objects

```
>>> a.setValue(5)
>>> b
(Variable(value=array(5)) * 4)
>>> 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)))
Variable(value=PhysicalField(array([[3, 3],
[3, 3]]), 'm'))
```

Parameters

- *value*: the initial value
- *unit*: the physical units of the *Variable*
- *array*: the storage array for the *Variable*
- *name*: the user-readable name of the *Variable*
- *cached*: whether to cache or always recalculate the value

all (*axis=None*)

```
>>> print Variable(value=(0, 0, 1, 1)).all()
0
>>> print Variable(value=(1, 1, 1, 1)).all()
1
```

allclose (*other*, *rtol=1.000000000000001e-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))
False
```

allequal (*other*)**any** (*axis=None*)

```
>>> print Variable(value=0).any()
0
>>> print Variable(value=(0, 0, 1, 1)).any()
1
```

arccos ()**arccosh** ()**arcsin** ()**arcsinh** ()**arctan** ()**arctan2** (*other*)**arctanh** ()

cacheMe (*recursive=False*)
ceil()
conjugate()
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]))
```

cos()
cosh()
dontCacheMe (*recursive=False*)
dot (*other, opShape=None, operatorClass=None, axis=0*)
exp()
floor()
getMag()
getName()
getNumericValue()
getShape()

```
>>> Variable(value=3).shape
()
>>> Variable(value=(3,)).shape
(1,)
>>> Variable(value=(3,4)).shape
(2,)

>>> Variable(value="3 m").shape
()
>>> Variable(value=(3,), unit="m").shape
(1,)
```

```
>>> Variable(value=(3, 4), unit="m").shape
(2,)
```

getSubscribedVariables()

getUnit()

Return the unit object of *self*.

```
>>> Variable(value="1 m").getUnit()
<PhysicalUnit m>
```

getValue()

“Evaluate” the *Variable* and return its value (longhand)

```
>>> a = Variable(value=3)
>>> print a.getValue()
3
>>> b = a + 4
>>> b
(Variable(value=array(3)) + 4)
>>> b.getValue()
7
```

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()
7088849.01085 kg*m**2/s**2/mol
```

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')
32.0 degF
```

If several units are specified, the return value is a tuple of *Variable* instances with one element per unit such that the sum of all quantities in the tuple equals the 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')
>>> [str(element) for element in t.inUnitsOf('d','h','min','s')]
['3.0 d', '15.0 h', '15.0 min', '59.0 s']
```

itemset (*value*)

log ()

log10 ()

max (*axis=None*)

min (*axis=None*)

put (*indices, value*)

reshape (*shape*)

setName (*name*)

setUnit (*unit*)

Change the unit object of *self* to *unit*

```
>>> a = Variable(value="1 m")
>>> a.setUnit("m**2/s")
>>> print a
1.0 m**2/s
```

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.setValue(3)
>>> print a
[3 3 3]
```

```
>>> b = numerix.array((3,4,5))
>>> a.setValue(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
```

shape

Tuple of array dimensions.

```
sign()
sin()
sinh()
sqrt()

>>> from fipy.meshes.grid1D import Grid1D
>>> mesh= Grid1D(nx=3)

>>> from fipy.variables.cellVariable import CellVariable
>>> var = CellVariable(mesh=mesh, value=((0., 2., 3.),), rank=1)
>>> print (var.dot(var)).sqrt()
[ 0.  2.  3.]

sum(axis=None)
take(ids, axis=0)
tan()
tanh()
toString(max_line_width=75, precision=8, suppress_small=False, separator=' ')
transpose()
```


viewers Package Documentation

This page contains the viewers Package documentation.

25.1 gistViewer Package Documentation

This page contains the gistViewer Package documentation.

25.1.1 The gistViewer Package

GistViewer (*vars*, *title=None*, *limits={}*, ***kwlimits*)

Generic function for creating a *GistViewer*.

The *GistViewer* factory will search the module tree and return an instance of the first *GistViewer* it finds of the correct dimension.

Parameters

vars a *CellVariable* or tuple of *CellVariable* objects to plot

title displayed at the top of the *Viewer* window

limits [dict] a (deprecated) alternative to limit keyword arguments

xmin, **xmax**, **ymin**, **ymax**, **datamin**, **datamax** 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.

class Gist1DViewer (*vars*, *title=None*, *xlog=0*, *ylog=0*, *style='work.gs'*, *limits={}*, ***kwlimits*)

Bases: `fipy.viewers.gistViewer.gistViewer._GistViewer`

Displays a y vs. x plot of one or more 1D *CellVariable* objects.

```
>>> from fipy import *
>>> mesh = Grid1D(nx=100)
>>> x, = mesh.getCellCenters()
>>> xVar = CellVariable(mesh=mesh, name="x", value=x)
>>> k = Variable(name="k", value=0.)
>>> viewer = Gist1DViewer(vars=(sin(k * xVar), cos(k * xVar / pi)),
...                         limits={'xmin': 10, 'xmax': 90},
...                         datamin=-0.9, datamax=2.0,
...                         title="Gist1DViewer test")
>>> for kval in numerix.arange(0, 0.3, 0.03):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()
```

Creates a *Gist1DViewer*.

Parameters

vars a *CellVariable* or tuple of *CellVariable* objects to plot
title displayed at the top of the *Viewer* window
xlog log scaling of x axis if *True*
ylog log scaling of y axis if *True*
style the Gist stylefile to use.
limits [dict] a (deprecated) alternative to limit keyword arguments
xmin, xmax, datamin, datamax displayed range of data. Any limit set to a (default) value of *None* will autoscale. (*ymin* and *ymax* are synonyms for *datamin* and *datamax*).
plot (*filename=None*)

class Gist2DViewer (vars, title=None, palette='heat.gp', grid=True, dpi=75, limits={}, **kwlimits)
Bases: `fipy.viewers.gistViewer.gistViewer._GistViewer`

Displays a contour plot of a 2D *CellVariable* object.

```
>>> from fipy import *
>>> mesh = Grid2D(nx=50, ny=100, dx=0.1, dy=0.01)
>>> x, y = mesh.getCellCenters()
>>> xyVar = CellVariable(mesh=mesh, name="x y", value=x * y)
>>> k = Variable(name="k", value=0.)
>>> viewer = Gist2DViewer(vars=sin(k * xyVar),
...                         limits={'ymin': 0.1, 'ymax': 0.9},
...                         datamin=-0.9, datamax=2.0,
...                         title="Gist2DViewer test")
>>> for kval in range(10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()

>>> from fipy import *
>>> mesh = (Grid2D(nx=5, ny=10, dx=0.1, dy=0.1)
...           + (Tri2D(nx=5, ny=5, dx=0.1, dy=0.1)
...               + ((0.5,), (0.2,))))
>>> x, y = mesh.getCellCenters()
>>> xyVar = CellVariable(mesh=mesh, name="x y", value=x * y)
>>> k = Variable(name="k", value=0.)
>>> viewer = Gist2DViewer(vars=sin(k * xyVar),
...                         limits={'ymin': 0.1, 'ymax': 0.9},
...                         datamin=-0.9, datamax=2.0,
...                         title="Gist2DViewer test")
>>> for kval in range(10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()
```

Creates a *Gist2DViewer*.

Parameters

vars a *CellVariable* object.
title displayed at the top of the *Viewer* window

palette the color scheme to use for the image plot. Default is *heat.gp*. Another choice would be *rainbow.gp*.

grid whether to show the grid lines in the plot.

limits [dict] a (deprecated) alternative to limit keyword arguments

xmin, xmax, ymin, ymax, datamin, datamax displayed range of data. Any limit set to a (default) value of *None* will autoscale.

plot (*filename=None*)

Plot the *CellVariable* as a contour plot.

plotMesh (*filename=None*)

class GistVectorViewer (*vars, title=None, limits={}, **kwlimits*)

Bases: `fipy.viewers.gistViewer.gistViewer._GistViewer`

Displays a vector plot of a 2D rank-1 *CellVariable* or *FaceVariable* object using *gist*.

```
>>> from fipy import *
>>> mesh = Grid2D(nx=50, ny=100, dx=0.1, dy=0.01)
>>> x, y = mesh.getCellCenters()
>>> xyVar = CellVariable(mesh=mesh, name="x y", value=x * y)
>>> k = Variable(name="k", value=0.)
>>> viewer = GistVectorViewer(vars=sin(k * xyVar).getGrad(),
...                             title="GistVectorViewer test")
>>> for kval in range(10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()

>>> viewer = GistVectorViewer(vars=sin(k * xyVar).getFaceGrad(),
...                             title="GistVectorViewer test")
>>> for kval in range(10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()

>>> from fipy import *
>>> mesh = (Grid2D(nx=5, ny=10, dx=0.1, dy=0.1)
...           + (Tri2D(nx=5, ny=5, dx=0.1, dy=0.1,
...                    + ((0.5,), (0.2,))))
...           + ((0.5,)))
>>> x, y = mesh.getCellCenters()
>>> xyVar = CellVariable(mesh=mesh, name="x y", value=x * y)
>>> k = Variable(name="k", value=0.)
>>> viewer = GistVectorViewer(vars=sin(k * xyVar).getGrad(),
...                             title="GistVectorViewer test")
>>> for kval in range(10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()

>>> viewer = GistVectorViewer(vars=sin(k * xyVar).getFaceGrad(),
...                             title="GistVectorViewer test")
>>> for kval in range(10):
...     k.setValue(kval)
```

```
...     viewer.plot()
>>> viewer._promptForOpinion()
```

Creates a *GistVectorViewer*.

Parameters

vars a rank-1 *CellVariable* or *FaceVariable* object.
title displayed at the top of the *Viewer* window
limits [dict] a (deprecated) alternative to limit keyword arguments
xmin, **xmax**, **ymin**, **ymax**, **datamin**, **datamax** displayed range of data. Any limit set to a (default) value of *None* will autoscale.

```
getArray()
plot(filename=None)
```

25.1.2 The colorbar Module

25.1.3 The *gist1DViewer* Module

```
class Gist1DViewer(vars, title=None, xlog=0, ylog=0, style='work.gs', limits={}, **kwlimits)
Bases: fipy.viewers.gistViewer.gistViewer._GistViewer
```

Displays a y vs. x plot of one or more 1D *CellVariable* objects.

```
>>> from fipy import *
>>> mesh = Grid1D(nx=100)
>>> x, = mesh.getCellCenters()
>>> xVar = CellVariable(mesh=mesh, name="x", value=x)
>>> k = Variable(name="k", value=0.)
>>> viewer = Gist1DViewer(vars=(sin(k * xVar), cos(k * xVar / pi)),
...                         limits={'xmin': 10, 'xmax': 90},
...                         datamin=-0.9, datamax=2.0,
...                         title="Gist1DViewer test")
>>> for kval in numerix.arange(0, 0.3, 0.03):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()
```

Creates a *Gist1DViewer*.

Parameters

vars a *CellVariable* or tuple of *CellVariable* objects to plot
title displayed at the top of the *Viewer* window
xlog log scaling of x axis if *True*
ylog log scaling of y axis if *True*
style the Gist stylefile to use.
limits [dict] a (deprecated) alternative to limit keyword arguments
xmin, **xmax**, **datamin**, **datamax** displayed range of data. Any limit set to a (default) value of *None* will autoscale. (*ymin* and *ymax* are synonyms for *datamin* and *datamax*).

plot (*filename=None*)

25.1.4 The `gist2DViewer` Module

class `Gist2DViewer` (*vars*, *title=None*, *palette='heat.gp'*, *grid=True*, *dpi=75*, *limits={}*, *kwlimits*)**

Bases: `fipy.viewers.gistViewer.gistViewer._GistViewer`

Displays a contour plot of a 2D *CellVariable* object.

```
>>> from fipy import *
>>> mesh = Grid2D(nx=50, ny=100, dx=0.1, dy=0.01)
>>> x, y = mesh.getCellCenters()
>>> xyVar = CellVariable(mesh=mesh, name="x y", value=x * y)
>>> k = Variable(name="k", value=0.)
>>> viewer = Gist2DViewer(vars=sin(k * xyVar),
...                         limits={'ymin': 0.1, 'ymax': 0.9},
...                         datamin=-0.9, datamax=2.0,
...                         title="Gist2DViewer test")
>>> for kval in range(10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()

>>> from fipy import *
>>> mesh = (Grid2D(nx=5, ny=10, dx=0.1, dy=0.1)
...           + (Tri2D(nx=5, ny=5, dx=0.1, dy=0.1,
...                     + ((0.5,), (0.2,))))
...           + ((0.5,), (0.2,)))
>>> x, y = mesh.getCellCenters()
>>> xyVar = CellVariable(mesh=mesh, name="x y", value=x * y)
>>> k = Variable(name="k", value=0.)
>>> viewer = Gist2DViewer(vars=sin(k * xyVar),
...                         limits={'ymin': 0.1, 'ymax': 0.9},
...                         datamin=-0.9, datamax=2.0,
...                         title="Gist2DViewer test")
>>> for kval in range(10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()
```

Creates a *Gist2DViewer*.

Parameters

vars a *CellVariable* object.

title displayed at the top of the *Viewer* window

palette the color scheme to use for the image plot. Default is *heat.gp*. Another choice would be *rainbow.gp*.

grid whether to show the grid lines in the plot.

limits [dict] a (deprecated) alternative to limit keyword arguments

xmin, **xmax**, **ymin**, **ymax**, **datamin**, **datamax** displayed range of data. Any limit set to a (default) value of *None* will autoscale.

plot (*filename=None*)

Plot the *CellVariable* as a contour plot.

plotMesh (*filename=None*)

25.1.5 The `gistVectorViewer` Module

class GistVectorViewer (*vars*, *title=None*, *limits={}*, ***kwlimits*)
Bases: `fipy.viewers.gistViewer.gistViewer._GistViewer`

Displays a vector plot of a 2D rank-1 *CellVariable* or *FaceVariable* object using `gist`.

```
>>> from fipy import *
>>> mesh = Grid2D(nx=50, ny=100, dx=0.1, dy=0.01)
>>> x, y = mesh.getCellCenters()
>>> xyVar = CellVariable(mesh=mesh, name="x y", value=x * y)
>>> k = Variable(name="k", value=0.)
>>> viewer = GistVectorViewer(vars=sin(k * xyVar).getGrad(),
...                             title="GistVectorViewer test")
>>> for kval in range(10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()

>>> viewer = GistVectorViewer(vars=sin(k * xyVar).getFaceGrad(),
...                             title="GistVectorViewer test")
>>> for kval in range(10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()

>>> from fipy import *
>>> mesh = (Grid2D(nx=5, ny=10, dx=0.1, dy=0.1)
...           + (Tri2D(nx=5, ny=5, dx=0.1, dy=0.1,
...                    + ((0.5,), (0.2,))))
...           + ((0.5,), (0.2,)))
...
>>> x, y = mesh.getCellCenters()
>>> xyVar = CellVariable(mesh=mesh, name="x y", value=x * y)
>>> k = Variable(name="k", value=0.)
>>> viewer = GistVectorViewer(vars=sin(k * xyVar).getGrad(),
...                             title="GistVectorViewer test")
>>> for kval in range(10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()

>>> viewer = GistVectorViewer(vars=sin(k * xyVar).getFaceGrad(),
...                             title="GistVectorViewer test")
>>> for kval in range(10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()
```

Creates a *GistVectorViewer*.

Parameters

vars a rank-1 *CellVariable* or *FaceVariable* object.

title displayed at the top of the *Viewer* window

```

limits [dict] a (deprecated) alternative to limit keyword arguments
xmin, xmax, ymin, ymax, datamin, datamax displayed range of data. Any limit set to a (default) value of None will autoscale.

getArray()
plot (filename=None)

```

25.1.6 The `gistViewer` Module

25.1.7 The `test` Module

Test numeric implementation of the mesh

25.2 `gnuplotViewer` Package Documentation

This page contains the `gnuplotViewer` Package documentation.

