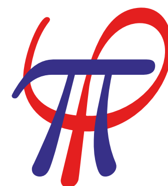


FiPy Manual

Release 3.0

Jonathan E. Guyer
Daniel Wheeler
James A. Warren

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

August 16, 2012

NIST
National Institute of
Standards and Technology
U.S. Department of Commerce

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 3.0? 3
- 1.3 Download and Installation 5
- 1.4 Support 5
- 1.5 Conventions and Notation 6

2 Installation 9
- 2.1 Installing Python Packages 9
- 2.2 Obtaining FiPy 9
- 2.3 Installing FiPy 10
- 2.4 Required Packages 10
- 2.5 Optional Packages 11
- 2.6 Platform-Specific Instructions 12
- 2.7 Development Environment 13
- 2.8 SVN usage 13

3 Solvers 15
- 3.1 PySparse 15
- 3.2 SciPy 16
- 3.3 PyAMG 16
- 3.4 Trilinos 16

4 Viewers 19
- 4.1 Matplotlib 19
- 4.2 Mayavi 19

5 Using FiPy 21
- 5.1 Testing FiPy 21
- 5.2 Command-line Flags and Environment Variables 22
- 5.3 Solving in Parallel 23
- 5.4 Meshing with Gmsh 25
- 5.5 Coupled and Vector Equations 25
- 5.6 Boundary Conditions 26
- 5.7 Running under Python 3 28
- 5.8 Manual 28

6 Theoretical and Numerical Background 29
- 6.1 General Conservation Equation 29

6.2 Finite Volume Method . . . 30
6.3 Discretization . . . 30
6.4 Linear Equations . . . 34
6.5 Numerical Schemes . . . 34

7 Design and Implementation **37**
7.1 Design . . . 37
7.2 Implementation . . . 39

8 Frequently Asked Questions **41**
8.1 How do I represent an equation in FiPy? . . . 41
8.2 How can I see what I'm doing? . . . 43
8.3 Iterations, timesteps, and sweeps? Oh, my! . . . 44
8.4 Why the distinction between `CellVariable` and `FaceVariable` coefficients? . . . 46
8.5 How do I represent boundary conditions? . . . 47
8.6 What does this error message mean? . . . 47
8.7 How do I change FiPy's default behavior? . . . 47
8.8 How can I tell if I'm running in parallel? . . . 47
8.9 Why don't my scripts work anymore? . . . 48
8.10 What if my question isn't answered here? . . . 48

9 Glossary **49**

II Examples **51**

10 Diffusion Examples **55**
10.1 examples.diffusion.mesh1D . . . 55
10.2 examples.diffusion.coupled . . . 67
10.3 examples.diffusion.mesh20x20 . . . 69
10.4 examples.diffusion.circle . . . 71
10.5 examples.diffusion.electrostatics . . . 76
10.6 examples.diffusion.nthOrder.input4thOrder1D . . . 80
10.7 examples.diffusion.anisotropy . . . 82

11 Convection Examples **85**
11.1 examples.convection.exponential1D.mesh1D . . . 85
11.2 examples.convection.exponential1DSource.mesh1D . . . 86
11.3 examples.convection.robin . . . 87
11.4 examples.convection.source . . . 89

12 Phase Field Examples **91**
12.1 examples.phase.simple . . . 91
12.2 examples.phase.binaryCoupled . . . 99
12.3 examples.phase.quaternary . . . 107
12.4 examples.phase.anisotropy . . . 113
12.5 examples.phase.impingement.mesh40x1 . . . 117
12.6 examples.phase.impingement.mesh20x20 . . . 120
12.7 examples.phase.polyxtal . . . 123
12.8 examples.phase.polyxtalCoupled . . . 129

13 Level Set Examples **135**
13.1 examples.levelSet.distanceFunction.mesh1D . . . 135
13.2 examples.levelSet.distanceFunction.circle . . . 136
13.3 examples.levelSet.advection.mesh1D . . . 137

13.4 examples.levelSet.advection.circle . . . 138
13.5 Superconformal Electrodeposition Examples . . . 140
13.6 examples.levelSet.electroChem.simpleTrenchSystem . . . 141
13.7 examples.levelSet.electroChem.gold . . . 144
13.8 examples.levelSet.electroChem.leveler . . . 145
13.9 examples.levelSet.electroChem.howToWriteAScript . . . 149

14 Cahn Hilliard Examples **157**
14.1 examples.cahnHilliard.mesh2DCoupled . . . 157
14.2 examples.cahnHilliard.sphere . . . 160

15 Fluid Flow Examples **163**
15.1 examples.flow.stokesCavity . . . 163

16 Reactive Wetting Examples **169**
16.1 examples.reactiveWetting.liquidVapor1D . . . 169

17 Updating FiPy **175**
17.1 examples.updating.update2_0to3_0 . . . 175
17.2 examples.updating.update1_0to2_0 . . . 176
17.3 examples.updating.update0_1to1_0 . . . 180

III fipy Package Documentation **185**

18 How to Read the Modules Documentation **187**
18.1 subpackage Package . . . 187

19 boundaryConditions Package **191**
19.1 boundaryConditions Package . . . 191
19.2 boundaryCondition Module . . . 192
19.3 constraint Module . . . 192
19.4 fixedFlux Module . . . 192
19.5 fixedValue Module . . . 192
19.6 nthOrderBoundaryCondition Module . . . 193
19.7 test Module . . . 193

20 matrices Package **195**
20.1 offsetSparseMatrix Module . . . 195
20.2 pysparseMatrix Module . . . 195
20.3 scipyMatrix Module . . . 195
20.4 sparseMatrix Module . . . 195
20.5 test Module . . . 195
20.6 trilinosMatrix Module . . . 195

21 meshes Package **197**
21.1 meshes Package . . . 197
21.2 abstractMesh Module . . . 205
21.3 cylindricalGrid1D Module . . . 209
21.4 cylindricalGrid2D Module . . . 210
21.5 cylindricalUniformGrid1D Module . . . 210
21.6 cylindricalUniformGrid2D Module . . . 211
21.7 factoryMeshes Module . . . 211
21.8 gmshMesh Module . . . 213
21.9 grid1D Module . . . 218

21.10 grid2D Module . . . 218
21.11 grid3D Module . . . 219
21.12 mesh Module . . . 219
21.13 mesh1D Module . . . 219
21.14 mesh2D Module . . . 220
21.15 periodicGrid1D Module . . . 220
21.16 periodicGrid2D Module . . . 221
21.17 skewedGrid2D Module . . . 222
21.18 test Module . . . 222
21.19 tri2D Module . . . 223
21.20 uniformGrid Module . . . 223
21.21 uniformGrid1D Module . . . 223
21.22 uniformGrid2D Module . . . 224
21.23 uniformGrid3D Module . . . 224
21.24 Subpackages . . . 225

22 models Package **227**
22.1 models Package . . . 227
22.2 test Module . . . 242
22.3 Subpackages . . . 242

23 solvers Package **293**
23.1 solvers Package . . . 293
23.2 pysparseMatrixSolver Module . . . 295
23.3 solver Module . . . 295
23.4 test Module . . . 296
23.5 Subpackages . . . 296

24 steppers Package **317**
24.1 steppers Package . . . 317
24.2 pidStepper Module . . . 318
24.3 pseudoRKQSStepper Module . . . 318
24.4 stepper Module . . . 318

25 terms Package **319**
25.1 terms Package . . . 319
25.2 abstractBinaryTerm Module . . . 329
25.3 abstractConvectionTerm Module . . . 329
25.4 abstractDiffusionTerm Module . . . 329
25.5 abstractUpwindConvectionTerm Module . . . 329
25.6 asymmetricConvectionTerm Module . . . 329
25.7 binaryTerm Module . . . 329
25.8 cellTerm Module . . . 329
25.9 centralDiffConvectionTerm Module . . . 329
25.10 coupledBinaryTerm Module . . . 330
25.11 diffusionTerm Module . . . 330
25.12 diffusionTermCorrection Module . . . 331
25.13 diffusionTermNoCorrection Module . . . 331
25.14 explicitDiffusionTerm Module . . . 331
25.15 explicitSourceTerm Module . . . 331
25.16 explicitUpwindConvectionTerm Module . . . 331
25.17 exponentialConvectionTerm Module . . . 332
25.18 faceTerm Module . . . 334
25.19 hybridConvectionTerm Module . . . 334
25.20 implicitDiffusionTerm Module . . . 335

25.21 implicitSourceTerm Module . . . 335
25.22 nonDiffusionTerm Module . . . 335
25.23 powerLawConvectionTerm Module . . . 335
25.24 residualTerm Module . . . 336
25.25 sourceTerm Module . . . 337
25.26 term Module . . . 337
25.27 test Module . . . 339
25.28 transientTerm Module . . . 339
25.29 unaryTerm Module . . . 340
25.30 upwindConvectionTerm Module . . . 340
25.31 vanLeerConvectionTerm Module . . . 342

26 tests Package **343**
26.1 tests Package . . . 343
26.2 doctestPlus Module . . . 343
26.3 lateImportTest Module . . . 344
26.4 testBase Module . . . 344
26.5 testClass Module . . . 344
26.6 testProgram Module . . . 344

27 tools Package **345**
27.1 tools Package . . . 345
27.2 copy_script Module . . . 355
27.3 debug Module . . . 355
27.4 decorators Module . . . 355
27.5 dump Module . . . 356
27.6 inline Module . . . 357
27.7 numerix Module . . . 357
27.8 parser Module . . . 364
27.9 test Module . . . 364
27.10 vector Module . . . 364
27.11 vitals Module . . . 365
27.12 Subpackages . . . 365

28 variables Package **383**
28.1 variables Package . . . 383
28.2 addOverFacesVariable Module . . . 403
28.3 arithmeticCellToFaceVariable Module . . . 403
28.4 betaNoiseVariable Module . . . 403
28.5 binaryOperatorVariable Module . . . 405
28.6 cellToFaceVariable Module . . . 405
28.7 cellVariable Module . . . 405
28.8 cellVolumeAverageVariable Module . . . 411
28.9 constant Module . . . 411
28.10 constraintMask Module . . . 411
28.11 coupledCellVariable Module . . . 411
28.12 exponentialNoiseVariable Module . . . 411
28.13 faceGradContributionsVariable Module . . . 412
28.14 faceGradVariable Module . . . 412
28.15 faceVariable Module . . . 412
28.16 gammaNoiseVariable Module . . . 413
28.17 gaussCellGradVariable Module . . . 415
28.18 gaussianNoiseVariable Module . . . 415
28.19 harmonicCellToFaceVariable Module . . . 417

28.20 histogramVariable Module . . . 417
28.21 leastSquaresCellGradVariable Module . . . 417
28.22 meshVariable Module . . . 417
28.23 minmodCellToFaceVariable Module . . . 417
28.24 modCellGradVariable Module . . . 417
28.25 modCellToFaceVariable Module . . . 417
28.26 modFaceGradVariable Module . . . 417
28.27 modPhysicalField Module . . . 417
28.28 modularVariable Module . . . 417
28.29 noiseVariable Module . . . 418
28.30 operatorVariable Module . . . 419
28.31 scharfetterGummelFaceVariable Module . . . 419
28.32 test Module . . . 419
28.33 unaryOperatorVariable Module . . . 419
28.34 uniformNoiseVariable Module . . . 419
28.35 variable Module . . . 420

29 viewers Package 429
29.1 viewers Package . . . 429
29.2 multiViewer Module . . . 439
29.3 test Module . . . 439
29.4 testinteractive Module . . . 439
29.5 tsvViewer Module . . . 440
29.6 viewer Module . . . 441
29.7 Subpackages . . . 442

Bibliography 463

Python Module Index 465

Index 471

Part I

Introduction

Chapter 1

Overview

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

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

The *FiPy* framework includes terms for transient diffusion, convection and standard sources, enabling the solution of arbitrary combinations of coupled elliptic, hyperbolic and parabolic PDEs. Currently implemented models include phase field [BoettingerReview:2002] [ChenReview:2002] [McFaddenReview:2002] treatments of polycrystalline, dendritic, and electrochemical phase transformations, as well as drug eluting stents [Saylor:2011p2794], reactive wetting [PhysRevE.82.051601], photovoltaics [Hangarter:2011p2795] and 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*, *Using FiPy* and *Frequently Asked Questions*, as well as `examples.diffusion.mesh1D`.

1.2 What's new in version 3.0?

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

The significant changes since version 2.1 are:

- *Coupled and Vector Equations* are now supported.
- A more robust mechanism for specifying *Boundary Conditions* is now used.
- Most `Mesh`es can be partitioned by *Meshing with Gmsh*.
- *PyAMG* and *SciPy* have been added to the *Solvers*.
- FiPy is capable of *Running under Python 3*.

- "getter" and "setter" methods have been pervasively changed to Python properties.
- The test suite now runs much faster.
- Tests can now be run on a full install using *fipy.test()*.
- The functions of the `numerix` module are no longer included in the `fipy` namespace. See `examples.updating.update2_0to3_0` for details.
- Equations containing a `TransientTerm`, must specify the timestep by passing a `dt=` argument when calling `solve()` or `sweep()`.

Tickets fixed in this release:

```
45  Navier Stokes
85  CellVariable hasOld() should set self.old
101 Grids should take Lx, Ly, Lz arguments
145 tests should be run with fipy.tests()
177 remove ones and zeros from numerix.py
178 Default time steps should be infinite
291 term multiplication changes result
296 FAQ gives bad guidance for anisotropic diffusion
297 Use physical velocity in the manual/FAQ
298 mesh manipulation of periodic meshes leads to errors
299 Give helpfull error on - or / of meshes
301 wrong cell to cell normal in periodic meshes
302 gnuplot1d gives error on plot of facevariable
309 pypi is failing
312 Fresh FiPy gives ""ImportError: No viewers found"""
314 Absence of enthought.tvtk causes test failures
319 mesh in FiPy name space
324 --pysparse configuration should never attempt MPI imports
327 factoryMeshes.py not up to date with respect to keyword arguments
331 changed constraints don't propagate
332 anisotropic diffusion and constraints don't mix
333 `--Trilinos --no-pysparse` uses PySparse?!?
336 Profile and merge reconstrain branch
339 close out reconstrain branch
341 Fix fipy.terms._BinaryTerm test failure in parallel
343 diffusionTerm(var=var1).solver(var=var0) should fail sensibly
346 TeX is wrong in examples.phase.quaternary
348 Include Benny's improved interpolation patch
354 GmshExport is not tested and does not work
355 Introduce mesh.x as shorthand for mesh.cellCenters[0] etc
356 GmshImport should support all element types
357 GmshImport should read element colors
363 Reduce the run times for chemotaxis tests
366 tests take *too* long!!!
369 Make DiffusionTermNoCorrection the default
370 Epetra Norm2 failure in parallel
373 remove deprecated `steps=` from Solver
376 remove deprecated `diffusionTerm=` argument to ConvectionTerm
377 remove deprecated `NthOrderDiffusionTerm`
380 remove deprecated Variable.transpose()
381 remove deprecated viewers.make()
382 get running in Py3k
384 gmsh importer and gmsh tests don't clean up after themselves
385 `diffusionTerm._test()` requires PySparse
390 Improve test reporting to avoid inconsequential buildbot failures
391 efficiency_test chokes on liquidVapor2D.py
```

```
393 two `--scipy` failures
395 `--pysparse --inline` failures
417 Memory consumption growth with repeated meshing, especially with Gmsh
418 Viewers not working when plotting meshes with zero cells in parallel
419 examples/cahnHilliard/mesh2D.py broken with --trilinos
420 Epetra.PyComm() broken on Debian
421 cellVariable.min() broken in parallel
426 Add in parallel buildbot testing on more than 2 processors
427 Slow PyAMG solutions
434 Gmsh I/O
438 changes to gmshImport.py caused --inline problems
439 gmshImport tests fail on Windows due to shared file
441 Explicit convetion terms should fail when the equation has no TransientTerm (dt=None)
445 getFaceCenters() should return a FaceVariable
446 constraining values with ImplictSourceTerm not documented?
448 Gmsh2D does not respect background mesh
452 Gmsh background mesh doesn't work in parallel
453 faceValue as FaceCenters gives inline failures
454 Py3k and Windows test failures
```

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

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: `fipy-request@nist.gov`

Subject: *(optional)*

Body: `subscribe`

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

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

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

List Archive

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 **tcsh** or **bash**.

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

```
$ like this
```

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

Text blocks of the form:

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

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

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

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

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

will correspond to, *e.g.*:

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

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

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

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

We may at times use a

Note: to indicate something that may be of interest

or a

Warning: to indicate something that could cause serious problems.

Chapter 2

Installation

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

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

2.1 Installing Python Packages

In general, it is best to use the following order of precedence when installing packages:

- Use the operating system package manager, if possible.
- Use the pip installs python (*pip*) tool to obtain software from the Python Package Index (*PyPI*) repository:

  ```
  $ pip install package
  ```

  Warning: *pip* takes care of dependencies that are themselves *Python* packages. It does not deal with non-*Python* dependencies.

- Download the packages manually, unpack and run:

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

  Further information about each `setup.py` script is available by typing:

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

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.

2.2 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 on 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).