25.2.1 The `gnuplotViewer` Package

GnuplotViewer (*vars, title=None, limits={}, **kwlimits*)

Generic function for creating a *GnuplotViewer*.

The *GnuplotViewer* factory will search the module tree and return an instance of the first *GnuplotViewer* it finds of the correct dimension.

Parameters

vars a *CellVariable* or tuple of *CellVariable* objects to plot

title displayed at the top of the Viewer window

limits [dict] a (deprecated) alternative to limit keyword arguments

xmin, xmax, ymin, ymax, datamin, datamax 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.

class Gnuplot1DViewer (*vars, title=None, **kwlimits*)

Bases: `fipy.viewers.gnuplotViewer.gnuplotViewer._GnuplotViewer`

Displays a y vs. x plot of one or more 1D *CellVariable* objects.

The *Gnuplot1DViewer* plots a 1D *CellVariable* using a front end python wrapper available to download ([Gnuplot.py](#)).

```

>>> from fipy import *
>>> mesh = Grid1D(nx=100)
>>> x, = mesh.getCellCenters()
>>> xVar = CellVariable(mesh=mesh, name="x", value=x)
>>> k = Variable(name="k", value=0.)
>>> viewer = Gnuplot1DViewer(vars=(sin(k * xVar), cos(k * xVar / pi)),
...                           limits={'xmin': 10, 'xmax': 90},
...                           datamin=-0.9, datamax=2.0,
...                           title="Gnuplot1DViewer test")
>>> for kval in numerix.arange(0, 0.3, 0.03):

```

```
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()
```

Different style script [demos](#) are available at the [Gnuplot](#) site.

Note: *Gnuplot1DViewer* requires [Gnuplot](#) version 4.0.

The *_GnuplotViewer* should not be called directly only *Gnuplot1DViewer* and *Gnuplot2DViewer* should be called.

Parameters

vars a *CellVariable* or tuple of *CellVariable* objects to plot

title displayed at the top of the *Viewer* window

xmin, xmax, ymin, ymax, datamin, datamax 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.

```
class Gnuplot2DViewer(vars, title=None, limits={}, **kwlimits)
```

Bases: `fipy.viewers.gnuplotViewer.gnuplotViewer._GnuplotViewer`

Displays a contour plot of a 2D *CellVariable* object.

The *Gnuplot2DViewer* plots a 2D *CellVariable* using a front end python wrapper available to download ([Gnuplot.py](#)).

```
>>> from fipy import *
>>> mesh = Grid2D(nx=50, ny=100, dx=0.1, dy=0.01)
>>> x, y = mesh.getCellCenters()
>>> xyVar = CellVariable(mesh=mesh, name="x y", value=x * y)
>>> k = Variable(name="k", value=0.)
>>> viewer = Gnuplot2DViewer(vars=sin(k * xyVar),
...                             limits={'ymin': 0.1, 'ymax': 0.9},
...                             datamin=-0.9, datamax=2.0,
...                             title="Gnuplot2DViewer test")
>>> for kval in range(10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()

>>> from fipy import *
>>> mesh = (Grid2D(nx=5, ny=10, dx=0.1, dy=0.1)
...           + (Tri2D(nx=5, ny=5, dx=0.1, dy=0.1)
...               + ((0.5,), (0.2,))))
>>> x, y = mesh.getCellCenters()
>>> xyVar = CellVariable(mesh=mesh, name="x y", value=x * y)
>>> k = Variable(name="k", value=0.)
>>> viewer = Gnuplot2DViewer(vars=sin(k * xyVar),
...                             limits={'ymin': 0.1, 'ymax': 0.9},
...                             datamin=-0.9, datamax=2.0,
...                             title="Gnuplot2DViewer test")
>>> for kval in range(10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()
```

Different style script [demos](#) are available at the [Gnuplot](#) site.

Note: *Gnuplot2DViewer* requires [Gnuplot](#) version 4.0.

Creates a *Gnuplot2DViewer*.

Parameters

vars a *CellVariable* object.
title displayed at the top of the *Viewer* window
limits [dict] a (deprecated) alternative to limit keyword arguments
xmin, xmax, ymin, ymax, datamin, datamax displayed range of data. Any limit set to a (default) value of *None* will autoscale.

25.2.2 The `gnuplot1DViewer` Module

```
class Gnuplot1DViewer(vars, title=None, **kwlimits)
```

Bases: `fipy.viewers.gnuplotViewer.gnuplotViewer._GnuplotViewer`

Displays a y vs. x plot of one or more 1D *CellVariable* objects.

The *Gnuplot1DViewer* plots a 1D *CellVariable* using a front end python wrapper available to download ([Gnuplot.py](#)).

```
>>> from fipy import *
>>> mesh = Grid1D(nx=100)
>>> x, = mesh.getCellCenters()
>>> xVar = CellVariable(mesh=mesh, name="x", value=x)
>>> k = Variable(name="k", value=0.)
>>> viewer = Gnuplot1DViewer(vars=(sin(k * xVar), cos(k * xVar / pi)),
...                           limits={'xmin': 10, 'xmax': 90},
...                           datamin=-0.9, datamax=2.0,
...                           title="Gnuplot1DViewer test")
>>> for kval in numerix.arange(0, 0.3, 0.03):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()
```

Different style script [demos](#) are available at the [Gnuplot](#) site.

Note: *Gnuplot1DViewer* requires [Gnuplot](#) version 4.0.

The *_GnuplotViewer* should not be called directly only *Gnuplot1DViewer* and *Gnuplot2DViewer* should be called.

Parameters

vars a *CellVariable* or tuple of *CellVariable* objects to plot
title displayed at the top of the *Viewer* window
xmin, xmax, ymin, ymax, datamin, datamax 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.

25.2.3 The `Gnuplot2DViewer` Module

```
class Gnuplot2DViewer(vars, title=None, limits={}, **kwlimits)
    Bases: fipy.viewers.gnuplotViewer.gnuplotViewer._GnuplotViewer
```

Displays a contour plot of a 2D *CellVariable* object.

The *Gnuplot2DViewer* plots a 2D *CellVariable* using a front end python wrapper available to download ([Gnuplot.py](#)).

```
>>> from fipy import *
>>> mesh = Grid2D(nx=50, ny=100, dx=0.1, dy=0.01)
>>> x, y = mesh.getCellCenters()
>>> xyVar = CellVariable(mesh=mesh, name="x y", value=x * y)
>>> k = Variable(name="k", value=0.)
>>> viewer = Gnuplot2DViewer(vars=sin(k * xyVar),
...                             limits={'ymin': 0.1, 'ymax': 0.9},
...                             datamin=-0.9, datamax=2.0,
...                             title="Gnuplot2DViewer test")
>>> for kval in range(10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()

>>> from fipy import *
>>> mesh = (Grid2D(nx=5, ny=10, dx=0.1, dy=0.1)
...           + (Tri2D(nx=5, ny=5, dx=0.1, dy=0.1)
...               + ((0.5,), (0.2,))))
>>> x, y = mesh.getCellCenters()
>>> xyVar = CellVariable(mesh=mesh, name="x y", value=x * y)
>>> k = Variable(name="k", value=0.)
>>> viewer = Gnuplot2DViewer(vars=sin(k * xyVar),
...                             limits={'ymin': 0.1, 'ymax': 0.9},
...                             datamin=-0.9, datamax=2.0,
...                             title="Gnuplot2DViewer test")
>>> for kval in range(10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()
```

Different style script [demos](#) are available at the [Gnuplot](#) site.

Note: *Gnuplot2DViewer* requires [Gnuplot](#) version 4.0.

Creates a *Gnuplot2DViewer*.

Parameters

vars a *CellVariable* object.

title displayed at the top of the *Viewer* window

limits [dict] a (deprecated) alternative to limit keyword arguments

xmin, **xmax**, **ymin**, **ymax**, **datamin**, **datamax** displayed range of data. Any limit set to a (default) value of *None* will autoscale.

25.2.4 The `gnuplotViewer` Module

25.2.5 The `test` Module

Test numeric implementation of the mesh

25.3 `matplotlibViewer` Package Documentation

This page contains the `matplotlibViewer` Package documentation.

25.3.1 The `matplotlibViewer` Package

`MatplotlibViewer` (`vars`, `title=None`, `limits={}`, `**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.

Parameters

`vars` a *CellVariable* or tuple of *CellVariable* objects to plot

`title` displayed at the top of the *Viewer* window

`limits` [dict] a (deprecated) alternative to limit keyword arguments

`xmin`, `xmax`, `ymin`, `ymax`, `datamin`, `datamax` 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.

`class Matplotlib1DViewer` (`vars`, `title=None`, `xlog=False`, `ylog=False`, `limits={}`, `**kwlimits`)

Bases: `fipy.viewers.matplotlibViewer.matplotlibViewer._MatplotlibViewer`

Displays a y vs. x plot of one or more 1D *CellVariable* objects using `Matplotlib`.

```
>>> from fipy import *
>>> mesh = Grid1D(nx=100)
>>> x, = mesh.getCellCenters()
>>> xVar = CellVariable(mesh=mesh, name="x", value=x)
>>> k = Variable(name="k", value=0.)
>>> viewer = Matplotlib1DViewer(vars=(sin(k * xVar), cos(k * xVar / pi)),
...                               limits={'xmin': 10, 'xmax': 90},
...                               datamin=-0.9, datamax=2.0,
...                               title="Matplotlib1DViewer test")
>>> for kval in numerix.arange(0, 0.3, 0.03):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()
```

Parameters

`vars` a *CellVariable* or tuple of *CellVariable* objects to plot

`title` displayed at the top of the *Viewer* window

`xlog` log scaling of x axis if `True`

`ylog` log scaling of y axis if `True`

limits [dict] a (deprecated) alternative to limit keyword arguments
xmin, xmax, datamin, datamax displayed range of data. Any limit set to a (default) value of *None* will autoscale. (*ymin* and *ymax* are synonyms for *datamin* and *datamax*).

class Matplotlib2DGridViewer (*vars*, *title=None*, *limits={}*, *cmap=None*, ***kwlimits*)
Bases: fipy.viewers.matplotlibViewer.matplotlibViewer._MatplotlibViewer

Displays an image plot of a 2D *CellVariable* object using [Matplotlib](#).

```
>>> from fipy import *
>>> mesh = Grid2D(nx=50, ny=100, dx=0.1, dy=0.01)
>>> x, y = mesh.getCellCenters()
>>> xyVar = CellVariable(mesh=mesh, name="x y", value=x * y)
>>> k = Variable(name="k", value=0.)
>>> viewer = Matplotlib2DGridViewer(vars=sin(k * xyVar),
...                                 limits={'ymin': 0.1, 'ymax': 0.9},
...                                 datamin=-0.9, datamax=2.0,
...                                 title="Matplotlib2DGridViewer test")
>>> for kval in range(10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()
```

Creates a *Matplotlib2DGridViewer*.

Parameters

vars A *CellVariable* object.
title displayed at the top of the *Viewer* window
limits [dict] a (deprecated) alternative to limit keyword arguments
cmap The colormap. Defaults to *pylab.cm.jet*
xmin, xmax, ymin, ymax, datamin, datamax displayed range of data. Any limit set to a (default) value of *None* will autoscale.

class Matplotlib2DGridContourViewer (*vars*, *title=None*, *limits={}*, ***kwlimits*)
Bases: fipy.viewers.matplotlibViewer.matplotlibViewer._MatplotlibViewer

Displays a contour plot of a 2D *CellVariable* object.

The *Matplotlib2DGridContourViewer* plots a 2D *CellVariable* using [Matplotlib](#).

```
>>> from fipy import *
>>> mesh = Grid2D(nx=50, ny=100, dx=0.1, dy=0.01)
>>> x, y = mesh.getCellCenters()
>>> xyVar = CellVariable(mesh=mesh, name="x y", value=x * y)
>>> k = Variable(name="k", value=0.)
>>> viewer = Matplotlib2DGridContourViewer(vars=sin(k * xyVar),
...                                         limits={'ymin': 0.1, 'ymax': 0.9},
...                                         datamin=-0.9, datamax=2.0,
...                                         title="Matplotlib2DGridContourViewer test")
>>> for kval in range(10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()
```

Creates a *Matplotlib2DViewer*.

Parameters

vars a *CellVariable* object.
title displayed at the top of the *Viewer* window
limits [dict] a (deprecated) alternative to limit keyword arguments
xmin, xmax, ymin, ymax, datamin, datamax displayed range of data. Any limit set to a (default) value of *None* will autoscale.

```
class Matplotlib2DViewer(vars, title=None, limits={}, cmap=None, **kwlimits)
```

Bases: `fipy.viewers.matplotlibViewer.matplotlibViewer._MatplotlibViewer`

Displays a contour plot of a 2D *CellVariable* object.

The *Matplotlib2DViewer* plots a 2D *CellVariable* using *Matplotlib*.

```
>>> from fipy import *
>>> mesh = (Grid2D(nx=5, ny=10, dx=0.1, dy=0.1)
...         + (Tri2D(nx=5, ny=5, dx=0.1, dy=0.1)
...             + ((0.5,), (0.2,))))
>>> x, y = mesh.getCellCenters()
>>> xyVar = CellVariable(mesh=mesh, name="x y", value=x * y)
>>> k = Variable(name="k", value=0.)
>>> viewer = Matplotlib2DViewer(vars=sin(k * xyVar),
...                               limits={'ymin': 0.1, 'ymax': 0.9},
...                               datamin=-0.9, datamax=2.0,
...                               title="Matplotlib2DViewer test")
>>> for kval in range(10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()
```

Creates a *Matplotlib2DViewer*.

Parameters

vars a *CellVariable* object.
title displayed at the top of the *Viewer* window
limits [dict] a (deprecated) alternative to limit keyword arguments
cmap the colormap. Defaults to *pylab.cm.jet*
xmin, xmax, ymin, ymax, datamin, datamax displayed range of data. Any limit set to a (default) value of *None* will autoscale.

```
class MatplotlibVectorViewer(vars, title=None, scale=None, sparsity=None, limits={}, **kwlimits)
```

Bases: `fipy.viewers.matplotlibViewer.matplotlibViewer._MatplotlibViewer`

Displays a vector plot of a 2D rank-1 *CellVariable* or *FaceVariable* object using *Matplotlib*

```
>>> from fipy import *
>>> mesh = Grid2D(nx=50, ny=100, dx=0.1, dy=0.01)
>>> x, y = mesh.getCellCenters()
>>> xyVar = CellVariable(mesh=mesh, name="x y", value=x * y)
>>> k = Variable(name="k", value=0.)
>>> viewer = MatplotlibVectorViewer(vars=sin(k * xyVar).getGrad(),
...                                   title="MatplotlibVectorViewer test")
>>> for kval in range(10):
...     k.setValue(kval)
```

```
...     viewer.plot()
>>> viewer._promptForOpinion()

>>> viewer = MatplotlibVectorViewer(vars=sin(k * xyVar).getFaceGrad(),
...                                 title="MatplotlibVectorViewer test")
>>> for kval in range(10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()

>>> for sparsity in arange(5000, 0, -500):
...     viewer.quiver(sparsity=sparsity)
...     viewer.plot()
>>> viewer._promptForOpinion()

>>> from fipy import *
>>> mesh = (Grid2D(nx=5, ny=10, dx=0.1, dy=0.1)
...           + (Tri2D(nx=5, ny=5, dx=0.1, dy=0.1,
...                     + ((0.5,), (0.2,))))
...           + ((0.5,), (0.2,)))
>>> x, y = mesh.getCellCenters()
>>> xyVar = CellVariable(mesh=mesh, name="x y", value=x * y)
>>> k = Variable(name="k", value=0.)
>>> viewer = MatplotlibVectorViewer(vars=sin(k * xyVar).getGrad(),
...                                 title="MatplotlibVectorViewer test")
>>> for kval in range(10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()

>>> viewer = MatplotlibVectorViewer(vars=sin(k * xyVar).getFaceGrad(),
...                                 title="MatplotlibVectorViewer test")
>>> for kval in range(10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()
```

Creates a *Matplotlib2DViewer*.

Parameters

- vars** a rank-1 *CellVariable* or *FaceVariable* object.
- title** displayed at the top of the *Viewer* window
- scale** if not *None*, scale all arrow lengths by this value
- sparsity** if not *None*, then this number of arrows will be randomly chosen (weighted by the cell volume or face area)
- limits** [dict] a (deprecated) alternative to limit keyword arguments
- xmin, xmax, ymin, ymax, datamin, datamax** displayed range of data. Any limit set to a (default) value of *None* will autoscale.
- quiver** (*sparsity=None*, *scale=None*)

25.3.2 The `matplotlib1DViewer` Module

```
class Matplotlib1DViewer(vars, title=None, xlog=False, ylog=False, limits={}, **kwlimits)
Bases: fipy.viewers.matplotlibViewer.matplotlibViewer._MatplotlibViewer

Displays a y vs. x plot of one or more 1D CellVariable objects using Matplotlib.
```

```
>>> from fipy import *
>>> mesh = Grid1D(nx=100)
>>> x, = mesh.getCellCenters()
>>> xVar = CellVariable(mesh=mesh, name="x", value=x)
>>> k = Variable(name="k", value=0.)
>>> viewer = Matplotlib1DViewer(vars=(sin(k * xVar), cos(k * xVar / pi)),
...                               limits={'xmin': 10, 'xmax': 90},
...                               datamin=-0.9, datamax=2.0,
...                               title="Matplotlib1DViewer test")
>>> for kval in numerix.arange(0, 0.3, 0.03):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()
```

Parameters

vars a *CellVariable* or tuple of *CellVariable* objects to plot
title displayed at the top of the *Viewer* window
xlog log scaling of x axis if *True*
ylog log scaling of y axis if *True*
limits [dict] a (deprecated) alternative to limit keyword arguments
xmin, xmax, datamin, datamax displayed range of data. Any limit set to a (default) value of *None* will autoscale. (*ymin* and *ymax* are synonyms for *datamin* and *datamax*).

25.3.3 The `matplotlib2DGridContourViewer` Module

```
class Matplotlib2DGridContourViewer(vars, title=None, limits={}, **kwlimits)
Bases: fipy.viewers.matplotlibViewer.matplotlibViewer._MatplotlibViewer

Displays a contour plot of a 2D CellVariable object.
```

The *Matplotlib2DGridContourViewer* plots a 2D *CellVariable* using `Matplotlib`.

```
>>> from fipy import *
>>> mesh = Grid2D(nx=50, ny=100, dx=0.1, dy=0.01)
>>> x, y = mesh.getCellCenters()
>>> xyVar = CellVariable(mesh=mesh, name="x y", value=x * y)
>>> k = Variable(name="k", value=0.)
>>> viewer = Matplotlib2DGridContourViewer(vars=sin(k * xyVar),
...                                         limits={'ymin': 0.1, 'ymax': 0.9},
...                                         datamin=-0.9, datamax=2.0,
...                                         title="Matplotlib2DGridContourViewer test")
>>> for kval in range(10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()
```

Creates a *Matplotlib2DViewer*.

Parameters

vars a *CellVariable* object.
title displayed at the top of the *Viewer* window
limits [dict] a (deprecated) alternative to limit keyword arguments
xmin, xmax, ymin, ymax, datamin, datamax displayed range of data. Any limit set to a (default) value of *None* will autoscale.

25.3.4 The `matplotlib2DGridViewer` Module

class `Matplotlib2DGridViewer` (*vars*, *title*=*None*, *limits*={}, *cmap*=*None*, ***kwlimits*)

Bases: `fipy.viewers.matplotlibViewer.matplotlibViewer._MatplotlibViewer`

Displays an image plot of a 2D *CellVariable* object using `Matplotlib`.

```
>>> from fipy import *
>>> mesh = Grid2D(nx=50, ny=100, dx=0.1, dy=0.01)
>>> x, y = mesh.getCellCenters()
>>> xyVar = CellVariable(mesh=mesh, name="x y", value=x * y)
>>> k = Variable(name="k", value=0.)
>>> viewer = Matplotlib2DGridViewer(vars=sin(k * xyVar),
...                                 limits={'ymin': 0.1, 'ymax': 0.9},
...                                 datamin=-0.9, datamax=2.0,
...                                 title="Matplotlib2DGridViewer test")
>>> for kval in range(10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()
```

Creates a *Matplotlib2DGridViewer*.

Parameters

vars A *CellVariable* object.
title displayed at the top of the *Viewer* window
limits [dict] a (deprecated) alternative to limit keyword arguments
cmap The colormap. Defaults to `pylab.cm.jet`
xmin, xmax, ymin, ymax, datamin, datamax displayed range of data. Any limit set to a (default) value of *None* will autoscale.

25.3.5 The `matplotlib2DViewer` Module

class `Matplotlib2DViewer` (*vars*, *title*=*None*, *limits*={}, *cmap*=*None*, ***kwlimits*)

Bases: `fipy.viewers.matplotlibViewer.matplotlibViewer._MatplotlibViewer`

Displays a contour plot of a 2D *CellVariable* object.

The *Matplotlib2DViewer* plots a 2D *CellVariable* using `Matplotlib`.

```

>>> from fipy import *
>>> mesh = (Grid2D(nx=5, ny=10, dx=0.1, dy=0.1)
...         + (Tri2D(nx=5, ny=5, dx=0.1, dy=0.1)
...             + ((0.5,), (0.2,))))
>>> x, y = mesh.getCellCenters()
>>> xyVar = CellVariable(mesh=mesh, name="x y", value=x * y)
>>> k = Variable(name="k", value=0.)
>>> viewer = Matplotlib2DViewer(vars=sin(k * xyVar),
...                               limits={'ymin': 0.1, 'ymax': 0.9},
...                               datamin=-0.9, datamax=2.0,
...                               title="Matplotlib2DViewer test")
>>> for kval in range(10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()

```

Creates a *Matplotlib2DViewer*.

Parameters

vars a *CellVariable* object.

title displayed at the top of the *Viewer* window

limits [dict] a (deprecated) alternative to limit keyword arguments

cmap the colormap. Defaults to *pylab.cm.jet*

xmin, xmax, ymin, ymax, datamin, datamax displayed range of data. Any limit set to a (default) value of *None* will autoscale.

25.3.6 The `matplotlibSparseMatrixViewer` Module

```

class MatplotlibSparseMatrixViewer(title='Sparsity')

    plot(matrix, RHSvector, log='auto')

class SignedLogFormatter(base=10.0, labelOnlyBase=True, threshold=0.0)
    Bases: matplotlib.ticker.LogFormatter
    Format values for log axis;
    if attribute decadeOnly is True, only the decades will be labelled.
    base is used to locate the decade tick, which will be the only one to be labeled if labelOnlyBase is False
    pprint_val(x, d)

class SignedLogLocator(base=10.0, subs=[1.0], threshold=0.0)
    Bases: matplotlib.ticker.LogLocator
    Determine the tick locations for "log" axes that express both positive and negative values
    place ticks on the location=base**i*subs[j]
    autoscale()
        Try to choose the view limits intelligently

```

25.3.7 The `matplotlibVectorViewer` Module

```
class MatplotlibVectorViewer(vars, title=None, scale=None, sparsity=None, limits={}, **kwlimits)
    Bases: fipy.viewers.matplotlibViewer.matplotlibViewer._MatplotlibViewer
```

Displays a vector plot of a 2D rank-1 *CellVariable* or *FaceVariable* object using Matplotlib

```
>>> from fipy import *
>>> mesh = Grid2D(nx=50, ny=100, dx=0.1, dy=0.01)
>>> x, y = mesh.getCellCenters()
>>> xyVar = CellVariable(mesh=mesh, name="x y", value=x * y)
>>> k = Variable(name="k", value=0.)
>>> viewer = MatplotlibVectorViewer(vars=sin(k * xyVar).getGrad(),
...                                 title="MatplotlibVectorViewer test")
>>> for kval in range(10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()

>>> viewer = MatplotlibVectorViewer(vars=sin(k * xyVar).getFaceGrad(),
...                                 title="MatplotlibVectorViewer test")
>>> for kval in range(10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()

>>> for sparsity in arange(5000, 0, -500):
...     viewer.quiver(sparsity=sparsity)
...     viewer.plot()
>>> viewer._promptForOpinion()

>>> from fipy import *
>>> mesh = (Grid2D(nx=5, ny=10, dx=0.1, dy=0.1)
...           + (Tri2D(nx=5, ny=5, dx=0.1, dy=0.1,
...                    + ((0.5,), (0.2,))))
...           + ((0.5,), (0.2,)))
...
>>> x, y = mesh.getCellCenters()
>>> xyVar = CellVariable(mesh=mesh, name="x y", value=x * y)
>>> k = Variable(name="k", value=0.)
>>> viewer = MatplotlibVectorViewer(vars=sin(k * xyVar).getGrad(),
...                                 title="MatplotlibVectorViewer test")
>>> for kval in range(10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()

>>> viewer = MatplotlibVectorViewer(vars=sin(k * xyVar).getFaceGrad(),
...                                 title="MatplotlibVectorViewer test")
>>> for kval in range(10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()
```

Creates a *Matplotlib2DViewer*.

Parameters

vars a rank-1 *CellVariable* or *FaceVariable* object.

title displayed at the top of the *Viewer* window
scale if not *None*, scale all arrow lengths by this value
sparsity if not *None*, then this number of arrows will be randomly chosen (weighted by the cell volume or face area)
limits [dict] a (deprecated) alternative to limit keyword arguments
xmin, xmax, ymin, ymax, datamin, datamax displayed range of data. Any limit set to a (default) value of *None* will autoscale.
quiver (*sparsity=None, scale=None*)

25.3.8 The `matplotlibViewer` Module

25.3.9 The `test` Module

Test numeric implementation of the mesh

25.4 mayaviViewer Package Documentation

This page contains the mayaviViewer Package documentation.

25.4.1 The `mayaviClient` Module

class MayaviClient (vars, title=None, daemon_file=None, fps=1.0, **kwlimits)
Bases: `fipy.viewers.viewer._Viewer`

The *MayaviClient* uses the **Mayavi** python plotting package.

Create a *MayaviClient*.

Parameters

vars a *CellVariable* or tuple of *CellVariable* objects to plot
title displayed at the top of the *Viewer* window
xmin, xmax, ymin, ymax, zmin, zmax, datamin, datamax displayed range of data. A 1D *Viewer* will only use *xmin* and *xmax*, a 2D viewer will also use *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 the path to the script to run the separate MayaVi viewer process. Defaults to “`fipy/viewers/mayaviViewer/mayaviDaemon.py`”
fps frames per second to attempt to display
plot (*filename=None*)

25.4.2 The `mayaviDaemon` Module

A simple script that polls a data file for changes and then updates the mayavi pipeline automatically.

This script is based heavily on the `poll_file.py` example in the mayavi distribution.

This script is to be run like so:

```
$ mayavi2 -x mayaviDaemon.py <options>
```

Or:

```
$ python mayaviDaemon.py <options>
```

Run:

```
$ python mayaviDaemon.py --help
```

to see available options.

class MayaviDaemon()

Bases: enthought.mayavi.plugins.app.Mayavi

Given a file name and a mayavi2 data reader object, this class polls the file for any changes and automatically updates the mayavi pipeline.

clip_data(src)

parse_command_line(argv)

Parse command line options.

Parameters - argv : list of strings

The list of command line arguments.

poll_file()

run()

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.

main(argv=None)

Simple helper to start up the mayavi application. This returns the running application.

25.4.3 The test Module

Test numeric implementation of the mesh

25.5 vtkViewer Package Documentation

This page contains the vtkViewer Package documentation.

25.5.1 The `vtkViewer` Package

VTKViewer (*vars*, *title*=*None*, *limits*={}, ***kwlimits*)

Generic function for creating a *MatplotlibViewer*.

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 a *_MeshVariable* or tuple of *_MeshVariable* objects to plot

title displayed at the top of the *Viewer* window

limits [dict] a (deprecated) alternative to limit keyword arguments

xmin, **xmax**, **ymin**, **ymax**, **zmin**, **zmax**, **datamin**, **datamax** displayed range of data. A 1D *Viewer* will only use *xmin* and *xmax*, a 2D viewer will also use *ymin* and *ymax*, and so on. All viewers will use *datamin* and *datamax*. Any limit set to a (default) value of *None* will autoscale.

class VTKCellViewer (*vars*, *title*=*None*, *limits*={}, ***kwlimits*)

Bases: `fipy.viewers.vtkViewer.vtkViewer._VTKViewer`

Renders *CellVariable* data in VTK format

Creates a VTKViewer

Parameters

vars a *_MeshVariable* or a tuple of them

title displayed at the top of the *Viewer* window

limits [dict] a (deprecated) alternative to limit keyword arguments

xmin, **xmax**, **ymin**, **ymax**, **zmin**, **zmax**, **datamin**, **datamax** displayed range of data. Any limit set to a (default) value of *None* will autoscale.

class VTKFaceViewer (*vars*, *title*=*None*, *limits*={}, ***kwlimits*)

Bases: `fipy.viewers.vtkViewer.vtkViewer._VTKViewer`

Renders *_MeshVariable* data in VTK format

Creates a VTKViewer

Parameters

vars a *_MeshVariable* or a tuple of them

title displayed at the top of the *Viewer* window

limits [dict] a (deprecated) alternative to limit keyword arguments

xmin, **xmax**, **ymin**, **ymax**, **zmin**, **zmax**, **datamin**, **datamax** displayed range of data. Any limit set to a (default) value of *None* will autoscale.