2.3 Installing FiPy

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

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

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

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

```
$ pip install fipy
```

or by unpacking the archive and typing:

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

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

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

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

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

2.4 Required Packages

2.4.1 Python

http://www.python.org/

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

```
$ python
Python 2.3 (#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 2.4.x of *Python*. See the specialized instructions if you wish to *RunUnderPython3*.

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

Enthought Python Distribution

http://www.enthought.com/epd

This installer provides a very large number of useful scientific packages for *Python*, including *NumPy*, *SciPy*, *Matplotlib*, *Mayavi*, and *IPython*, as well as a *Python* interpreter. Installers are available for Windows, Mac OS X and RedHat Linux, Solaris, Ubuntu Linux, and OpenSuSE Linux.

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

Python(x,y)

http://www.pythonxy.com/

Another comprehensive *Python* package installer for scientific applications, presently only available for Windows.

Attention: *PySparse* and *FiPy* are not presently included in python(x,y), so you will need to separately install them manually.

2.4.2 NumPy

http://numpy.scipy.org

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

2.5 Optional Packages

2.5.1 Gmsh

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

Gmsh is an application that allows the creation of irregular meshes.

2.5.2 SciPy

http://www.scipy.org/

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

2.6 Platform-Specific Instructions

FiPy is tested regularly on Mac OS X, Debian Linux, Ubuntu Linux, and Windows XP. We welcome reports of compatibility with other systems, particularly if any additional steps are necessary to install (see Miscellaneous Build Recipes for user contributed installation tips).

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.

2.6.1 Mac OS X Installation

There is no official package manager for Mac OS X, but there are several third-party package managers that provide many, but not all of *FiPy*'s *Required Packages* and *Optional Packages*. Options include

Fink is based on the Debian package management system. It installs all of its dependencies into `/sw`.

MacPorts is a package manager originally part of OpenDarwin. It installs all of its dependencies into `/opt`.

Homebrew is a recent, lightweight package manager based on Ruby scripts. It installs all of its dependencies into `/usr/local` (although it can be directed not to).

In addition, there is an *Enthought Python Distribution* installer for Mac OS X.

> **Attention:** *PySparse* and *FiPy* are not presently included in any of these package managers or installers, so you will need to separately install them manually.

We presently find that the combination of Homebrew and *pip* is a pretty straightforward way to get most of *FiPy*'s prerequesites. See the Miscellaneous Build Recipes for up-to-date directions.

2.6.2 Windows Installation

There is no official package manager for Windows, but the *Enthought Python Distribution* and *Python(x,y)* installers provide most of *FiPy*'s prerequisites.

> **Attention:** *PySparse* and *FiPy* are not presently included in EPD or python(x,y), so you will need to separately install them manually.

2.6.3 Ubuntu/Debian Installation

FiPy now has a *.deb* for Ubuntu/Debian systems that can be downloaded from <http://www.ctcms.nist.gov/fipy/download>. Simply run:

```
$ dpkg -i python-fipy_x.y.z-1_all.deb
```

to install. The *.deb* includes dependencies for all of the *Required Packages* and *Optional Packages*.

2.6.4 Miscellaneous Build Recipes

We often post miscellaneous installation instructions on the *FiPy* blog and wiki pages. The most useful of these include:

- Installing FiPy on Mac OS X using Homebrew
- Building a 64-bit scientific python environment for FiPy from source
- Installing FiPy with pip

Note: We encourange you to contribute your own build recipes on the wiki if they are significantly different.

2.7 Development Environment

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

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

The simplest way to use a *Python* package without installing it is to work in the base directory of the unpacked package and set the `PYTHONPATH` environment variable to "`.`". In order to work in an directory other than the package's base directory, the `PYTHONPATH` environment variable must be set to "`~/path/to/package`". This method of working is adequate for one package, but quickly becomes unmanageable with multiple *Python* packages. In order to manage multiple packages, it is better to choose a standard location other than the default installation path.

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 alternative 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`.

An alternative to setting the `PYTHONPATH` is to employ one of the utilities that manage packages and their dependencies independently of the system package manager and the system directories. These utilities include Stow, Virtualenv and zc.buildout, amongst others. Here we'll describe the use of Virtualenv, which we highly recommend.

2.7.1 Virtualenv

http://pypi.python.org/pypi/virtualenv

Virtualenv enables the installation of packages in multiple isolated environments. It organizes the installation of *Python* packages especially well and also provides a handy location for installing non-*Python* packages. In addition Virtualenv works seamlessly with the *PyPI* package manager (*pip*).

The utility of Virtualenv is significantly augmented with Virtualenvwrapper.

In general, the initial installation of Virtualenv and Virtualenvwrapper requires admin privileges, but thereafter, creating new virtual environments and installing packages into them does not require admin privileges.

2.8 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*.

2.8.1 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.

2.8.2 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 `<label>` can be obtained with:

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

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/<label> .
```

in the base directory of the working copy. The main tags (`<label>`) 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 3.0.

`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: 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://build.cmi.kent.edu:8010

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].

Chapter 3

Solvers

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

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

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

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

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

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

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

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

3.1 PySparse

http://pysparse.sourceforge.net

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

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

3.2 SciPy

http://www.scipy.org/

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

3.3 PyAMG

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

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

3.4 Trilinos

http://trilinos.sandia.gov

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

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

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

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

Trilinos requires cmake, *NumPy*, and swig. The following are the minimal steps to build and install *Trilinos* (with *PyTrilinos*) for *FiPy*:

```
$ cd trilinos-X.Y/
$ SOURCE_DIR=`pwd`
$ mkdir BUILD_DIR
$ cd BUILD_DIR
$ cmake \
>   -D CMAKE_BUILD_TYPE:STRING=RELEASE \
>   -D Trilinos_ENABLE_PyTrilinos:BOOL=ON \
>   -D BUILD_SHARED_LIBS:BOOL=ON \
>   -D Trilinos_ENABLE_ALL_OPTIONAL_PACKAGES:BOOL=ON \
>   -D TPL_ENABLE_MPI:BOOL=ON \
>   -D Trilinos_ENABLE_TESTS:BOOL=ON \
>   -D DART_TESTING_TIMEOUT:STRING=600 \
>   ${SOURCE_DIR}
$ make
$ make install
```

Depending on your platform, other options may be helpful or necessary; see 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 -D CMAKE_INSTALL_PREFIX:PATH=${*INSTALL_DIR*} and -D PyTrilinos_INSTALL_PREFIX:PATH=${*INSTALL_DIR* flags to the configure step. If *Trilinos* is installed in a non-standard 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`.

3.4.1 mpi4py

http://mpi4py.scipy.org/

For *Solving in Parallel*, *FiPy* requires mpi4py, in addition to *Trilinos*.

Chapter 4

Viewers

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

4.1 Matplotlib

http://matplotlib.sourceforge.net

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

4.2 Mayavi

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

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

Mayavi requires VTK, which can be difficult to build from source. *Mayavi* and VTK can be most easily obtained through

- the Ubuntu or Debian package managers
- the *Enthought Python Edition*
- *python(x,y)*
- the Homebrew package manager for Mac OS X (VTK only, not *Mayavi*)

Note: MayaVi 1 is no longer supported.

Chapter 5

Using FiPy

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

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

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

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

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

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

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

In order to customize the examples, or to develop your own scripts, some knowledge of Python syntax is required. We recommend you familiarize yourself with the excellent Python tutorial [PythonTutorial] or with Dive Into Python [DiveIntoPython].

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

5.1 Testing FiPy

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

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

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

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

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

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

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

We have a few known, intermittent failures:

#425 The test suite can freeze, usually in `examples.chemotaxis`, when running on multiple processors. This has never affected us in an actual parallel simulation, only in the test suite.

#430 When running in parallel, the tests for `_BinaryTerm` sometimes return one erroneous result. This is not reliably reproducible and doesn't seem to have an effect on actual simulations.

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

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

5.1.1 Parallel Tests

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

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

or:

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

5.2 Command-line Flags and Environment Variables

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

5.2.1 Command-line Flags

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

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

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

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

--pysparse
Forces the use of the *PySparse* solvers.

--trilinos
Forces the use of the *Trilinos* solvers, but uses *PySparse* to construct the matrices.

--no-pysparse
Forces the use of the *Trilinos* solvers without any use of *PySparse*.

--scipy
Forces the use of the *SciPy* solvers.

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

5.2.2 Environment Variables

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

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

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`", "`trilinos`", "`no-pysparse`", "`scipy`" and "`pyamg`".

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

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

FIPY_INCLUDE_NUMERIX_ALL
If present, causes the inclusion of all funcions and variables of the `numerix` module in the `fipy` namespace.

5.3 Solving in Parallel

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

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

Attention: *FiPy* requires *mpi4py* to work in parallel. See the *mpi4py* installation guide.

Note: Parallel efficiency is greatly improved by installing *PySparse* in addition to *Trilinos*. If *PySparse* is not installed be sure to use the `--no-pysparse` flag when running in parallel.

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

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

instead of:

```
$ python myScript.py
```

To confirm that *FiPy* and *Trilinos* are properly configured to solve in parallel, the easiest way to tell is to run one of the examples, e.g.,:

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

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

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

which should print out:

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

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

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

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

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

Note: *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.

5.4 Meshing with Gmsh

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

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

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

5.5 Coupled and Vector Equations

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

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

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

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

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

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

In this case the matrix system will have the form

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

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

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

5.6 Boundary Conditions

5.6.1 Applying fixed value (Dirichlet) boundary conditions

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

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

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

5.6.2 Applying fixed gradient boundary conditions (Neumann)

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

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

5.6.3 Applying fixed flux boundary conditions

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

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

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

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

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

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

5.6.4 Applying outlet or inlet boundary conditions

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

5.6.5 Applying spatially varying boundary conditions

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

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

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

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

then

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

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

5.6.6 Applying internal boundary conditions

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

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

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

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

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

One must be careful to distinguish between constraining internal cell values during the solve step and simply applying arbitrary constraints to a `CellVariable`. Applying a constraint,

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

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

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

Apply a constraint to the faces for a right side boundary condition (which works).

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

Create the equation with the source term constraint described above

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

and the expected value is obtained.

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

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

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

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

5.7 Running under Python 3

It is possible to run *FiPy* scripts under *Python 3*, but there is admittedly little advantage in doing so at this time. We still develop and use *FiPy* under *Python* 2.x. To use, you must first convert *FiPy*'s code to *Python 3* syntax. From within the main *FiPy* directory:

```
$ 2to3 --write .
$ 2to3 --write --doctests_only .
```

You can expect some harmless warnings from this conversion.

The minimal prerequisites are:

- *NumPy* version 1.5 or greater.
- *SciPy* version 0.9 or greater.
- *Matplotlib* version 1.2 or greater (this hasn't been released yet, and we haven't been able to successfully test the `matplotlibViewer` classes with their development code).

5.8 Manual

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

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

Note: This mechanism is intended primarily for the developers. At a minimum, you will need at least version 1.1.2 of Sphinx, plus all of its prerequisites.

Chapter 6

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

6.1 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

$$\frac{\partial \phi}{\partial t} = H(\phi, \lambda_i) \tag{6.1}$$

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 Schroedinger equation.

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

$$\underbrace{\frac{\partial (\rho \phi)}{\partial t}}_{\text{transient}} + \underbrace{\nabla \cdot (\vec{u} \phi)}_{\text{convection}} = \underbrace{\left[\nabla \cdot (\Gamma_i \nabla)\right]^n \phi}_{\text{diffusion}} + \underbrace{S_\phi}_{\text{source}} \tag{6.2}$$

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 Fickean 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 $n = 2$, or a phase field crystal term [*i.e.*, $\nabla \cdot (\Gamma_1 \nabla [\nabla \cdot \Gamma_2 \nabla \{\nabla \cdot \Gamma_3 \nabla \phi)\}])$) [Elder:2011p2811]] when $n = 3$, although spectral methods are probably a better approach. Higher order terms ($n > 3$) are also possible, but the matrix condition number becomes quite poor.

6.2 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.

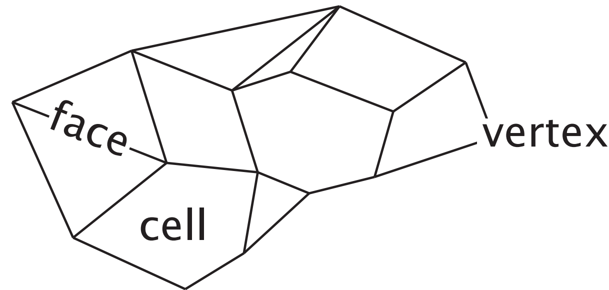


Figure 6.1: Mesh
A mesh consists of cells, faces and vertices. For the purposes of *FiPy*, the divider between two cells is known as a face for all dimensions.

6.2.1 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.

6.2.2 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.

6.3 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.

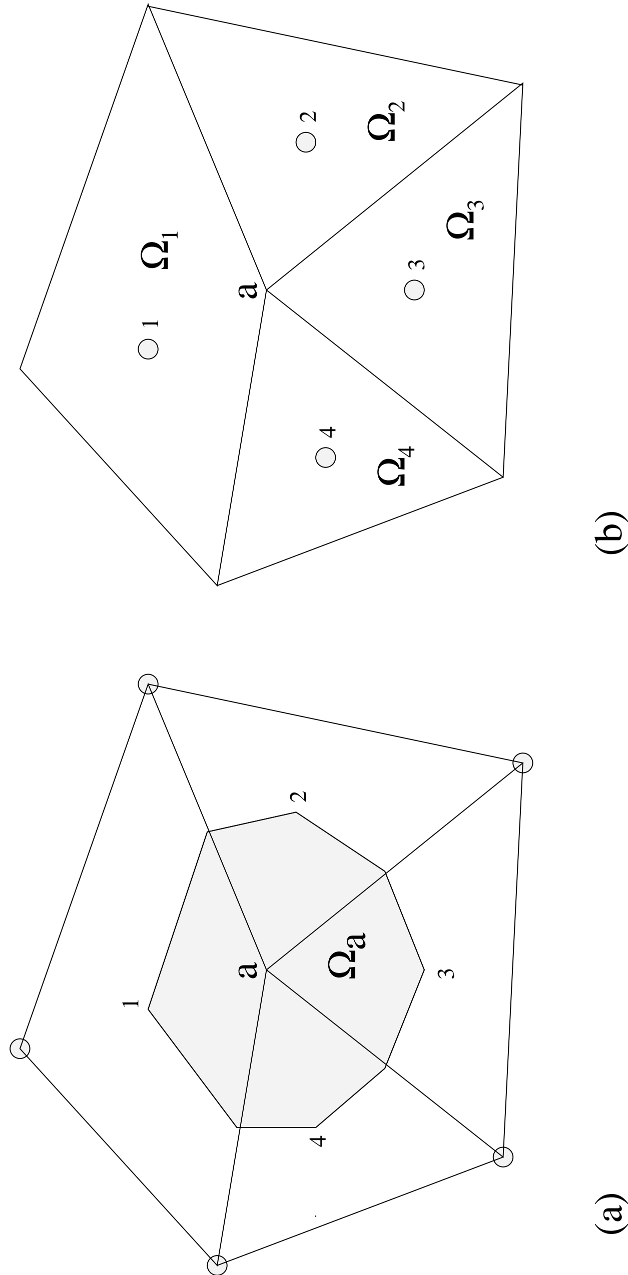


Figure 6.2: CV structure for an unstructured mesh
(a) Ω_a represents a vertex-based CV and (b) Ω_1, Ω_2, Ω_3 and Ω_4 represent cell centered CVs.

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

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

$$\int_V \frac{\partial(\rho\phi)}{\partial t} dV \simeq \frac{(\rho_P \phi_P - \rho_P^{\text{old}} \phi_P^{\text{old}}) V_P}{\Delta t} \tag{6.3}$$

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

This term is represented in *FiPy* as

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

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

The discretization for the convection term is given by

$$\begin{aligned}\int_V \nabla \cdot (\vec{u}\phi)\, dV &= \int_S (\vec{n} \cdot \vec{u})\phi\, dS \\ &\simeq \sum_f (\vec{n} \cdot \vec{u})_f \phi_f A_f\end{aligned} \tag{6.4}$$

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

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

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

This term is represented in *FiPy* as

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

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

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

The discretization for the diffusion term is given by

$$\begin{aligned}\int_V \nabla \cdot (\Gamma \nabla \{\ldots\}) dV &= \int_S \Gamma(\vec{n} \cdot \nabla \{\ldots\}) dS \\ &\simeq \sum_f \Gamma_f (\vec{n} \cdot \nabla \{\ldots\})_f A_f\end{aligned} \tag{6.5}$$

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

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

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].

This term is represented in *FiPy* as

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

or

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

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

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

Higher order diffusion

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

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

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

6.3.4 Source Term

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

$$\int_V S_\phi \, dV \simeq S_\phi V_P. \tag{6.6}$$

Including any negative dependence of S_ϕ on ϕ increases solution stability. The dependence can only be included in a linear manner so Equation (6.6) 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 coeficient of the source which is linearly dependent on ϕ.