25.5.2 The `test` Module

Test numeric implementation of the mesh

25.5.3 The `vtkCellViewer` Module

```
class VTKCellViewer (vars, title=None, limits={}, **kwlimits)
    Bases: fipy.viewers.vtkViewer.vtkViewer._VTKViewer
    Renders CellVariable data in VTK format
    Creates a VTKViewer

    Parameters
        vars a _MeshVariable or a tuple of them
        title displayed at the top of the Viewer window
        limits [dict] a (deprecated) alternative to limit keyword arguments
        xmin, xmax, ymin, ymax, zmin, zmax, datamin, datamax displayed range of data. Any limit
            set to a (default) value of None will autoscale.
```

25.5.4 The `vtkFaceViewer` Module

```
class VTKFaceViewer (vars, title=None, limits={}, **kwlimits)
    Bases: fipy.viewers.vtkViewer.vtkViewer._VTKViewer
    Renders _MeshVariable data in VTK format
    Creates a VTKViewer

    Parameters
        vars a _MeshVariable or a tuple of them
        title displayed at the top of the Viewer window
        limits [dict] a (deprecated) alternative to limit keyword arguments
        xmin, xmax, ymin, ymax, zmin, zmax, datamin, datamax displayed range of data. Any limit
            set to a (default) value of None will autoscale.
```

25.5.5 The `vtkViewer` Module

25.6 The `viewers` Package

```
exception MeshDimensionError
    Bases: exceptions.IndexError

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 ‘gist’, ‘gnuplot’, ‘matplotlib’, ‘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 a *CellVariable* or tuple of *CellVariable* objects to plot
title displayed at the top of the *Viewer* window
limits [dict] a (deprecated) alternative to limit keyword arguments
FIPY_VIEWER a specific viewer to attempt (possibly multiple times for multiple variables)
xmin, xmax, ymin, ymax, zmin, zmax, datamin, datamax displayed range of data. A 1D *Viewer* will only use *xmin* and *xmax*, a 2D viewer will also use *ymin* and *ymax*, and so on. All viewers will use *datamin* and *datamax*. Any limit set to a (default) value of *None* will autoscale.
make (*args, **kwargs)
A deprecated synonym for *Viewer*

25.7 The `multiViewer` Module

```
class MultiViewer(viewers)
Bases: fipy.viewers.viewer._Viewer
```

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] the viewers to bind together
getViewers()
plot()
setLimits(limits={}, **kwlimits)

25.8 The `test` Module

Test implementation of the viewers

25.9 The `testinteractive` Module

Interactively test the viewers

25.10 The `tsvViewer` Module

```
class TSVViewer (vars, title=None, limits={}, **kwlimits)
Bases: fipy.viewers.viewer._Viewer
“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 a *CellVariable*, a *FaceVariable*, a tuple of *CellVariable* objects, or a tuple of *FaceVariable* objects to plot
title displayed at the top of the *Viewer* window
limits [dict] a (deprecated) alternative to limit keyword arguments
xmin, xmax, ymin, ymax, zmin, zmax, datamin, datamax 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.grid1D 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.getGrad())).plot()
x      var      var_gauss_grad_x
0.2    0        2.5
0.6    2        6.25
1      5        3.75

>>> from fipy.meshes.grid2D 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.getGrad())).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 If not *None*, the name of a file to save the image into.

25.11 The viewer Module

make (*vars*, *title=None*, *limits=None*)

Bibliography

- [BoettingerReview:2002] Boettinger, W J, et al. 2002. Phase-field simulation of solidification. *Annual Review of Materials Research* 32, 163-194.
- [CahnHilliardII] Cahn, John W. 1959. Free energy of a nonuniform system. II. Thermodynamic basis. *jcp* 30, 1121-1124.
- [CahnHilliardI] Cahn, John W, and John E Hilliard. 1958. Free energy of a nonuniform system. I. Interfacial free energy. *jcp* 28, 258-267.
- [CahnHilliardIII] Cahn, John W, and John E Hilliard. 1959. Free energy of a nonuniform system. III. Nucleation in a two-component incompressible fluid. *jcp* 31, 688-699.
- [ChenReview:2002] Chen, L Q. 2002. Phase-field mdoels for microstructure evolution. *Annual Review of Materials Research* 32, 113-140.
- [SubversionRedBean] Collins-Sussman, Ben, Brian W Fitzpatrick, and C Michael Pilato. (2004) *Version Control with Subversion.* : O'Reilly Media.
- [croftphd] Croft, T N. 1998. Unstructured Mesh - Finite Volume Algorithms for Swirling, Turbulent Reacting Flows.
- [NIST:damascene:2001] Josell, D, et al. 2001. Superconformal Electrodeposition in Submicron Features. *prl* 87, 016102.
- [Mattiussi:1997] Mattiussi, C. 1997. An analysis of finite volume, finite element, and finite difference methods using some concepts from algebraic topology. *Journal of Computational Physics* 133, 289-309.
- [McFaddenReview:2002] McFadden, G B. 2002. Phase-field models of solidification. *Contemporary Mathematics* 306, 107-145.
- [moffatInterface:2004] Moffat, T P, D Wheeler, and D Josell. 2004. Superfilling and the Curvature Enhanced Accelerator Coverage Mechanism. *The Electrochemical Society, Interface* 13, 46-52.
- [patanker] Patanker, S V. (1980) *Numerical Heat Transfer and Fluid Flow.* : Taylor and Francis.
- [DiveIntoPython] Pilgrim, Mark. (2004) *Dive Into Python.* : Apress.
- [PythonTutorial] van Rossum, Guido. . Python Tutorial.
- [versteegMalalasekera] Versteeg, H K, and W Malalasekera. (1995) *An Introduction to Computational Fluid Dynamics.* : Longman Scientific and Technical.
- [InstallingPythonModules] Ward, Greg. . Installing Python Modules.
- [WarrenPolycrystal] Warren, James A, et al. 2003. Extending Phase Field Models of Solidification to Polycrystalline Materials. *Acta Materialia* 51, 6035-6058.

Module Index

E

examples.cahnHilliard.mesh2D, 141
examples.cahnHilliard.sphere, 142
examples.convection.exponential1D.mesh1D, 79
examples.convection.exponential1DSource.mesh1D, 80
examples.convection.robin, 82
examples.convection.source, 83
examples.diffusion.anisotropy, 77
examples.diffusion.circle, 66
examples.diffusion.electrostatics, 71
examples.diffusion.mesh1D, 51
examples.diffusion.mesh20x20, 64
examples.diffusion.nthOrder.input4thOrder, 75
examples.flow.stokesCavity, 147
examples.levelSet.advection.circle, 122
examples.levelSet.advection.mesh1D, 121
examples.levelSet.distanceFunction.circle, 120
examples.levelSet.distanceFunction.mesh1D, 119
examples.levelSet.electroChem.gold, 128
examples.levelSet.electroChem.howToWrite, 133
examples.levelSet.electroChem.leveler, 130
examples.levelSet.electroChem.simpleTrench, 125
examples.phase.anisotropy, 107
examples.phase.binary, 93
examples.phase.impingement.mesh20x20, 114
examples.phase.impingement.mesh40x1, 111
examples.phase.quaternary, 102
examples.phase.simple, 85
examples.updating.update0_1to1_0, 151
examples.updating.update1_0to2_0, 155

F

fipy.boundaryConditions.boundaryCondition, 165
fipy.boundaryConditions.fixedFlux, 165
fipy.boundaryConditions.fixedValue, 166
fipy.boundaryConditions.nthOrderBoundaryCondition, 166
fipy.boundaryConditions.test, 167
fipy.meshes.common.mesh, 169
fipy.meshes.cylindricalGrid1D, 187
fipy.meshes.cylindricalGrid2D, 187
fipy.meshes.grid1D, 188
fipy.meshes.grid2D, 188
fipy.meshes.grid3D, 188
fipy.meshes.numMesh.cell, 171
fipy.meshes.numMesh.cylindricalGrid1D, 171
fipy.meshes.numMesh.cylindricalGrid2D, 172
fipy.meshes.numMesh.cylindricalUniformGrid1D, 172
fipy.meshes.numMesh.cylindricalUniformGrid2D, 173
fipy.meshes.numMesh.face, 173
fipy.meshes.numMesh.gmshExport, 173
fipy.meshes.numMesh.gmshImport, 173
fipy.meshes.numMesh.grid1D, 176
fipy.meshes.numMesh.grid2D, 177
fipy.meshes.numMesh.grid3D, 177
fipy.meshes.numMesh.mesh, 178
fipy.meshes.numMesh.mesh1D, 178
fipy.meshes.numMesh.mesh2D, 178
fipy.meshes.numMesh.periodicGrid1D, 179
fipy.meshes.numMesh.periodicGrid2D, 180
fipy.meshes.numMesh.skewedGrid2D, 181
fipy.meshes.numMesh.test, 181
fipy.meshes.numMesh.tri2D, 182
fipy.meshes.numMesh.uniformGrid1D, 182
fipy.meshes.numMesh.uniformGrid2D, 182
fipy.meshes.numMesh.uniformGrid3D, 183
fipy.meshes.pyMesh.cell, 184

```
fipy.meshes.pyMesh.face, 184
fipy.meshes.pyMesh.face2D, 185
fipy.meshes.pyMesh.grid2D, 185
fipy.meshes.pyMesh.mesh, 187
fipy.meshes.pyMesh.test, 187
fipy.meshes.pyMesh.vertex, 187
fipy.meshes.test, 188
fipy.models.levelSet.advection.advection, 189
fipy.models.levelSet.advection.advectionTerm, 189
fipy.models.levelSet.advection.higherOrderAdvectionEquation, 189
fipy.models.levelSet.advection.higherOrderAdvectionTerm, 190
fipy.models.levelSet.distanceFunction.distanceVariable, 190
fipy.models.levelSet.distanceFunction.levelSetDiffusionEquation, 193
fipy.models.levelSet.distanceFunction.levelSetDiffusionVariable, 193
fipy.models.levelSet.electroChem.gapFillMesh, 194
fipy.models.levelSet.electroChem.metalIonDiffusionEquation, 196
fipy.models.levelSet.electroChem.metalIonSourceVariable, 197
fipy.models.levelSet.electroChem.test, 197
fipy.models.levelSet.surfactant.adsorbingSurfactantEquation, 198
fipy.models.levelSet.surfactant.convectionCoeff, 202
fipy.models.levelSet.surfactant.lines, 202
fipy.models.levelSet.surfactant.matplotlibSurfactantViewer, 202
fipy.models.levelSet.surfactant.mayaviSurfactantViewer, 203
fipy.models.levelSet.surfactant.surfactantBulkDiffusionEquation, 204
fipy.models.levelSet.surfactant.surfactantEquation, 205
fipy.models.levelSet.surfactant.surfactantEquation, 206
fipy.models.levelSet.test, 207
fipy.models.test, 207
fipy.solvers.pysparse.linearCGSSolver, 209
fipy.solvers.pysparse.linearGMRESSolver, 209
fipy.solvers.pysparse.linearJORSolver, 209
fipy.solvers.pysparse.linearLUSolver, 210
fipy.solvers.pysparse.linearPCGSolver, 210
fipy.solvers.pysparse.pysparseSolver, 210
fipy.solvers.solver, 214
fipy.solvers.trilinos.linearBicgstabSolver, 212
fipy.solvers.trilinos.linearCGSSolver, 212
fipy.solvers.trilinos.linearGMRESSolver, 212
fipy.solvers.trilinos.linearLUSolver, 212
fipy.solvers.trilinos.linearPCGSolver, 212
fipy.solvers.trilinos.preconditioners.domDecompPreconditioner, 212
fipy.solvers.trilinos.preconditioners.jacobiPreconditioner, 212
fipy.solvers.trilinos.preconditioners.multilevelDDM, 212
fipy.solvers.trilinos.preconditioners.multilevelDDM, 212
fipy.solvers.trilinos.preconditioners.multilevelNS, 212
fipy.solvers.trilinos.preconditioners.multilevelSAM, 212
fipy.solvers.trilinos.trilinosAztecOOsolver, 212
fipy.solvers.trilinos.trilinosMLTest, 212
fipy.solvers.trilinos.trilinosSolver, 212
fipy.steppers.pidStepper, 218
fipy.steppers.pseudoRKQSStepper, 219
fipy.steppers.rk45Stepper, 219
fipy.terms.cellTerm, 221
fipy.terms.centralDiffConvectionTerm, 221
fipy.terms.collectedDiffusionTerm, 222
fipy.terms.convectionTerm, 222
fipy.terms.diffusionTerm, 223
fipy.terms.equation, 224
fipy.terms.explicitDiffusionTerm, 224
fipy.terms.explicitSourceTerm, 224
```

fipy.terms.explicitUpwindConvectionTerm, fipy.variables.faceGradContributionsVariable,
 224
 fipy.terms.exponentialConvectionTerm,
 225
 fipy.terms.faceTerm, 227
 fipy.terms.hybridConvectionTerm, 227
 fipy.terms.implicitSourceTerm, 228
 fipy.terms.mulTerm, 228
 fipy.terms.nthOrderDiffusionTerm, 228
 fipy.terms.powerLawConvectionTerm, 228
 fipy.terms.sourceTerm, 229
 fipy.terms.term, 230
 fipy.terms.test, 232
 fipy.terms.transientTerm, 232
 fipy.terms.upwindConvectionTerm, 233
 fipy.terms.vanLeerConvectionTerm, 234
 fipy.test, 237
 fipy.tests.doctestPlus, 239
 fipy.tests.lateImportTest, 239
 fipy.tests.testBase, 239
 fipy.tests.testProgram, 239
 fipy.tools, 256
 fipy.tools.debug, 256
 fipy.tools.dimensions.DictWithDefault,
 241
 fipy.tools.dimensions.NumberDict, 241
 fipy.tools.dimensions.physicalField, 241
 fipy.tools.dump, 256
 fipy.tools.inline, 257
 fipy.tools.memoryLeak, 257
 fipy.tools.memoryLogger, 257
 fipy.tools.memoryUsage, 257
 fipy.tools.numerix, 257
 fipy.tools.parser, 266
 fipy.tools.pysparseMatrix, 267
 fipy.tools.sparseMatrix, 267
 fipy.tools.test, 267
 fipy.tools.trilinosMatrix, 267
 fipy.tools.vector, 267
 fipy.tools.vitals, 267
 fipy.variables.addOverFacesVariable, 269
 fipy.variables.arithmeticCellToFaceVariable,
 269
 fipy.variables.betaNoiseVariable, 269
 fipy.variables.binaryOperatorVariable,
 271
 fipy.variables.cellToFaceVariable, 271
 fipy.variables.cellVariable, 271
 fipy.variables.cellVolumeAverageVariable,
 275
 fipy.variables.constant, 275
 fipy.variables.exponentialNoiseVariable, fipy.viewers.gnuplotViewer.test, 301
 275
 276
 fipy.variables.faceGradVariable, 276
 fipy.variables.faceVariable, 276
 fipy.variables.fixedBCFaceGradVariable,
 277
 fipy.variables.gammaNoiseVariable, 277
 fipy.variables.gaussCellGradVariable,
 278
 fipy.variables.gaussianNoiseVariable,
 278
 fipy.variables.harmonicCellToFaceVariable,
 280
 fipy.variables.histogramVariable, 280
 fipy.variables.leastSquaresCellGradVariable,
 281
 fipy.variables.meshVariable, 281
 fipy.variables.minmodCellToFaceVariable,
 281
 fipy.variables.modCellGradVariable, 281
 fipy.variables.modCellToFaceVariable,
 281
 fipy.variables.modFaceGradVariable, 281
 fipy.variables.modPhysicalField, 281
 fipy.variables.modularVariable, 281
 fipy.variables.noiseVariable, 282
 fipy.variables.operatorVariable, 283
 fipy.variables.scharfetterGummelFaceVariable,
 283
 fipy.variables.test, 283
 fipy.variables.unaryOperatorVariable,
 283
 fipy.variables.uniformNoiseVariable, 283
 fipy.variables.variable, 284
 fipy.viewers, 312
 fipy.viewers.gistViewer, 291
 fipy.viewers.gistViewer.gist1DViewer,
 294
 fipy.viewers.gistViewer.gist2DViewer,
 295
 fipy.viewers.gistViewer.gistVectorViewer,
 296
 fipy.viewers.gistViewer.gistViewer, 297
 fipy.viewers.gistViewer.test, 297
 fipy.viewers.gnuplotViewer, 297
 fipy.viewers.gnuplotViewer.gnuplot1DViewer,
 299
 fipy.viewers.gnuplotViewer.gnuplot2DViewer,
 300
 fipy.viewers.gnuplotViewer.gnuplotViewer,
 301
 fipy.viewers.matplotlibViewer, 301

```
fipy.viewers.matplotlibViewer.matplotlib1DViewer,  
    305  
fipy.viewers.matplotlibViewer.matplotlib2DGridContourViewer,  
    305  
fipy.viewers.matplotlibViewer.matplotlib2DGridViewer,  
    306  
fipy.viewers.matplotlibViewer.matplotlib2DViewer,  
    306  
fipy.viewers.matplotlibViewer.matplotlibSparseMatrixViewer,  
    307  
fipy.viewers.matplotlibViewer.matplotlibVectorViewer,  
    308  
fipy.viewers.matplotlibViewer.matplotlibViewer,  
    309  
fipy.viewers.matplotlibViewer.test, 309  
fipy.viewers.mayaviViewer.mayaviClient,  
    309  
fipy.viewers.mayaviViewer.mayaviDaemon,  
    309  
fipy.viewers.mayaviViewer.test, 310  
fipy.viewers.multiViewer, 313  
fipy.viewers.test, 313  
fipy.viewers.testinteractive, 313  
fipy.viewers.tsvViewer, 314  
fipy.viewers.viewer, 315  
fipy.viewers.vtkViewer, 311  
fipy.viewers.vtkViewer.test, 311  
fipy.viewers.vtkViewer.vtkCellViewer,  
    312  
fipy.viewers.vtkViewer.vtkFaceViewer,  
    312  
fipy.viewers.vtkViewer.vtkViewer, 312
```