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

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

6.4 Linear Equations

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

Combining Equations (6.3), (6.4), (6.5) and (6.6), the complete discretization for equation (**??**) can now be written for each CV as

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

Equation (6.4) 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

$$a_P \phi_P = \sum_f a_A \phi_A + b_P, \tag{6.7}$$

where

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

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

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

6.5 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

$$\frac{1}{1-\alpha_f} > P_f > -\frac{1}{\alpha_f} \tag{6.8}$$

must be satisfied. The parameter α_f is defined by the chosen scheme, depending on Equation (6.8). The various differencing schemes are:

the central differencing scheme, where

$$\alpha_f = \frac{1}{2} \tag{6.9}$$

so that $|P_f| < 2$ satisfies Equation (6.8). Thus, the central differencing scheme is only numerically stable for a low values of P_f.

the upwind scheme, where

$$\alpha_f = \begin{cases} 1 & \text{if } P_f > 0, \\ 0 & \text{if } P_f < 0. \end{cases} \tag{6.10}$$

Equation (6.10) satisfies the inequality in Equation (6.8) 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

$$\alpha_f = \frac{(P_f - 1)\exp(P_f) + 1}{P_f(\exp(P_f) - 1)}. \tag{6.11}$$

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

$$\alpha_f = \begin{cases} \frac{P_f - 1}{P_f} & \text{if } P_f > 2, \\ \frac{1}{2} & \text{if } |P_f| < 2, \\ -\frac{1}{P_f} & \text{if } P_f < -2. \end{cases} \tag{6.12}$$

The hybrid scheme is formulated by allowing $P_f \to \infty$, $P_f \to 0$ and $P_f \to -\infty$ 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

$$\alpha_f = \begin{cases} \frac{P_f - 1}{P_f} & \text{if } P_f > 10, \\ \frac{(P_f - 1) + (1 - P_f/10)^5}{P_f} & \text{if } 0 < P_f < 10, \\ \frac{(1 - P_f/10)^5 - 1}{P_f} & \text{if } -10 < P_f < 0, \\ -\frac{1}{P_f} & \text{if } P_f < -10. \end{cases} \tag{6.13}$$

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.

Chapter 7

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.

7.1 Design

7.1.1 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.

7.1.2 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.

7.1.3 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.

7.1.4 Open Source

In recent years, there has been a movement to release software under open source and associated unrestrictied 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.

7.1.5 High-Level Scripting Language

Programming languages can be broadly lumped into two categories: compiled languages and interpreted (or scripting) languages. Compiled languages are converted from a human-readable text source file to a machine-readable binary application file by a sequence of operations generally referred to as "compiling" and "linking." The binary application can then be run as many times as desired, but changes will provoke a new cycle of compiling and linking. Interpreted languages are converted from human-readable to machine-readable on the fly, each time the script is executed. Because the conversion happens every time [1], 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 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.

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

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.

7.1.6 *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 mathemeticians (*SciPy*, *NumPy*, *ScientificPython*, *PySparse*).
- easily integrated with low-level languages such as C (`weave`, `blitz`, *PyRex*).

7.2 Implementation

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

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

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

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

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

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

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

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

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

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

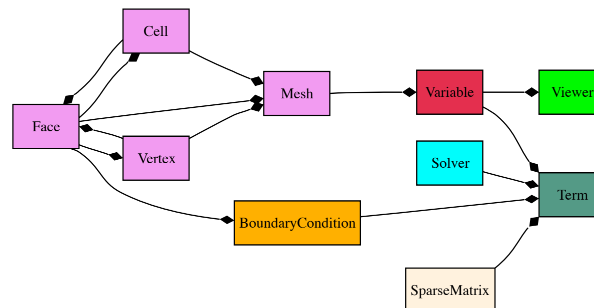


Figure 7.1: Primary object relationships in *FiPy*.

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

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

Chapter 8

Frequently Asked Questions

8.1 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

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

and the individual terms are discussed in *Discretization*.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

How do I represent a diffusive source?

If the governing equation for ϕ is

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

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

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

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

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

How do I represent a convective source?

The convection of an independent field ξ as in

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

can be rendered as

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

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

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

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

How do I represent a transient source?

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

8.2 How can I see what I'm doing?

8.2.1 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`.

8.2.2 How do I save a plot image?

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

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

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

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

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

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

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

8.2.3 How do I make a movie?

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

8.2.4 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 wan to change, while letting *FiPy* do most of the heavy lifting. See `examples.phase.anisotropy` and `examples.phase.polyxtal` for examples of creating a custom *Matplotlib* `Viewer` class; see `examples.cahnHilliard.sphere` for an example of creating a custom *Mayavi* `Viewer` class.

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

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

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

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

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

Solver iterations are changed from their defaults in `examples.flow.stokesCavity` and `examples.updating.update0_1to1_0`.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

or more elaborate:

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

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

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

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

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

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

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

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

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.

8.5 How do I represent boundary conditions?

See the *Boundary Conditions* section for more details.

8.6 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/mailinglists/mailman?fn=scipy-dev/2005-June/003010.html.

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

8.7 How do I change FiPy's default behavior?

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

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

See *Solving in Parallel*.

8.9 Why don't my scripts work anymore?

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

8.10 What if my question isn't answered here?

Please post your question to the mailing list <http://www.ctcms.nist.gov/fipy/mail.html> or file a Tracker request at <http://matforge.org/fipy/report>.

Chapter 9

Glossary

Buildbot The BuildBot is a system to automate the compile/test cycle required by most software projects to validate code changes. See http://www.nist.gov/cgi-bin/exit_nist.cgi?url=http://trac.buildbot.net/.

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

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

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

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 and *Matplotlib*.

Mayavi The `mayavi` Data Visualizer is a free, easy to use scientific data visualizer. It displays 1D, 2D and 3D data. It is the only *FiPy* viewer available for 3D data. Other viewers are probably better for 1D or 2D viewing. See http://www.nist.gov/cgi-bin/exit_nist.cgi?url=http://code.enthought.com/projects/mayavi and *Mayavi*.

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.

pip "pip installs python" is a tool for installing and managing Python packages, such as those found in *PyPI*. See http://www.nist.gov/cgi-bin/exit_nist.cgi?url=http://www.pip-installer.org.

PyAMG A suite of python-based preconditoners. See http://www.nist.gov/cgi-bin/exit_nist.cgi?url=http://code.google.com/p/pyamg/ and *PyAMG*.

PyPI The Python Package Index is a repository of software for the *Python* programming language. See http://www.nist.gov/cgi-bin/exit_nist.cgi?url=http://pypi.python.org/pypi.

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

PySparse The `pysparse` *Python* package provides sparse matrix storage, solvers, and linear algebra routines. See http://www.nist.gov/cgi-bin/exit_nist.cgi?url=http://pysparse.sourceforge.net and *PySparse*.

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/.

Python 3 The (likely) future of the *Python* programming language. Third-party packages are slowly being adapted, but many that *FiPy* uses are not yet available. See http://www.nist.gov/cgi-bin/exit_nist.cgi?url=http://docs.python.org/py3k/ and **PEP 3000**.

PyTrilinos *Python* wrapper for *Trilinos*. See http://www.nist.gov/cgi-bin/exit_nist.cgi?url=http://trilinos.sandia.gov/packages/pytrilinos

PyxViewer A now defunct python viewer.

ScientificPython A collection of useful utilities for scientists. See http://www.nist.gov/cgi-bin/exit_nist.cgi?url=http://dirac.cnrs-orleans.fr/plone/software/scientificpython.

SciPy The `scipy` package provides a wide range of scientific and mathematical operations. *FiPy* can use `scipy.weave` for enhanced performance with C language inlining and *Scipy*'s solver suite for linear solutions. See http://www.nist.gov/cgi-bin/exit_nist.cgi?url=http://www.scipy.org/. and *SciPy*.

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 provides sparse matrix storage, solvers, and preconditioners, and can be used instead of *PySparse*. *Trilinos* preconditioning allows for iterative solutions to some difficult problems that *PySparse* cannot solve. See http://www.nist.gov/cgi-bin/exit_nist.cgi?url=http://trilinos.sandia.gov and *Trilinos*.

Part II

Examples

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

These examples can be used in at least four ways:

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

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

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

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

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

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

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

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

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

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

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

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

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

Chapter 10

Diffusion Examples

examples.diffusion.mesh1D	Solve a one-dimensional diffusion equation under different conditi
examples.diffusion.coupled	Solve the biharmonic equation as a coupled pair of diffusion equat
examples.diffusion.mesh20x20	Solve a two-dimensional diffusion problem in a square domain.
examples.diffusion.circle	Solve the diffusion equation in a circular domain meshed with tria
examples.diffusion.electrostatics	Solve the Poisson equation in one dimension.
examples.diffusion.nthOrder.input4thOrder1D	Solve a fourth-order diffusion problem.
examples.diffusion.anisotropy	Solve the diffusion equation with an anisotropic diffusion coefficie

10.1 examples.diffusion.mesh1D

Solve a one-dimensional diffusion equation under different conditions.

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

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

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

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

$$\frac{\partial \phi}{\partial t} = D \nabla^2 \phi. \tag{10.1}$$

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.facesLeft`. The boundary conditions is applied using `phi. constrain()` with tthese faces and a value (`valueLeft`).

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

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

If you have ever tried to numerically solve Eq. (10.1), 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 $\Delta t \le \Delta x^2/(2D)$.

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

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

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

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

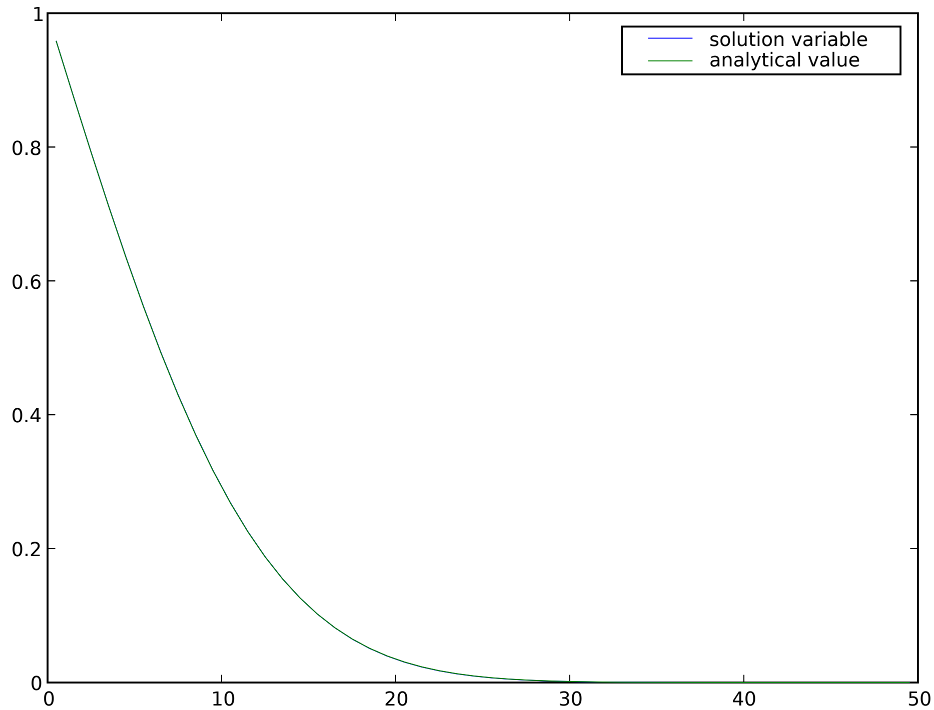
```
>>> if __name__ == '__main__':
...     viewer = Viewer(vars=(phi, phiAnalytical),
...                     datamin=0., datamax=1.)
...     viewer.plot()
```

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

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

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

We then solve the equation by repeatedly looping in time:

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

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

>>> 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. (10.1) as:

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

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

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

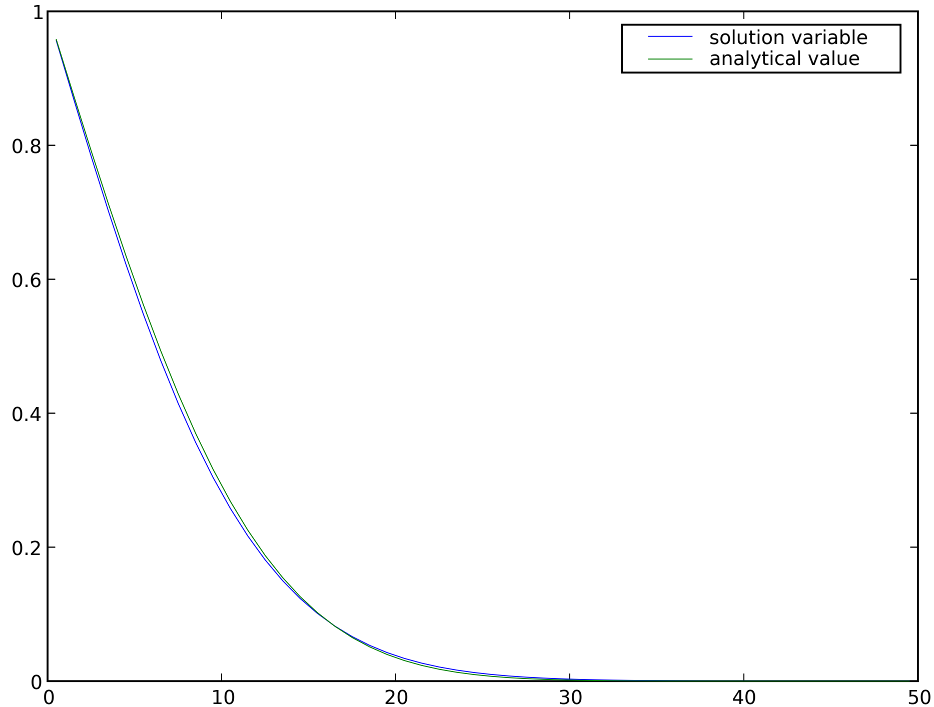
reset the problem

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

and rerun with much larger time steps

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

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

```
>>> 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,
...                dt=timeStepDuration)
...     if __name__ == '__main__':
...         viewer.plot()
>>> eqI.solve(var=phi,
```

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

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

>>> 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

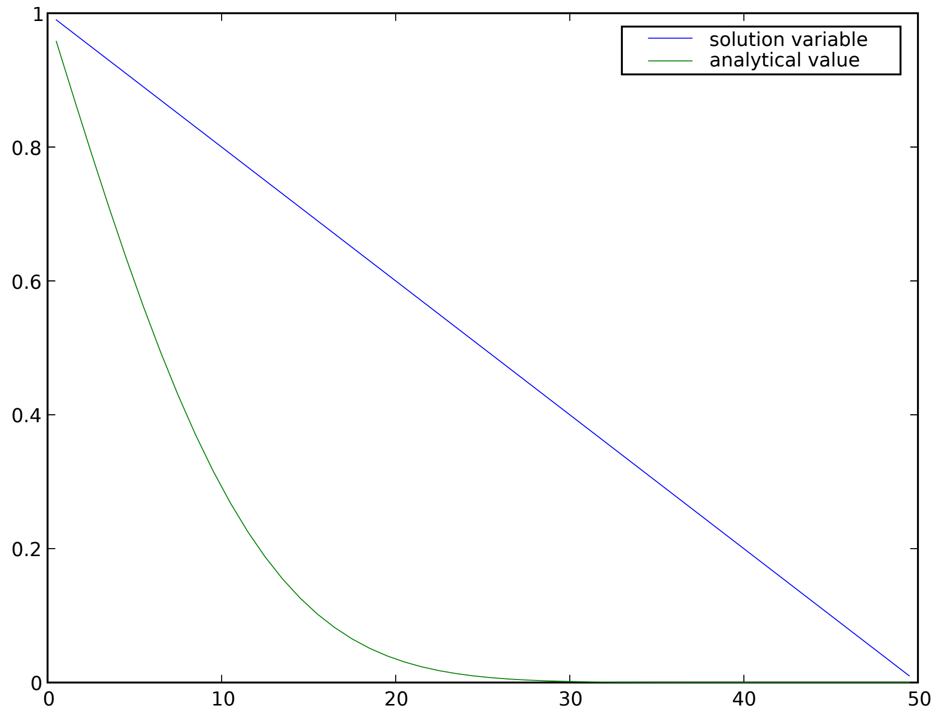
$$D\nabla^2\phi = 0$$

is represented in *FiPy* by

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

>>> 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()
```

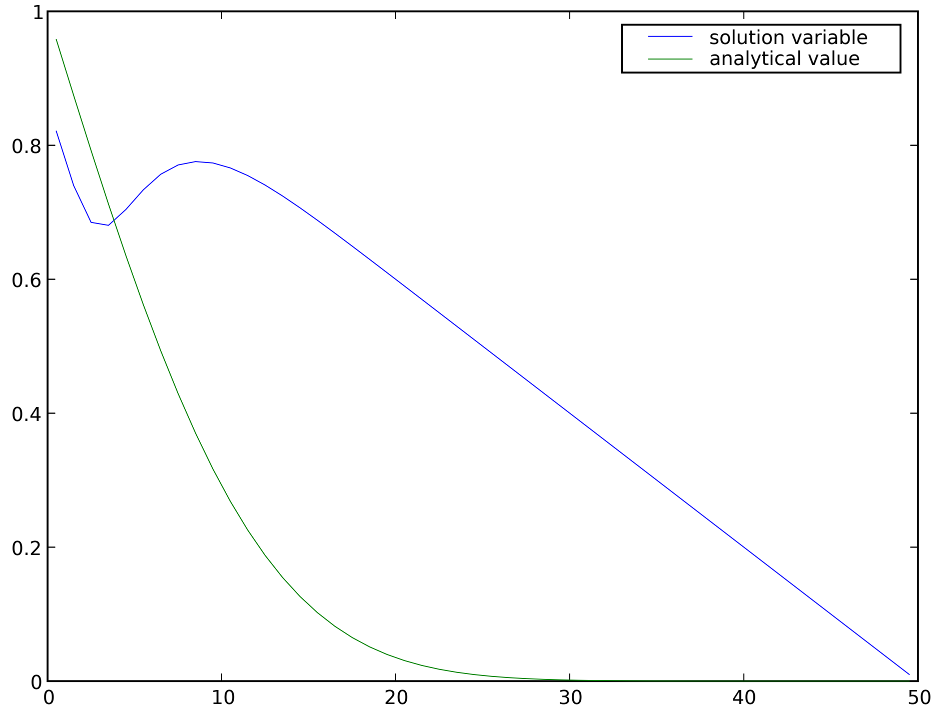
and then declare our boundary condition as a function of this `Variable`

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

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

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

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