Index

Symbols

-PySparse
 command line option, 43

-Trilinos
 command line option, 43

-inline
 command line option, 43

:math:`\pi`^{108, 114, 115}
:module: fipy.tools.dump, 117
:module: fipy.tools.parser, 114
:module: fipy.viewers, 114, 116, 149
:module: parser, 134
:module: viewers, 99, 106

A

add() (fipy.tools.dimensions.physicalField.PhysicalField
 method), 244

addBoundingCell() (fipy.meshes.pyMesh.face.Face
 method), 184

AdsorbingSurfactantEquation (class
 in
 fipy.models.levelSet.surfactant.adsorbingSurfactantEquation),
 198

all() (fipy.variables.variable.Variable method), 285

allclose, 107

allclose() (fipy.tools.dimensions.physicalField.PhysicalField
 method), 244

allclose() (fipy.variables.variable.Variable method), 285

allclose() (in module fipy.tools.numerix), 258

allequal() (fipy.tools.dimensions.physicalField.PhysicalField
 method), 245

allequal() (fipy.variables.variable.Variable method), 285

allequal() (in module fipy.tools.numerix), 258

any() (fipy.variables.variable.Variable method), 285

appendChild() (fipy.tools.vitals.Vitals method), 267

appendInfo() (fipy.tools.vitals.Vitals method), 267

arccos() (fipy.tools.dimensions.physicalField.PhysicalField
 method), 245

arccos() (fipy.variables.variable.Variable method), 285

arccos() (in module fipy.tools.numerix), 258

arccosh() (fipy.tools.dimensions.physicalField.PhysicalField
 method), 245

arccosh() (fipy.variables.variable.Variable method), 285
arccosh() (in module fipy.tools.numerix), 259
arcsin() (fipy.tools.dimensions.physicalField.PhysicalField
 method), 245

arcsin() (fipy.variables.variable.Variable method), 285

arcsin() (in module fipy.tools.numerix), 259

arcsinh() (fipy.variables.variable.Variable method), 285

arcsinh() (in module fipy.tools.numerix), 259

arctan, 108

arctan() (fipy.tools.dimensions.physicalField.PhysicalField
 method), 245

arctan() (fipy.variables.variable.Variable method), 285

arctan() (in module fipy.tools.numerix), 260

arctan2, 108

arctan2() (fipy.tools.dimensions.physicalField.PhysicalField
 method), 246

arctan2() (fipy.variables.variable.Variable method), 285

arctan2() (in module fipy.tools.numerix), 260

arctanh() (fipy.tools.dimensions.physicalField.PhysicalField
 method), 246

arctanh() (fipy.variables.variable.Variable method), 285

arctanh() (in module fipy.tools.numerix), 260

array, 98

autoscale() (fipy.viewers.matplotlibViewer.matplotlibSparseMatrixViewer.S
 method), 307

B

BetaNoiseVariable (class
 in
 fipy.variables.betaNoiseVariable), 269

BoundaryCondition (class
 in
 fipy.boundaryConditions.boundaryCondition),
 165

buildAdvectionEquation() (in
 module
 fipy.models.levelSet.advection.advectionEquation),
 189

buildHigherOrderAdvectionEquation, 137

buildHigherOrderAdvectionEquation() (in
 module
 fipy.models.levelSet.advection.higherOrderAdvectionEquation),
 189

buildMetalIonDiffusionEquation, 137

buildMetalIonDiffusionEquation() (in
 module
 fipy.models.levelSet.electroChem.metalIonDiffusionEquation),

196
buildSurfactantBulkDiffusionEquation, 138
buildSurfactantBulkDiffusionEquation() (in module `copy()` (fipy.variables.noiseVariable.NoiseVariable
`fipy.models.levelSet.surfactant.surfactantBulkDiffusionEquation`), 282
`204`
buildTransitionMesh() (`fipy.models.levelSet.electroChem.gapField.MeshGapFieldMeshes`.physicalField.PhysicalField
`method`), 194

C

cacheMatrix, 149
cacheMatrix() (`fipy.terms.term.Term` method), 230
cacheMe() (`fipy.variables.variable.Variable` method), 285
cacheRHSvector, 149
cacheRHSvector() (`fipy.terms.term.Term` method), 230
calcDistanceFunction() (`fipy.models.levelSet.distanceFunction`.`distanceFunction`.`method`), 192
ceil() (`fipy.tools.dimensions.physicalField.PhysicalField` method), 246
ceil() (`fipy.variables.variable.Variable` method), 286
ceil() (in module `fipy.tools.numerix`), 261
Cell (class in `fipy.meshes.numMesh.cell`), 171
Cell (class in `fipy.meshes.pyMesh.cell`), 184
CellTerm (class in `fipy.terms.cellTerm`), 221
CellVariable, 76, 81, 86, 94, 102, 112, 115, 136, 148, 152,
`154`
CellVariable (class in `fipy.variables.cellVariable`), 271
CentralDifferenceConvectionTerm (class in `fipy.terms.centralDiffConvectionTerm`), 221
clip_data() (`fipy.viewers.mayaviViewer.mayaviDaemon.MayaviDaemon` method), 310
command line option
 –`PySparse`, 43
 –`Trilinos`, 43
 –`inline`, 43
conjugate() (`fipy.tools.dimensions.physicalField.PhysicalField` method), 246
conjugate() (`fipy.variables.variable.Variable` method), 286
conjugate() (in module `fipy.tools.numerix`), 261
ConvectionTerm (class in `fipy.terms.convectionTerm`), 222
conversionFactorTo() (`fipy.tools.dimensions.physicalField.PhysicalUnit` method), 254
conversionTupleTo() (`fipy.tools.dimensions.physicalField.PhysicalUnit` method), 254
convertToUnit() (`fipy.tools.dimensions.physicalField.PhysicalField` method), 246
copy() (`fipy.models.levelSet.surfactant.surfactantVariable.SurfactantVariable`.`SurfactantVariable` method), 206
copy() (`fipy.terms.term.Term` method), 230
copy() (`fipy.tools.dimensions.physicalField.PhysicalField` method), 246
copy() (`fipy.variables.cellVariable.CellVariable` method), 271

copy() (`fipy.variables.faceVariable.FaceVariable` method),
`276`
copy() (fipy.variables.noiseVariable.NoiseVariable
`fipy.models.levelSet.surfactant.surfactantBulkDiffusionEquation`), 282
copy() (fipy.variables.variable.Variable method), 286
copy() (fipy.tools.numerix), 261
cosh() (`fipy.tools.dimensions.physicalField.PhysicalField` method), 247
cosh() (`fipy.variables.variable.Variable` method), 286
cosh() (in module `fipy.tools.numerix`), 261
CylindricalGrid1D (class in `fipy.meshes.numMesh.cylindricalGrid1D`),
`171`
CylindricalGrid1D() (in module `fipy.meshes.cylindricalGrid1D`), 187
CylindricalGrid2D (class in `fipy.meshes.numMesh.cylindricalGrid2D`),
`172`
CylindricalGrid2D() (in module `fipy.meshes.cylindricalGrid2D`), 187
CylindricalUniformGrid1D (class in `fipy.meshes.numMesh.cylindricalUniformGrid1D`),
`172`
CylindricalUniformGrid2D (class in `fipy.meshes.numMesh.cylindricalUniformGrid2D`),
`173`

D

DefaultAsymmetricSolver, 81, 99, 106
dictToXML() (`fipy.tools.vitals.Vitals` method), 267
DiffusionTerm (class in `fipy.terms.diffusionTerm`), 223
DiffusionTermNoCorrection (class in `fipy.terms.diffusionTerm`), 224
DISPLAY, 16
DistanceVariable, 135
DistanceVariable (class in `fipy.models.levelSet.distanceFunction.distanceVariable`),
`190`
divide() (`fipy.tools.dimensions.physicalField.PhysicalField` method), 247
DomDecompPreconditioner (class in `fipy.solvers.trilinos.preconditioners.domDecompPreconditioner`),
`211`
dotAndCopyMat (`fipy.variables.variable.Variable` method),
`286`
dot() (`fipy.tools.dimensions.physicalField.PhysicalField` method), 248
dot() (`fipy.variables.variable.Variable` method), 286
dot() (in module `fipy.tools.numerix`), 261
DYLD_LIBRARY_PATH, 14

E

environment variable
 DISPLAY, 16
 DYLD_LIBRARY_PATH, 14
 FIPY_DISPLAY_MATRIX, 43
 FIPY_INLINE, 43
 FIPY_INLINE_COMMENT, 43
 FIPY_SOLVERS, 14, 43
 FIPY_VIEWER, 9, 44
 GISTPATH, 10
 LD_LIBRARY_PATH, 14
 PYTHONPATH, 8, 9, 11, 13
 error() (in module fipy.steppers), 218
 examples.cahnHilliard.mesh2D (module), 141
 examples.cahnHilliard.sphere (module), 142
 examples.convection.exponential1D.mesh1D (module), 79
 examples.convection.exponential1DSource.mesh1D (module), 80
 examples.convection.robin (module), 82
 examples.convection.source (module), 83
 examples.diffusion.anisotropy (module), 77
 examples.diffusion.circle (module), 66
 examples.diffusion.electrostatics (module), 71
 examples.diffusion.mesh1D (module), 51
 examples.diffusion.mesh20x20 (module), 64
 examples.diffusion.nthOrder.input4thOrder1D (module), 75
 examples.flow.stokesCavity (module), 147
 examples.levelSet.advection.circle (module), 122
 examples.levelSet.advection.mesh1D (module), 121
 examples.levelSet.distanceFunction.circle (module), 120
 examples.levelSet.distanceFunction.mesh1D (module), 119
 examples.levelSet.electroChem.gold (module), 128
 examples.levelSet.electroChem.howToWriteAScript (module), 133
 examples.levelSet.electroChem.leveler (module), 130
 examples.levelSet.electroChem.simpleTrenchSystem (module), 125
 examples.phase.anisotropy (module), 107
 examples.phase.binary (module), 93
 examples.phase.impingement.mesh20x20 (module), 114
 examples.phase.impingement.mesh40x1 (module), 111
 examples.phase.quaternary (module), 102
 examples.phase.simple (module), 85
 examples.updating.update0_1to1_0 (module), 151
 examples.updating.update1_0to2_0 (module), 155
 execButNoTest() (in module fipy.tests.doctestPlus), 239
 exp, 80, 81, 98, 113, 116, 135
 exp() (fipy.variables.variable.Variable method), 286
 exp() (in module fipy.tools.numerix), 262
 ExplicitDiffusionTerm, 52, 113, 116

ExplicitDiffusionTerm (class in fipy.terms.explicitDiffusionTerm), 224
 ExplicitNthOrderDiffusionTerm (class in fipy.terms.nthOrderDiffusionTerm), 228
 ExplicitUpwindConvectionTerm (class in fipy.terms.explicitUpwindConvectionTerm), 224
 ExponentialConvectionTerm, 152
 ExponentialConvectionTerm (class in fipy.terms.exponentialConvectionTerm), 225
 ExponentialNoiseVariable (class in fipy.variables.exponentialNoiseVariable), 275
 exportAsMesh() (in module fipy.meshes.numMesh.gmshExport), 173
 extendVariable() (fipy.models.levelSet.distanceFunction.distanceVariable.D method), 192
 extrude() (fipy.meshes.numMesh.mesh2D.Mesh2D method), 178

F

Face (class in fipy.meshes.numMesh.face), 173
 Face (class in fipy.meshes.pyMesh.face), 184
 Face2D (class in fipy.meshes.pyMesh.face2D), 185
 FaceTerm (class in fipy.terms.faceTerm), 227
 FaceVariable, 58
 FaceVariable (class in fipy.variables.faceVariable), 276
 failFn() (fipy.steppers.stepper.Stepper static method), 219
 FiPy, 45
 fipy.boundaryConditions.boundaryCondition (module), 165
 fipy.boundaryConditions.fixedFlux (module), 165
 fipy.boundaryConditions.fixedValue (module), 166
 fipy.boundaryConditions.nthOrderBoundaryCondition (module), 166
 fipy.boundaryConditions.test (module), 167
 fipy.meshes.common.mesh (module), 169
 fipy.meshes.cylindricalGrid1D (module), 187
 fipy.meshes.cylindricalGrid2D (module), 187
 fipy.meshes.grid1D (module), 188
 fipy.meshes.grid2D (module), 188
 fipy.meshes.grid3D (module), 188
 fipy.meshes.numMesh.cell (module), 171
 fipy.meshes.numMesh.cylindricalGrid1D (module), 171
 fipy.meshes.numMesh.cylindricalGrid2D (module), 172
 fipy.meshes.numMesh.cylindricalUniformGrid1D (module), 172
 fipy.meshes.numMesh.cylindricalUniformGrid2D (module), 173
 fipy.meshes.numMesh.face (module), 173
 fipy.meshes.numMesh.gmshExport (module), 173
 fipy.meshes.numMesh.gmshImport (module), 173
 fipy.meshes.numMesh.grid1D (module), 176

fipy.meshes.numMesh.grid2D (module), 177
fipy.meshes.numMesh.grid3D (module), 177
fipy.meshes.numMesh.mesh (module), 178
fipy.meshes.numMesh.mesh1D (module), 178
fipy.meshes.numMesh.mesh2D (module), 178
fipy.meshes.numMesh.periodicGrid1D (module), 179
fipy.meshes.numMesh.periodicGrid2D (module), 180
fipy.meshes.numMesh.skewedGrid2D (module), 181
fipy.meshes.numMesh.test (module), 181
fipy.meshes.numMesh.tri2D (module), 182
fipy.meshes.numMesh.uniformGrid1D (module), 182
fipy.meshes.numMesh.uniformGrid2D (module), 182
fipy.meshes.numMesh.uniformGrid3D (module), 183
fipy.meshes.pyMesh.cell (module), 184
fipy.meshes.pyMesh.face (module), 184
fipy.meshes.pyMesh.face2D (module), 185
fipy.meshes.pyMesh.grid2D (module), 185
fipy.meshes.pyMesh.mesh (module), 187
fipy.meshes.pyMesh.test (module), 187
fipy.meshes.pyMesh.vertex (module), 187
fipy.meshes.test (module), 188
fipy.models.levelSet.advection.advectionEquation (module), 189
fipy.models.levelSet.advection.advectionTerm (module), 189
fipy.models.levelSet.advection.higherOrderAdvectionEquation (module), 189
fipy.models.levelSet.advection.higherOrderAdvectionTerm (module), 190
fipy.models.levelSet.distanceFunction.distanceVariable (module), 190
fipy.models.levelSet.distanceFunction.levelSetDiffusionEquation (module), 193
fipy.models.levelSet.distanceFunction.levelSetDiffusionVariable (module), 193
fipy.models.levelSet.electroChem.gapFillMesh (module), 194
fipy.models.levelSet.electroChem.metalIonDiffusionEquation (module), 196
fipy.models.levelSet.electroChem.metalIonSourceVariable (module), 197
fipy.models.levelSet.electroChem.test (module), 197
fipy.models.levelSet.surfactant.adsorbingSurfactantEquation (module), 198
fipy.models.levelSet.surfactant.convectionCoeff (module), 202
fipy.models.levelSet.surfactant.lines (module), 202
fipy.models.levelSet.surfactant.matplotlibSurfactantViewer (module), 202
fipy.models.levelSet.surfactant.mayaviSurfactantViewer (module), 203
fipy.models.levelSet.surfactant.surfactantBulkDiffusionEquation (module), 204