```
>>> 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

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

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

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

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

The boundary conditions are a fixed value of

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

to the left and a fixed flux of

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

to the right:

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

We re-initialize the solution variable

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

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

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

The analytical solution is simply

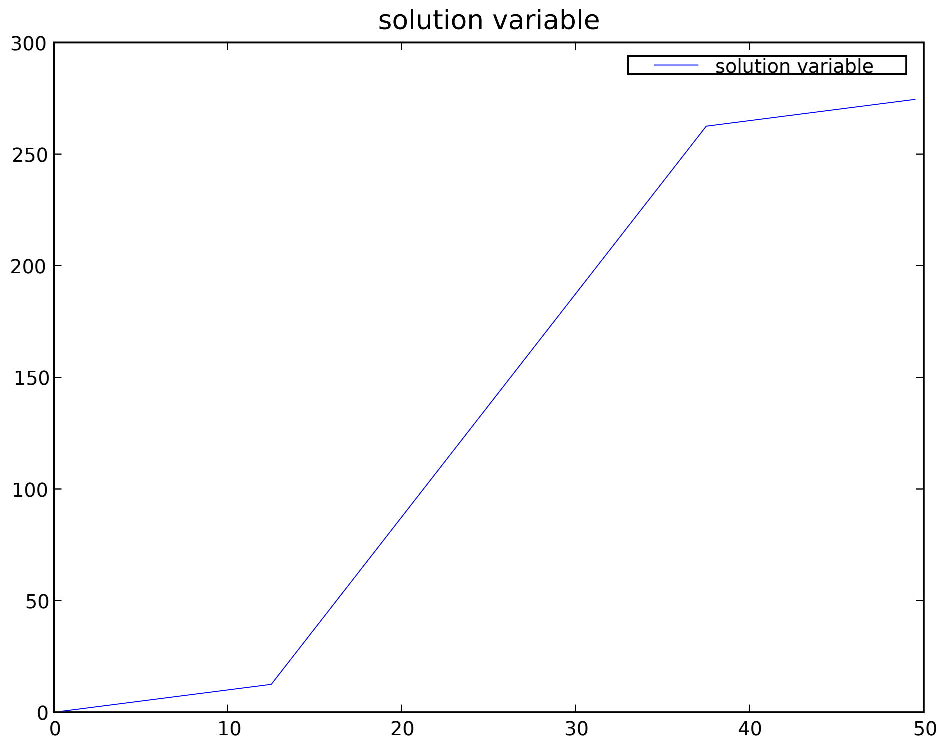
$$\phi = \begin{cases} x & \text{for } 0 < x < L/4, \\ 10x - 9L/4 & \text{for } L/4 \le x < 3L/4, \\ x + 18L/4 & \text{for } 3L/4 \le x < L, \end{cases}$$

or

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

And finally, we can plot the result

```
>>> 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

$$\frac{\partial \phi}{\partial t} = \nabla \cdot [D(\phi) \nabla \phi]. \tag{10.2}$$

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 explicity 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)
... ]
```

If, for example,

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

we would simply write Eq. (10.2) as

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

Note: Because of the non-linearity, the Crank-Nicholson scheme does not work for this problem.

We apply the same boundary conditions that we used for the uniform diffusivity cases

```
>>> phi[0].constrain(valueRight, mesh.facesRight)
>>> phi[0].constrain(valueLeft, mesh.facesLeft)
```

Although this problem does not have an exact transient solution, it can be solved in steady-state, with

$$\phi(x) = 1 - \sqrt{\frac{x}{L}}$$

```
>>> x = mesh.cellCenters[0]
>>> phiAnalytical.setValue(1. - numerix.sqrt(x/L))
```

We create a viewer to compare the different numbers of sweeps with the analytical solution from before.

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

As described above, an inner "sweep" loop is generally required for the solution of non-linear or multiple equation sets. Often a conditional is required to exit this "sweep" loop given some convergence criteria. Instead of using the `solve()` method equation, when sweeping, it is often useful to call `sweep()` instead. The `sweep()` method behaves the same way as `solve()`, but returns the residual that can then be used as part of the exit condition.

We now repeatedly run the problem with increasing numbers of sweeps.

```
>>> for sweeps in range(1,5):
...     phi[0].setValue(valueRight)
...     for step in range(steps):
...         # only move forward in time once per time step
...         phi[0].updateOld()
...
...         # but "sweep" many times per time step
...         for sweep in range(sweeps):
...             res = eq.sweep(var=phi[0],
...                            dt=timeStepDuration)
...         if __name__ == '__main__':
...             viewer.plot()
...
...     # copy the final result into the appropriate display variable
...     phi[sweeps].setValue(phi[0])
...     if __name__ == '__main__':
```

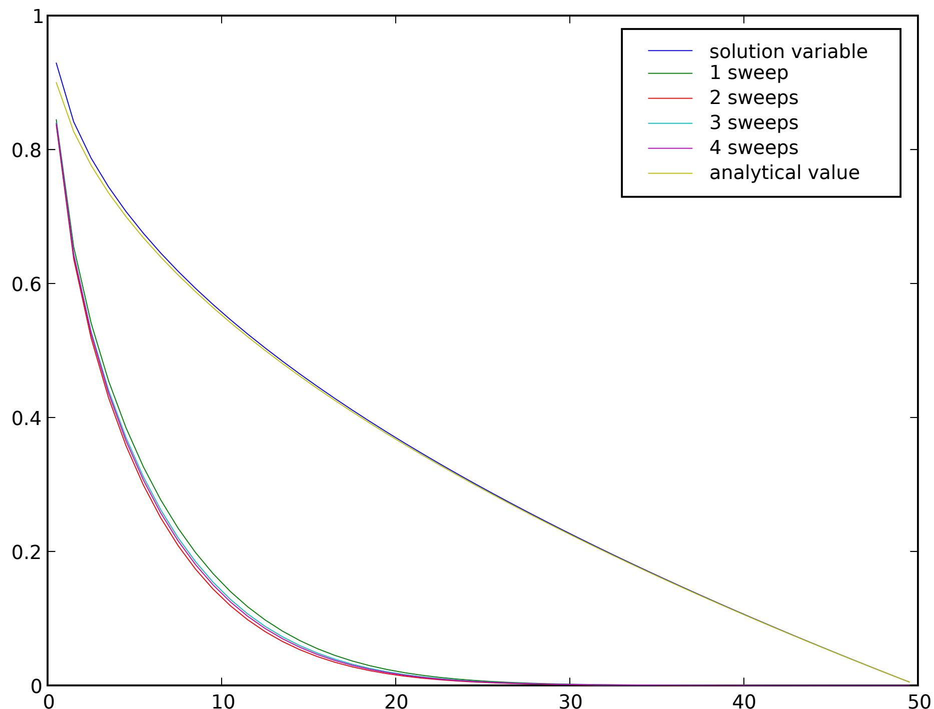
```
...         viewer.plot()
...         raw_input("Implicit variable diffusity. %d sweep(s). \
... Residual = %f. Press <return> to proceed..." % (sweeps, (abs(res))))
```

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],
...                    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],
                   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.

10.2 examples.diffusion.coupled

Solve the biharmonic equation as a coupled pair of diffusion equations.

FiPy has only first order time derivatives so equations such as the biharmonic wave equation written as

$$\frac{\partial^4 v}{\partial x^4} + \frac{\partial^2 v}{\partial t^2} = 0$$

cannot be represented as a single equation. We need to decompose the biharmonic equation into two equations that are first order in time in the following way,

$$\begin{aligned}\frac{\partial^2 v_0}{\partial x^2} + \frac{\partial v_1}{\partial t} &= 0 \\ \frac{\partial^2 v_1}{\partial x^2} - \frac{\partial v_0}{\partial t} &= 0\end{aligned}$$

Historically, *FiPy* required systems of coupled equations to be solved successively, "sweeping" the equations to convergence. As a practical example, we use the following system

$$\frac{\partial v_0}{\partial t} = 0.01 \nabla^2 v_0 - \nabla^2 v_1$$
$$\frac{\partial v_1}{\partial t} = \nabla^2 v_0 + 0.01 \nabla^2 v_1$$

subject to the boundary conditions

$$v_0|_{x=0} = 0 \qquad v_0|_{x=1} = 1$$
$$v_1|_{x=0} = 1 \qquad v_1|_{x=1} = 0$$