fipy.models.levelSet.surfactant.surfactantEquation (module), 205
fipy.models.levelSet.surfactant.surfactantVariable (module), 206
fipy.models.levelSet.test (module), 207
fipy.models.test (module), 207
fipy.solvers.pysparse.linearCGSSolver (module), 209
fipy.solvers.pysparse.linearGMRESSolver (module), 209
fipy.solvers.pysparse.linearJORSolver (module), 209
fipy.solvers.pysparse.linearLUSolver (module), 210
fipy.solvers.pysparse.linearPCGSolver (module), 210
fipy.solvers.pysparse.pysparseSolver (module), 210
fipy.solvers.solver (module), 214
fipy.solvers.test (module), 215
fipy.solvers.trilinos.linearBicgstabSolver (module), 212
fipy.solvers.trilinos.linearCGSSolver (module), 212
fipy.solvers.trilinos.linearGMRESSolver (module), 213
fipy.solvers.trilinos.linearLUSolver (module), 213
fipy.solvers.trilinos.linearPCGSolver (module), 213
fipy.solvers.trilinos.preconditioners.domDecompPreconditioner (module), 211
fipy.solvers.trilinos.preconditioners.icPreconditioner (module), 211
fipy.solvers.trilinos.preconditioners.jacobiPreconditioner (module), 211
fipy.solvers.trilinos.preconditioners.multilevelDDMLPreconditioner (module), 211
fipy.solvers.trilinos.preconditioners.multilevelDDPPreconditioner (module), 211
fipy.solvers.trilinos.preconditioners.multilevelNSSAPreconditioner (module), 211
fipy.solvers.trilinos.preconditioners.multilevelSAPreconditioner (module), 212
fipy.solvers.trilinos.preconditioners.preconditioner (module), 212
fipy.solvers.trilinos.trilinosAztecOOsolver (module), 213
fipy.solvers.trilinos.trilinosMLTest (module), 214
fipy.solvers.trilinos.trilinosSolver (module), 214
fipy.steppers (module), 217
fipy.steppers.pidStepper (module), 218
fipy.steppers.pseudoRKQSSStepper (module), 219
fipy.steppers.stepper (module), 219
fipy.terms.cellTerm (module), 221
fipy.terms.centralDiffConvectionTerm (module), 221
fipy.terms.collectedDiffusionTerm (module), 222
fipy.terms.convectionTerm (module), 222
fipy.terms.diffusionTerm (module), 223
fipy.terms.equation (module), 224
fipy.terms.explicitDiffusionTerm (module), 224
fipy.terms.explicitSourceTerm (module), 224
fipy.terms.explicitUpwindConvectionTerm (module), 224
fipy.terms.exponentialConvectionTerm (module), 225
fipy.terms.faceTerm (module), 227
fipy.terms.hybridConvectionTerm (module), 227

fipy.terms.implicitSourceTerm (module), 228
fipy.terms.mulTerm (module), 228
fipy.terms.nthOrderDiffusionTerm (module), 228
fipy.terms.powerLawConvectionTerm (module), 228
fipy.terms.sourceTerm (module), 229
fipy.terms.term (module), 230
fipy.terms.test (module), 232
fipy.terms.transientTerm (module), 232
fipy.terms.upwindConvectionTerm (module), 233
fipy.terms.vanLeerConvectionTerm (module), 234
fipy.test (module), 237
fipy.tests doctestPlus (module), 239
fipy.tests.lateImportTest (module), 239
fipy.tests.testBase (module), 239
fipy.tests.testProgram (module), 239
fipy.tools (module), 256
fipy.tools.debug (module), 256
fipy.tools.dimensions.DictWithDefault (module), 241
fipy.tools.dimensions.NumberDict (module), 241
fipy.tools.dimensions.physicalField (module), 241
fipy.tools.dump (module), 256
fipy.tools.inline (module), 257
fipy.tools.memoryLeak (module), 257
fipy.tools.memoryLogger (module), 257
fipy.tools.memoryUsage (module), 257
fipy.tools.numerix (module), 257
fipy.tools.parser (module), 266
fipy.tools.pysparseMatrix (module), 267
fipy.tools.sparseMatrix (module), 267
fipy.tools.test (module), 267
fipy.tools.trilinosMatrix (module), 267
fipy.tools.vector (module), 267
fipy.tools.vitals (module), 267
fipy.variables.addOverFacesVariable (module), 269
fipy.variables.arithmeticCellToFaceVariable (module), 269
fipy.variables.betaNoiseVariable (module), 269
fipy.variables.binaryOperatorVariable (module), 271
fipy.variables.cellToFaceVariable (module), 271
fipy.variables.cellVariable (module), 271
fipy.variables.cellVariable.CellVariable
object, 67
fipy.variables.cellVolumeAverageVariable (module), 275
fipy.variables.constant (module), 275
fipy.variables.exponentialNoiseVariable (module), 275
fipy.variables.faceGradContributionsVariable (module), 276
fipy.variables.faceGradVariable (module), 276
fipy.variables.faceVariable (module), 276
fipy.variables.fixedBCFaceGradVariable (module), 277
fipy.variables.gammaNoiseVariable (module), 277
fipy.variables.gaussCellGradVariable (module), 278
fipy.variables.gaussianNoiseVariable (module), 278
fipy.variables.harmonicCellToFaceVariable (module), 280
fipy.variables.histogramVariable (module), 280
fipy.variables.leastSquaresCellGradVariable (module), 281
fipy.variables.meshVariable (module), 281
fipy.variables.minmodCellToFaceVariable (module), 281
fipy.variables.modCellGradVariable (module), 281
fipy.variables.modCellToFaceVariable (module), 281
fipy.variables.modFaceGradVariable (module), 281
fipy.variables.modPhysicalField (module), 281
fipy.variables.modularVariable (module), 281
fipy.variables.noiseVariable (module), 282
fipy.variables.operatorVariable (module), 283
fipy.variables.scharfetterGummelFaceVariable (module), 283
fipy.variables.test (module), 283
fipy.variables.unaryOperatorVariable (module), 283
fipy.variables.uniformNoiseVariable (module), 283
fipy.variables.variable (module), 284
fipy.viewers
module, 52, 65, 77, 86
fipy.viewers (module), 312
fipy.viewers.gistViewer (module), 291
fipy.viewers.gistViewer.gist1DViewer (module), 294
fipy.viewers.gistViewer.gist2DViewer (module), 295
fipy.viewers.gistViewer.gistVectorViewer (module), 296
fipy.viewers.gistViewer.gistViewer (module), 297
fipy.viewers.gistViewer.test (module), 297
fipy.viewers.gnuplotViewer (module), 297
fipy.viewers.gnuplotViewer.gnuplot1DViewer (module), 299
fipy.viewers.gnuplotViewer.gnuplot2DViewer (module), 300
fipy.viewers.gnuplotViewer.gnuplotViewer (module), 301
fipy.viewers.gnuplotViewer.test (module), 301
fipy.viewers.matplotlibViewer (module), 301
fipy.viewers.matplotlibViewer.matplotlib1DViewer
(module), 305
fipy.viewers.matplotlibViewer.matplotlib2DGridContourViewer
(module), 305
fipy.viewers.matplotlibViewer.matplotlib2DGridViewer
(module), 306
fipy.viewers.matplotlibViewer.matplotlib2DViewer
(module), 306
fipy.viewers.matplotlibViewer.matplotlibSparseMatrixViewer
(module), 307
fipy.viewers.matplotlibViewer.matplotlibVectorViewer
(module), 308
fipy.viewers.matplotlibViewer.matplotlibViewer (mod-
ule), 309
fipy.viewers.matplotlibViewer.test (module), 309
fipy.viewers.mayaviViewer.mayaviClient (module), 309

fipy.viewers.mayaviViewer.mayaviDaemon (module), 309
fipy.viewers.mayaviViewer.test (module), 310
fipy.viewers.multiViewer (module), 313
fipy.viewers.test (module), 313
fipy.viewers.testinteractive (module), 313
fipy.viewers.tsvViewer (module), 314
fipy.viewers.tsvViewer.TSVViewer object, 69
fipy.viewers.viewer (module), 315
fipy.viewers.vtkViewer (module), 311
fipy.viewers.vtkViewer.test (module), 311
fipy.viewers.vtkViewer.vtkCellViewer (module), 312
fipy.viewers.vtkViewer.vtkFaceViewer (module), 312
fipy.viewers.vtkViewer.vtkViewer (module), 312
FIPY_SOLVERS, 14, 43
FIPY_VIEWER, 9
FixedFlux, 59, 76, 151
FixedFlux (class in fipy.boundaryConditions.fixedFlux), 165
FixedValue, 64, 76, 81, 137, 149, 151, 153
FixedValue (class in fipy.boundaryConditions.fixedValue), 166
floor() (fipy.tools.dimensions.physicalField.PhysicalField method), 248
floor() (fipy.variables.variable.Variable method), 286
floor() (in module fipy.tools.numerix), 262

G

GammaNoiseVariable (class in fipy.variables.gammaNoiseVariable), 277
GapFillMesh (class in fipy.models.levelSet.electroChem.gapFillMesh), 194
GaussianNoiseVariable (class in fipy.variables.gaussianNoiseVariable), 278
getArea() (fipy.meshes.numMesh.face.Face method), 173
getArea() (fipy.meshes.pyMesh.face.Face method), 184
getArithmetricFaceValue() (fipy.variables.cellVariable.CellVariable method), 271
getArithmetricFaceValue() (fipy.variables.modularVariable.ModularVariable method), 282
getArray() (fipy.viewers.gistViewer.GistVectorViewer method), 294
getArray() (fipy.viewers.gistViewer.gistVectorViewer.GistVectorViewer method), 297
getBottomFaces() (fipy.models.levelSet.electroChem.gapFillMesh.TrefftzMesh method), 196
getBoundingCells() (fipy.meshes.pyMesh.cell.Cell method), 184
getCellCenters() (fipy.meshes.common.mesh.Mesh method), 169
(getCellCenters() (fipy.meshes.numMesh.cylindricalGrid1D.CylindricalGrid method), 172
getCellCenters() (fipy.meshes.numMesh.cylindricalGrid2D.CylindricalGrid method), 172
getCellCenters() (fipy.meshes.numMesh.periodicGrid1D.PeriodicGrid1D method), 180
getCellCenters() (fipy.meshes.pyMesh.grid2D.Grid2D method), 186
getCellDistance() (fipy.meshes.pyMesh.face.Face method), 184
getCellID() (fipy.meshes.numMesh.face.Face method), 173
getCellID() (fipy.meshes.pyMesh.face.Face method), 184
getCellIDsAboveFineRegion() (fipy.models.levelSet.electroChem.gapFillMesh.GapFillMesh method), 195
getCellInterfaceAreas() (fipy.models.levelSet.distanceFunction.distanceVari method), 192
getCells() (fipy.meshes.pyMesh.face.Face method), 185
getCellVolumeAverage() (fipy.variables.cellVariable.CellVariable method), 272
getCellVolumes() (fipy.meshes.common.mesh.Mesh method), 169
getCellVolumes() (fipy.meshes.numMesh.cylindricalGrid2D.CylindricalGrid method), 172
getCellVolumes() (fipy.meshes.numMesh.cylindricalUniformGrid1D.Cylind method), 172
getCellVolumes() (fipy.meshes.numMesh.cylindricalUniformGrid2D.Cylind method), 173
getCellVolumes() (fipy.meshes.numMesh.gmshImport.GmshImporter2D method), 176
getCellVolumes() (fipy.meshes.numMesh.gmshImport.GmshImporter3D method), 176
getCellVolumes() (fipy.meshes.numMesh.uniformGrid1D.UniformGrid1D method), 182
getCellVolumes() (fipy.meshes.numMesh.uniformGrid2D.UniformGrid2D method), 183
getCellVolumes() (fipy.meshes.numMesh.uniformGrid3D.UniformGrid3D method), 183
getCellVolumes() (fipy.meshes.pyMesh.grid2D.Grid2D method), 186
getCenter() (fipy.meshes.numMesh.cell.Cell method), 171
getCenter() (fipy.meshes.numMesh.face.Face method), 173
getCenter() (fipy.meshes.pyMesh.cell.Cell method), 184
getCenter() (fipy.meshes.pyMesh.face.Face method), 185
getCoordinates() (fipy.meshes.pyMesh.vertex.Vertex method), 187
getDefaultSolver() (fipy.terms.term.Term method), 230
getDim() (fipy.meshes.common.mesh.Mesh method), 169
getDim() (fipy.meshes.numMesh.grid1D.Grid1D method), 177

getDivergence() (fipy.variables.faceVariable.FaceVariable method), 276

getElectrolyteMask() (fipy.models.levelSet.electroChem.gapFillMeshFace) (fipy.meshes.common.mesh.Mesh method), 196

getExteriorFaces() (fipy.meshes.common.mesh.Mesh method), 169

getExteriorFaces() (fipy.meshes.numMesh.mesh.Mesh method), 178

getExteriorFaces() (fipy.meshes.numMesh.uniformGrid2D.UniformGrid2D) (fipy.meshes.common.mesh.Mesh method), 183

getExteriorFaces() (fipy.meshes.numMesh.uniformGrid3D.UniformGrid3D) (fipy.meshes.pyMesh.grid2D.Grid2D method), 183

getExteriorFaces() (fipy.meshes.pyMesh.mesh.Mesh method), 187

getFaceCellIDs() (fipy.meshes.numMesh.mesh.Mesh method), 178

getFaceCellIDs() (fipy.meshes.numMesh.uniformGrid1D.UniformGrid1D) (fipy.meshes.common.mesh.Mesh method), 182

getFaceCellIDs() (fipy.meshes.numMesh.uniformGrid2D.UniformGrid2D) (fipy.variables.cellVariable.CellVariable method), 183

getFaceCellIDs() (fipy.meshes.numMesh.uniformGrid3D.UniformGrid3D) (fipy.meshes.numMesh.gmshImport.MshFile method), 183

getFaceCenters() (fipy.meshes.numMesh.cylindricalGrid1D.CylindricalGrid1D) (fipy.models.levelSet.electroChem.gapFillMeshFace) (fipy.meshes.common.mesh.Mesh method), 172

getFaceCenters() (fipy.meshes.numMesh.cylindricalGrid2D.CylindricalGrid2D) (fipy.variables.cellVariable.CellVariable method), 172

getFaceCenters() (fipy.meshes.numMesh.mesh.Mesh method), 178

getFaceCenters() (fipy.meshes.numMesh.uniformGrid1D.UniformGrid1D) (fipy.variables.faceVariable.FaceVariable method), 182

getFaceCenters() (fipy.meshes.numMesh.uniformGrid2D.UniformGrid2D) (fipy.variables.cellVariable.CellVariable method), 183

getFaceCenters() (fipy.meshes.numMesh.uniformGrid3D.UniformGrid3D) (fipy.variables.modularVariable.ModularVariable method), 183

getFaceGrad() (fipy.variables.cellVariable.CellVariable method), 272

getFaceGrad() (fipy.variables.modularVariable.ModularVariable method), 282

getFaceGradAverage() (fipy.variables.cellVariable.CellVariable method), 272

getFaceGradNoMod() (fipy.variables.modularVariable.ModularVariable method), 282

getFaceIDs() (fipy.meshes.pyMesh.cell.Cell method), 184

getFaceOrientations() (fipy.meshes.pyMesh.cell.Cell method), 184

getFaceOrientations() (fipy.meshes.pyMesh.mesh.Mesh method), 187

getFaces() (fipy.meshes.pyMesh.cell.Cell method), 184

getFacesBack() (fipy.meshes.common.mesh.Mesh method), 169

getFacesBottom() (fipy.meshes.common.mesh.Mesh method), 169

getFacesBottom() (fipy.meshes.pyMesh.grid2D.Grid2D method), 186

getFacesDown() (fipy.meshes.common.mesh.Mesh method), 170

getFacesLeft() (fipy.meshes.common.mesh.Mesh method), 170

getFacesLeft() (fipy.meshes.pyMesh.grid2D.Grid2D method), 186

getFacesTop() (fipy.meshes.common.mesh.Mesh method), 170

getFacesTop() (fipy.meshes.pyMesh.grid2D.Grid2D method), 186

getFaceCellID() (fipy.meshes.common.mesh.Mesh method), 171

getFaceValue() (fipy.variables.cellVariable.CellVariable method), 272

getFaceGridID() (fipy.meshes.common.mesh.Mesh method), 176

getGlobalValue() (fipy.variables.cellVariable.CellVariable method), 273

getGlobalValue() (fipy.variables.faceVariable.FaceVariable method), 277

getGridID() (fipy.variables.cellVariable.CellVariable method), 273

getHarmonicFaceValue() (fipy.variables.cellVariable.CellVariable method), 273

getID() (fipy.meshes.numMesh.cell.Cell method), 171

getID() (fipy.meshes.numMesh.face.Face method), 173

getID() (fipy.meshes.pyMesh.cell.Cell method), 184

getInterfaceVar() (fipy.models.levelSet.surfactant.surfactantVariable.SurfactantVariable method), 206

getInteriorFaces() (fipy.meshes.common.mesh.Mesh method), 171

getInteriorFaces() (fipy.meshes.numMesh.mesh.Mesh method), 178

getInteriorFaces() (fipy.meshes.numMesh.uniformGrid1D.UniformGrid1D method), 182

getInteriorFaces() (fipy.meshes.numMesh.uniformGrid2D.UniformGrid2D method), 183

getInteriorFaces() (fipy.meshes.numMesh.uniformGrid3D.UniformGrid3D method), 183

getLeastSquaresGrad() (fipy.variables.cellVariable.CellVariable method), 273

getMag() (fipy.variables.variable.Variable method), 286

getMatrix, 149

getMatrix() (fipy.terms.term.Term method), 230

getMesh() (fipy.meshes.numMesh.cell.Cell method), 171

getMesh() (fipy.meshes.numMesh.face.Face method), 173

getMinmodFaceValue() (fipy.variables.cellVariable.CellVariable method), 274

getName() (fipy.variables.variable.Variable method), 286

getNearestCell() (fipy.meshes.common.mesh.Mesh method), 171

getNormal() (fipy.meshes.numMesh.cell.Cell method), 171

getNormal() (fipy.meshes.pyMesh.face.Face method), 185

getNumberOfCells() (fipy.meshes.common.mesh.Mesh method), 171

getNumericValue() (fipy.tools.dimensions.physicalField.PhysicalField method), 248

getNumericValue() (fipy.variables.variable.Variable method), 286

getOld() (fipy.variables.cellVariable.CellVariable method), 274

getPhysicalShape() (fipy.meshes.numMesh.grid1D.Grid1D method), 177

getPhysicalShape() (fipy.meshes.numMesh.grid2D.Grid2D method), 177

getPhysicalShape() (fipy.meshes.numMesh.grid3D.Grid3D method), 177

getPhysicalShape() (fipy.meshes.numMesh.skewedGrid2D.SkewedGrid2D method), 181

getPhysicalShape() (fipy.meshes.numMesh.tri2D.Tri2D method), 182

getPhysicalShape() (fipy.meshes.pyMesh.grid2D.Grid2D method), 187

getPhysicalShape() (fipy.meshes.pyMesh.mesh.Mesh method), 187

getRHSvector, 149

getRHSvector() (fipy.terms.term.Term method), 230

getScale() (fipy.meshes.numMesh.grid1D.Grid1D method), 177

getScale() (fipy.meshes.numMesh.grid2D.Grid2D method), 177

getScale() (fipy.meshes.numMesh.grid3D.Grid3D method), 177

getScale() (fipy.meshes.numMesh.skewedGrid2D.SkewedGrid2D method), 181

getScale() (fipy.meshes.numMesh.tri2D.Tri2D method), 182

getScale() (fipy.meshes.pyMesh.mesh.Mesh method), 187

getShape() (fipy.tools.dimensions.physicalField.PhysicalField method), 248

getShape() (fipy.variables.variable.Variable method), 287

getShape() (fipy.meshes.numMesh.grid1D.Grid1D method), 177

getShape() (fipy.meshes.numMesh.grid2D.Grid2D method), 177

getShape() (fipy.meshes.numMesh.grid3D.Grid3D method), 177

getShape() (fipy.meshes.numMesh.skewedGrid2D.SkewedGrid2D method), 181

getShape() (fipy.meshes.numMesh.tri2D.Tri2D method), 182

getShape() (fipy.meshes.pyMesh.grid2D.Grid2D method), 187

getShape() (fipy.tools.dimensions.physicalField.PhysicalField method), 248

getShape() (fipy.variables.variable.Variable method), 286

getShape() (in module fipy.tools.numerix), 262

getShapeListedVariables() (fipy.variables.variable.Variable method), 287

getTopFaces() (fipy.models.levelSet.electroChem.gapFillMesh.TrenchMesh method), 196

getUnit() (fipy.tools.dimensions.physicalField.PhysicalField method), 248

getUnit() (fipy.variables.variable.Variable method), 287

getUnit() (in module fipy.tools.numerix), 263

getValue() (fipy.variables.variable.Variable method), 287

getVertexCoords() (fipy.meshes.numMesh.cylindricalGrid1D.CylindricalGrid1D method), 172

getVertexCoords() (fipy.meshes.numMesh.cylindricalGrid2D.CylindricalGrid2D method), 172

getVertexCoords() (fipy.meshes.numMesh.mesh.Mesh method), 178

getVertexCoords() (fipy.meshes.numMesh.uniformGrid1D.UniformGrid1D method), 182

getVertexCoords() (fipy.meshes.numMesh.uniformGrid2D.UniformGrid2D method), 183

getVertexCoords() (fipy.meshes.numMesh.uniformGrid3D.UniformGrid3D method), 183

getViewers() (fipy.viewers.multiViewer.MultiViewer method), 313

getVolume() (fipy.meshes.pyMesh.cell.Cell method), 184

getVTKCellDataSet() (fipy.meshes.numMesh.mesh.Mesh method), 178

getVTKFaceDataSet() (fipy.meshes.numMesh.mesh.Mesh method), 178

Gist1DViewer, 154

Gist1DViewer (class in fipy.viewers.gistViewer), 291

Gist1DViewer (class in fipy.viewers.gistViewer.gist1DViewer), 294

Gist2DViewer (class in fipy.viewers.gistViewer), 292

Gist2DViewer	(class fipy.viewers.gistViewer.gist2DViewer),	295	in	inBaseUnits() (fipy.tools.dimensions.physicalField.PhysicalField method), 248
GISTPATH, 10				inBaseUnits() (fipy.variables.variable.Variable method), 287
GistVectorViewer (class in fipy.viewers.gistViewer),	293		in	indices() (in module fipy.tools.numerix), 263
GistVectorViewer	(class fipy.viewers.gistViewer.gistVectorViewer),	296		inDimensionless() (fipy.tools.dimensions.physicalField.PhysicalField method), 248
GistViewer() (in module fipy.viewers.gistViewer),	291			inRadians() (fipy.tools.dimensions.physicalField.PhysicalField method), 249
Gmsh, 45				inSIUnits() (fipy.tools.dimensions.physicalField.PhysicalField method), 249
gmsh, 127, 129, 130				inUnitsOf() (fipy.tools.dimensions.physicalField.PhysicalField method), 249
GmshImporter2D	(class fipy.meshes.numMesh.gmshImport),	176	in	inUnitsOf() (fipy.variables.variable.Variable method), 287
GmshImporter2DIn3DSpace	(class fipy.meshes.numMesh.gmshImport),	176	in	isAngle() (fipy.tools.dimensions.physicalField.PhysicalUnit method), 254
GmshImporter3D	(class fipy.meshes.numMesh.gmshImport),	176	in	isclose() (in module fipy.tools.numerix), 263
gnuplot, 45				isCompatible() (fipy.tools.dimensions.physicalField.PhysicalField method), 250
Gnuplot1DViewer (class in fipy.viewers.gnuplotViewer),	297			isCompatible() (fipy.tools.dimensions.physicalField.PhysicalUnit method), 255
Gnuplot1DViewer	(class fipy.viewers.gnuplotViewer.gnuplot1DViewer),	299	in	isDimensionless() (fipy.tools.dimensions.physicalField.PhysicalUnit method), 255
Gnuplot2DViewer (class in fipy.viewers.gnuplotViewer),	298			isDimensionlessOrAngle() (fipy.tools.dimensions.physicalField.PhysicalUnit method), 255
Gnuplot2DViewer	(class fipy.viewers.gnuplotViewer.gnuplot2DViewer),	300	in	isFloat() (in module fipy.tools.numerix), 263
GnuplotViewer()	(in fipy.viewers.gnuplotViewer),	297	module	isInt() (in module fipy.tools.numerix), 263
Grid1D, 75, 79, 80, 86, 93, 102, 111, 122, 153				isInverseAngle() (fipy.tools.dimensions.physicalField.PhysicalUnit method), 255
Grid1D (class in fipy.meshes.numMesh.grid1D),	176			itemset() (fipy.tools.dimensions.physicalField.PhysicalField method), 250
Grid1D() (in module fipy.meshes.grid1D),	188			itemset() (fipy.variables.variable.Variable method), 288
Grid2D, 64, 114, 119, 120, 135, 148, 151				Iterator, 152
Grid2D (class in fipy.meshes.numMesh.grid2D),	177			
Grid2D (class in fipy.meshes.pyMesh.grid2D),	185			
Grid2D() (in module fipy.meshes.grid2D),	188			
Grid2DGistViewer, 153				
Grid3D (class in fipy.meshes.numMesh.grid3D),	177			
Grid3D() (in module fipy.meshes.grid3D),	188			
H				
HistogramVariable	(class fipy.variables.histogramVariable),	280	in	
HybridConvectionTerm	(class fipy.terms.hybridConvectionTerm),	227	in	
I				
ICPreconditioner	(class fipy.solvers.trilinos.preconditioners.icPreconditioner),	211	in	
IIIConditionedPreconditionerWarning,	215			
ImplicitSourceTerm, 89, 113, 116				
ImplicitSourceTerm	(class fipy.terms.implicitSourceTerm),	228	in	
J				
JacobiPreconditioner	(class fipy.solvers.trilinos.preconditioners.jacobiPreconditioner),	211	in	
justResidualVector() (fipy.terms.term.Term method),	230			
L				
L1error() (in module fipy.steppers),	217			
L1norm() (in module fipy.tools.numerix),	258			
L2error() (in module fipy.steppers),	217			
L2norm() (in module fipy.tools.numerix),	258			
LD_LIBRARY_PATH, 14				
LinearBicgstabSolver	(class fipy.solvers.trilinos.linearBicgstabSolver),	212	in	
LinearCGSSolver	(class fipy.solvers.pysparse.linearCGSSolver),	209	in	

LinearCGSSolver	(class fipy.solvers.trilinos.linearCGSSolver),	212	in	Matplotlib2DViewer	(class fipy.viewers.matplotlibViewer),	303	in
LinearGMRESSolver	(class fipy.solvers.pysparse.linearGMRESSolver),	209	in	Matplotlib2DViewer	(class fipy.viewers.matplotlibViewer.matplotlib2DViewer),	306	in
LinearGMRESSolver	(class fipy.solvers.trilinos.linearGMRESSolver),	213	in	MatplotlibSparseMatrixViewer	(class fipy.viewers.matplotlibViewer.matplotlibSparseMatrixViewer),	307	in
LinearJORSolver	(class fipy.solvers.pysparse.linearJORSolver),	209	in	MatplotlibSurfactantViewer	(class fipy.models.levelSet.surfactant.matplotlibSurfactantViewer),	202	in
LinearLUSolver	, 152		in	MatplotlibVectorViewer	(class fipy.viewers.matplotlibViewer),	303	in
LinearLUSolver	(class fipy.solvers.pysparse.linearLUSolver),	210	in	MatplotlibVectorViewer	(class fipy.viewers.matplotlibViewer.matplotlibVectorViewer),	308	in
LinearPCGSolver	(class fipy.solvers.pysparse.linearPCGSolver),	210	in	MatplotlibViewer()	(in fipy.viewers.matplotlibViewer),	301	module
LinearPCGSolver	(class fipy.solvers.trilinos.linearPCGSolver),	213	in	MatrixIllConditionedWarning	, 215		
LINFerror()	(in module fipy.steppers),	217		max()	(fipy.variables.variable.Variable method),	288	
LINFnorm()	(in module fipy.tools.numerix),	258		MaximumIterationWarning	, 215		
loadtxt	, 114, 117, 139			MayaVi	, 45		
log	, 98, 104			Mayavi	, 45		
log()	(fipy.tools.dimensions.physicalField.PhysicalField method),	250		MayaviClient	(class fipy.viewers.mayaviViewer.mayaviClient),	309	in
log()	(fipy.variables.variable.Variable method),	288		MayaviDaemon	(class fipy.viewers.mayaviViewer.mayaviDaemon),	310	in
log()	(in module fipy.tools.numerix),	263		MayaviSurfactantViewer	, 138		
log10()	(fipy.tools.dimensions.physicalField.PhysicalField method),	250		MayaviSurfactantViewer	(class fipy.models.levelSet.surfactant.mayaviSurfactantViewer),	203	in
log10()	(fipy.variables.variable.Variable method),	288		MemoryHighWaterThread	(class fipy.tools.memoryLogger),	257	in
log10()	(in module fipy.tools.numerix),	263		MemoryLogger	(class in fipy.tools.memoryLogger),	257	
M				Mesh	(class in fipy.meshes.common.mesh),	169	
main	(in module fipy.tests.testProgram),	239		Mesh	(class in fipy.meshes.numMesh.mesh),	178	
main()	(in module fipy.viewers.mayaviViewer.mayaviDaemon),	310		Mesh	(class in fipy.meshes.pyMesh.mesh),	187	
make()	(in module fipy.viewers),	313		Mesh1D	(class in fipy.meshes.numMesh.mesh1D),	178	
make()	(in module fipy.viewers.viewer),	315		Mesh2D	(class in fipy.meshes.numMesh.mesh2D),	178	
Matplotlib	, 45			MeshAdditionError	, 178		
Matplotlib1DViewer	(class fipy.viewers.matplotlibViewer),	301	in	MeshDimensionError	, 312		
Matplotlib1DViewer	(class fipy.viewers.matplotlibViewer.matplotlib1DViewer),	305	in	MeshExportError	, 173		
Matplotlib2DGridContourViewer	(class fipy.viewers.matplotlibViewer),	302	in	MeshImportError	, 176		
Matplotlib2DGridContourViewer	(class fipy.viewers.matplotlibViewer.matplotlib2DGridContourViewer),	305	in	min()	(fipy.variables.variable.Variable method),	288	
Matplotlib2DGridViewer	(class fipy.viewers.matplotlibViewer),	302	in	ModularVariable	, 115		
Matplotlib2DGridViewer	(class fipy.viewers.matplotlibViewer.matplotlib2DGridViewer),	306	in	ModularVariable	(class fipy.variables.modularVariable),	281	in

MshFile (class in `fipy.meshes.numMesh.gmshImport`), 176
 MultilevelDDMLPreconditioner (class in `fipy.solvers.trilinos.preconditioners.multilevelDDM`), 211
 MultilevelDDPreconditioner (class in `fipy.solvers.trilinos.preconditioners.multilevelDDP`), 211
 MultilevelNSSAPreconditioner (class in `fipy.solvers.trilinos.preconditioners.multilevelNSSP`), 211
 MultilevelSAPreconditioner (class in `fipy.solvers.trilinos.preconditioners.multilevelSAP`), 212
`multiply()` (`fipy.tools.dimensions.physicalField.PhysicalField` method), 250
`MultiViewer` (class in `fipy.viewers.multiViewer`), 313

N

`name()` (`fipy.tools.dimensions.physicalField.PhysicalUnit` method), 255
`NoiseVariable` (class in `fipy.variables.noiseVariable`), 282
`NthOrderBoundaryCondition`, 76
`NthOrderBoundaryCondition` (class in `fipy.boundaryConditions.nthOrderBoundaryCondition`), 166
`NthOrderDiffusionTerm` (class in `fipy.terms.nthOrderDiffusionTerm`), 228
`numarray`, 45
`Numeric`, 45
`numerix`, 152
`NumPy`, 45

O

`obj2sctype()` (in module `fipy.tools.numerix`), 263
`object`
`fipy.variables.cellVariable.CellVariable`, 67
`fipy.viewers.tsvViewer.TSVViewer`, 69
`ones()` (in module `fipy.tools.numerix`), 264

P

`Parallel` (class in `fipy.tools`), 256
`parallelRandom()` (`fipy.variables.gaussianNoiseVariable.GaussianNoiseVariable` method), 280
`parallelRandom()` (`fipy.variables.noiseVariable.NoiseVariable` method), 282
`parse()` (in module `fipy.tools.parser`), 266
`parse_command_line()` (`fipy.viewers.mayaviViewer.mayaviDaemon` method), 310
`PeriodicGrid1D` (class in `fipy.meshes.numMesh.periodicGrid1D`), 179

`PeriodicGrid2D` (class in `fipy.meshes.numMesh.periodicGrid2D`), 180
`PeriodicGrid2DLeftRight` (class in `fipy.meshes.numMesh.periodicGrid2D`), 181
`PeriodicGrid2DTopBottom` (class in `fipy.meshes.numMesh.periodicGrid2D`), 181
`PhysicalField` (class in `fipy.tools.dimensions.physicalField`), 243
`PhysicalUnit` (class in `fipy.tools.dimensions.physicalField`), 254
`pi`, 108, 114, 115
`PIDStepper` (class in `fipy.steppers.pidStepper`), 218
`plot()` (`fipy.models.levelSet.surfactant.mayaviSurfactantViewer.MayaviSurfa` method), 204
`plot()` (`fipy.viewers.gistViewer.Gist1DViewer` method), 292
`plot()` (`fipy.viewers.gistViewer.gist1DViewer.Gist1DViewer` method), 294
`plot()` (`fipy.viewers.gistViewer.Gist2DViewer` method), 293
`plot()` (`fipy.viewers.gistViewer.gist2DViewer.Gist2DViewer` method), 295
`plot()` (`fipy.viewers.gistViewer.GistVectorViewer` method), 294
`plot()` (`fipy.viewers.gistViewer.gistVectorViewer.GistVectorViewer` method), 297
`plot()` (`fipy.viewers.matplotlibViewer.matplotlibSparseMatrixViewer.Matpl` method), 307
`plot()` (`fipy.viewers.mayaviViewer.mayaviClient.MayaviClient` method), 309
`plot()` (`fipy.viewers.multiViewer.MultiViewer` method), 313
`plot()` (`fipy.viewers.tsvViewer.TSVViewer` method), 314
`plotMesh()` (`fipy.viewers.gistViewer.Gist2DViewer` method), 293
`plotMesh()` (`fipy.viewers.gistViewer.gist2DViewer.Gist2DViewer` method), 295
`poll_file()` (`fipy.viewers.mayaviViewer.mayaviDaemon.MayaviDaemon` method), 310
`PowerLawConvectionTerm`, 97, 105
`PowerLawConvectionTerm` (class in `fipy.terms.powerLawConvectionTerm`), 228
`pprint_val()` (`fipy.viewers.matplotlibViewer.matplotlibSparseMatrixViewer.` method), 307
`Preconditioner` (class in `fipy.solvers.trilinos.preconditioners.preconditioner`), 212
`PreconditionerNotPositiveDefiniteWarning`, 215
`PreconditionerWarning`, 215
`PRINT()` (in module `fipy.tools.debug`), 256
`prune()` (in module `fipy.tools.vector`), 267

PseudoRKQSStepper (class in `fipy.steppers.pseudoRKQSStepper`), 219
 put() (`fipy.tools.dimensions.physicalField.PhysicalField` method), 251
 put() (`fipy.variables.variable.Variable` method), 288
 put() (in module `fipy.tools.numerix`), 264
 putAdd() (in module `fipy.tools.vector`), 267
 Pygist, 45
 Pyrex, 45
 PySparse, 45
 PysparseSolver (class in `fipy.solvers.pysparse.pysparseSolver`), 210
 Python, 45
 PYTHONPATH, 8, 9, 11, 13

Q
 quiver() (`fipy.viewers.matplotlibViewer.MatplotlibVectorViewer` method), 304
 quiver() (`fipy.viewers.matplotlibViewer.matplotlibVectorViewer` method), 309

R
 random() (`fipy.variables.betanoiseVariable.BetaNoiseVariable` method), 270
 random() (`fipy.variables.exponentialNoiseVariable.ExponentialNoiseVariable` method), 276
 random() (`fipy.variables.gammaNoiseVariable.GammaNoiseVariable` method), 278
 random() (`fipy.variables.noiseVariable.NoiseVariable` method), 282
 random() (`fipy.variables.uniformNoiseVariable.UniformNoiseVariable` method), 284
 rank() (in module `fipy.tools.numerix`), 264
 read() (in module `fipy.tools.dump`), 256
 remove() (`fipy.meshes.numMesh.gmshImport.MshFile` method), 176
 reshape() (`fipy.tools.dimensions.physicalField.PhysicalField` method), 251
 reshape() (`fipy.variables.variable.Variable` method), 288
 reshape() (in module `fipy.tools.numerix`), 264
 residual() (in module `fipy.steppers`), 218
 residualVectorAndNorm() (`fipy.terms.term.Term` method), 230
 run() (`fipy.tools.memoryLogger.MemoryHighWaterThread` method), 257
 run() (`fipy.viewers.mayaviViewer.mayaviDaemon.MayaviDaemon` method), 310
 runGold, 128
 runLeveler, 130
 runSimpleTrenchSystem, 125

S
 save() (`fipy.tools.vitals.Vitals` method), 267
 ScalarQuantityOutOfRangeWarning, 215

ScharfetterGummelFaceVariable (class in `fipy.variables.scharfetterGummelFaceVariable`), 283
 ScientificPython, 45
 SciPy, 45, 92, 99
 scipy module, 70
 scramble() (`fipy.variables.noiseVariable.NoiseVariable` method), 282
 Serial (class in `fipy.tools`), 256
 setLimits() (in `fipy.viewers.multiViewer.MultiViewer` method), 313
 setName() (`fipy.tools.dimensions.physicalField.PhysicalUnit` method), 256
 setName() (`fipy.variables.variable.Variable` method), 288
 setScale() (`fipy.meshes.common.mesh.Mesh` method), 171
 setScale() (`fipy.meshes.pyMesh.mesh.Mesh` method), 187
 setUnit() (`fipy.tools.dimensions.physicalField.PhysicalField` method), 251
 setUnit() (`fipy.variables.variable.Variable` method), 288
 setup_source() (`fipy.viewers.mayaviViewer.mayaviDaemon.MayaviDaemon` method), 310
 setValue() (`fipy.variables.cellVariable.CellVariable` method), 274
 setValue() (`fipy.variables.faceVariable.FaceVariable` method), 277
 setValue() (`fipy.variables.variable.Variable` method), 288
 shape (`fipy.tools.dimensions.physicalField.PhysicalField` attribute), 251
 shape (`fipy.variables.variable.Variable` attribute), 288
 sign() (`fipy.tools.dimensions.physicalField.PhysicalField` method), 251
 sign() (`fipy.variables.variable.Variable` method), 289
 sign() (in module `fipy.tools.numerix`), 264
 SignedLogFormatter (class in `fipy.viewers.matplotlibViewer.matplotlibSparseMatrixViewer`), 307
 SignedLogLocator (class in `fipy.viewers.matplotlibViewer.matplotlibSparseMatrixViewer`), 307
 sin() (`fipy.tools.dimensions.physicalField.PhysicalField` method), 252
 sin() (`fipy.variables.variable.Variable` method), 289
 sin() (in module `fipy.tools.numerix`), 264
 sinh() (`fipy.tools.dimensions.physicalField.PhysicalField` method), 252
 sinh() (`fipy.variables.variable.Variable` method), 289
 sinh() (in module `fipy.tools.numerix`), 264
 SkewedGrid2D (class in `fipy.meshes.numMesh.skewedGrid2D`), 181
 solve, 99
 solve() (`fipy.models.levelSet.surfactant.adsorbingSurfactantEquation.AdSOR` method), 201

solve() (fipy.models.levelSet.surfactant.surfactantEquation.SurfactantEquation method), 205

solve() (fipy.terms.term.Term method), 231

Solver (class in fipy.solvers.solver), 215

SolverConvergenceWarning, 215

SourceTerm (class in fipy.terms.sourceTerm), 229

Sphinx, 45

$\sqrt{}$, 87, 114, 135

\arcsin

\cos , 70

$\sqrt{}$ (fipy.tools.dimensions.physicalField.PhysicalField method), 252

$\sqrt{}$ (fipy.variables.variable.Variable method), 289

$\sqrt{}$ (in module fipy.tools.numerix), 265

$\sqrt{\cdot}$ (in module fipy.tools.numerix), 265

StagnatedSolverWarning, 215

start() (fipy.tools.memoryLogger.MemoryLogger method), 257

SteadyConvectionDiffusionScEquation, 152

step() (fipy.steppers stepper.Stepper method), 219

Stepper (class in fipy.steppers stepper), 219

stop() (fipy.tools.memoryLogger.MemoryHighWaterThread method), 257

stop() (fipy.tools.memoryLogger.MemoryLogger method), 257

subtract() (fipy.tools.dimensions.physicalField.PhysicalField method), 252

successFn() (fipy.steppers stepper.Stepper static method), 219

sum() (fipy.tools.dimensions.physicalField.PhysicalField method), 253

sum() (fipy.variables.variable.Variable method), 289

sum() (in module fipy.tools.numerix), 265

SurfactantEquation (class in fipy.models.levelSet.surfactant.surfactantEquation), 205

SurfactantVariable, 136

SurfactantVariable (class in fipy.models.levelSet.surfactant.surfactantVariable), 206

svn() (fipy.tools.vitals.Vitals method), 267

svnemd() (fipy.tools.vitals.Vitals method), 267

sweep, 91, 99, 149

sweep() (fipy.models.levelSet.surfactant.adsorbingSurfactantEquation method), 201

sweep() (fipy.models.levelSet.surfactant.surfactantEquation.SurfactantEquation method), 205

sweep() (fipy.terms.term.Term method), 231

sweepFn() (fipy.steppers stepper.Stepper static method), 219

sweepMonotonic() (in module fipy.steppers), 218

T

take, 107

take() (fipy.tools.dimensions.physicalField.PhysicalField method), 253

take() (fipy.variables.variable.Variable method), 289

take() (in module fipy.tools.numerix), 265

\tan , 108

$\tan{}$ (fipy.tools.dimensions.physicalField.PhysicalField method), 253

$\tan{}$ (fipy.variables.variable.Variable method), 289

$\tan{}$ (in module fipy.tools.numerix), 265

\tanh , 87

$\tanh{}$ (fipy.tools.dimensions.physicalField.PhysicalField method), 253

$\tanh{}$ (fipy.variables.variable.Variable method), 289

$\tanh{}$ (in module fipy.tools.numerix), 265

Term (class in fipy.terms.term), 230

toString() (fipy.tools.dimensions.physicalField.PhysicalField method), 253

toString() (fipy.variables.variable.Variable method), 289

toString() (in module fipy.tools.numerix), 266

TransientTerm, 52, 88, 113, 116

TransientTerm (class in fipy.terms.transientTerm), 232

transpose() (fipy.variables.variable.Variable method), 289

TrenchMesh (class in fipy.models.levelSet.electroChem.gapFillMesh), 195

Tri2D (class in fipy.meshes.numMesh.tri2D), 182

Trilinos (class in fipy.solvers.trilinos.Trilinos), 45

TrilinosAztecOOsolver (class in fipy.solvers.trilinos.trilinosAztecOOsolver), 213

TrilinosMLTest (class in fipy.solvers.trilinos.trilinosMLTest), 214

TrilinosSolver (class in fipy.solvers.trilinos.trilinosSolver), 214

TSVViewer (class in fipy.viewers.tsvViewer), 314

tupleToXML() (fipy.tools.vitals.Vitals method), 267

U

UniformGrid1D (class in fipy.meshes.numMesh.uniformGrid1D), 182

UniformGrid2D (class in fipy.meshes.numMesh.uniformGrid2D), 182

UniformGrid3D (class in fipy.meshes.numMesh.uniformGrid3D), 183

UniformNoiseVariable (class in fipy.variables.uniformNoiseVariable), 283

update_pipeline (fipy.viewers.mayaviViewer.mayaviDaemon.MayaviDaemon method), 310

updateOld() (fipy.variables.cellVariable.CellVariable method), 274

updateOld() (fipy.variables.modularVariable.ModularVariable method), 282

UpwindConvectionTerm (class in fipy.terms.upwindConvectionTerm), 233

V

VanLeerConvectionTerm (class in `fipy.terms.vanLeerConvectionTerm`), [234](#)
Variable, [91](#), [94](#)
Variable (class in `fipy.variables.variable`), [284](#)
Vertex (class in `fipy.meshes.pyMesh.vertex`), [187](#)
`view_data()` (`fipy.viewers.mayaviViewer.mayaviDaemon.MayaviDaemon` method), [310](#)
`Viewer()` (in module `fipy.viewers`), [312](#)
viewers, [155](#)
 module, [80](#), [81](#)
Vitals (class in `fipy.tools.vitals`), [267](#)
VTKCellViewer (class in `fipy.viewers.vtkViewer`), [311](#)
VTKCellViewer (class in `fipy.viewers.vtkViewer.vtkCellViewer`), [312](#)
VTKFaceViewer (class in `fipy.viewers.vtkViewer`), [311](#)
VTKFaceViewer (class in `fipy.viewers.vtkViewer.vtkFaceViewer`), [312](#)
`VTKViewer()` (in module `fipy.viewers.vtkViewer`), [311](#)

W

`write()` (in module `fipy.tools.dump`), [256](#)

Z

`zeros()` (in module `fipy.tools.numerix`), [266](#)