This system closely resembles the pure biharmonic equation, but has an additional diffusion contribution to improve numerical stability. The example system is solved with the following block of code using explicit coupling for the cross-coupled terms.

```
>>> from fipy import Grid1D, CellVariable, TransientTerm, DiffusionTerm, Viewer

>>> m = Grid1D(nx=100, Lx=1.)

>>> v0 = CellVariable(mesh=m, hasOld=True, value=0.5)
>>> v1 = CellVariable(mesh=m, hasOld=True, value=0.5)

>>> v0.constrain(0, m.facesLeft)
>>> v0.constrain(1, m.facesRight)

>>> v1.constrain(1, m.facesLeft)
>>> v1.constrain(0, m.facesRight)

>>> eq0 = TransientTerm() == DiffusionTerm(coeff=0.01) - v1.faceGrad.divergence
>>> eq1 = TransientTerm() == v0.faceGrad.divergence + DiffusionTerm(coeff=0.01)

>>> vi = Viewer((v0, v1))

>>> for t in range(100):
...     v0.updateOld()
...     v1.updateOld()
...     res0 = res1 = 1e100
...     while max(res0, res1) > 0.1:
...         res0 = eq0.sweep(var=v0, dt=1e-5)
...         res1 = eq1.sweep(var=v1, dt=1e-5)
...     if t % 10 == 0:
...         vi.plot()
```

The uncoupled method still works, but it can be advantageous to solve the two equations simultaneously. In this case, by coupling the equations, we can eliminate the explicit sources and dramatically increase the time steps:

```
>>> v0.value = 0.5
>>> v1.value = 0.5

>>> eqn0 = TransientTerm(var=v0) == DiffusionTerm(0.01, var=v0) - DiffusionTerm(1, var=v1)
>>> eqn1 = TransientTerm(var=v1) == DiffusionTerm(1, var=v0) + DiffusionTerm(0.01, var=v1)

>>> eqn = eqn0 & eqn1

>>> for t in range(1):
...     v0.updateOld()
```

```
...     v1.updateOld()
...     eqn.solve(dt=1.e-3)
...     vi.plot()
```

It is also possible to pose the same equations in vector form:

```
>>> v = CellVariable(mesh=m, hasOld=True, value=[[0.5], [0.5]], elementshape=(2,))

>>> v.constrain([[0], [1]], m.facesLeft)
>>> v.constrain([[1], [0]], m.facesRight)

>>> eqn = TransientTerm([[1, 0],
...                      [0, 1]]) == DiffusionTerm([[[0.01, -1],
...                                                  [1, 0.01]]])

>>> vi = Viewer((v[0], v[1]))

>>> for t in range(1):
...     v.updateOld()
...     eqn.solve(var=v, dt=1.e-3)
...     vi.plot()
```

Whether you pose your problem in coupled or vector form should be dictated by the underlying physics. If v_0 and v_1 represent the concentrations of two conserved species, then it is natural to write two seperate governing equations and to couple them. If they represent two components of a vector field, then the vector formulation is obviously more natural. FiPy will solve the same matrix system either way.

10.3 examples.diffusion.mesh20x20

Solve a two-dimensional diffusion problem in a square domain.

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

```
>>> from fipy import *

>>> nx = 20
>>> ny = nx
>>> dx = 1.
>>> dy = dx
>>> L = dx * nx
>>> mesh = Grid2D(dx=dx, dy=dy, nx=nx, ny=ny)
```

We create a `CellVariable` and initialize it to zero:

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

and then create a diffusion equation. This is solved by default with an iterative conjugate gradient solver.

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

We apply Dirichlet boundary conditions

```
>>> valueTopLeft = 0
>>> valueBottomRight = 1
```

to the top-left and bottom-right corners. Neumann boundary conditions are automatically applied to the top-right and bottom-left corners.

```
>>> X, Y = mesh.faceCenters
>>> facesTopLeft = ((mesh.facesLeft & (Y > L / 2))
...                 | (mesh.facesTop & (X < L / 2)))
>>> facesBottomRight = ((mesh.facesRight & (Y < L / 2))
...                     | (mesh.facesBottom & (X > L / 2)))
```

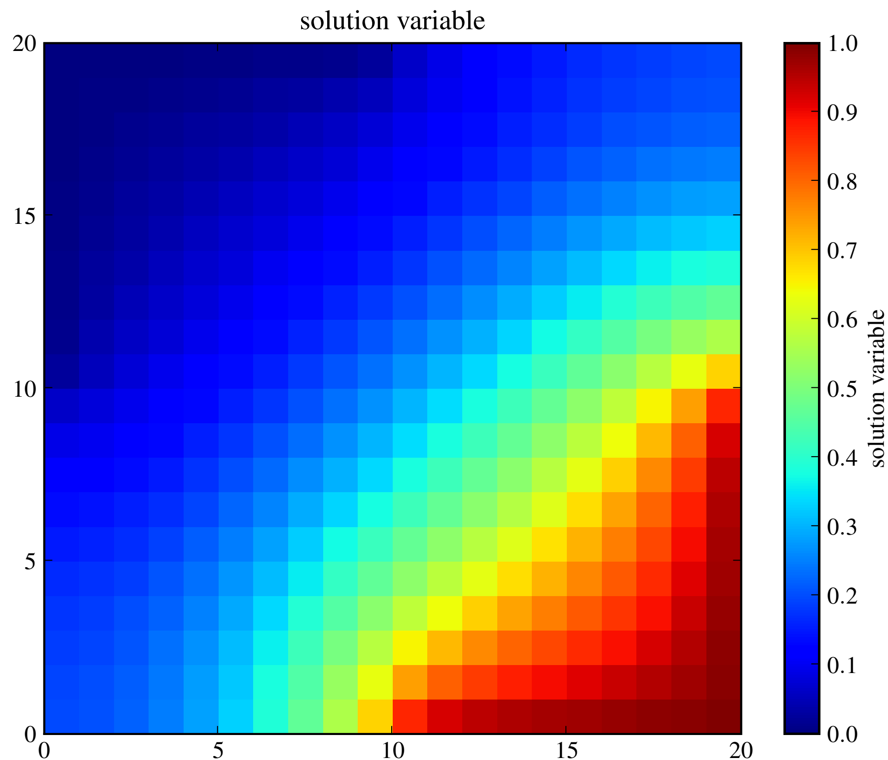
```
>>> phi.constrain(valueTopLeft, facesTopLeft)
>>> phi.constrain(valueBottomRight, facesBottomRight)
```

We create a viewer to see the results

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

and solve the equation by repeatedly looping in time:

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

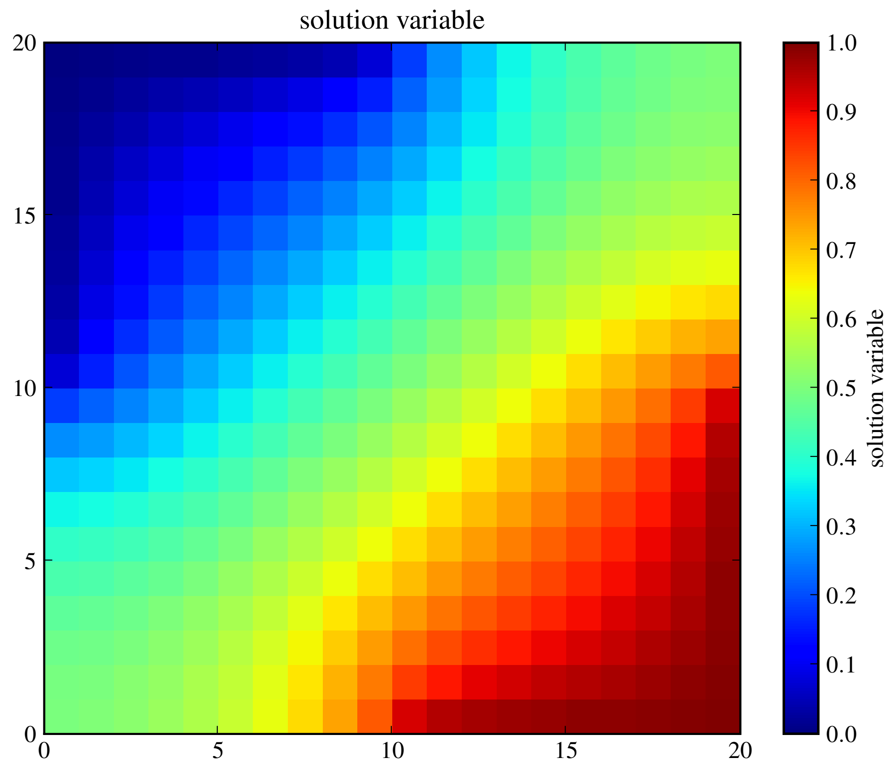


We can test the value of the bottom-right corner cell.

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

>>> 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)
>>> 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...")
```

10.4 examples.diffusion.circle

Solve the diffusion equation in a circular domain meshed with triangles.

This example demonstrates how to solve a simple diffusion problem on a non-standard mesh with varying boundary conditions. The *Gmsh* package is used to create the mesh. Firstly, define some parameters for the creation of the mesh,

```
>>> cellSize = 0.05
>>> radius = 1.
```

The *cellSize* is the preferred edge length of each mesh element and the *radius* is the radius of the circular mesh domain. In the following code section a file is created with the geometry that describes the mesh. For details of how to write such geometry files for *Gmsh*, see the gmsh manual.

The mesh created by *Gmsh* is then imported into *FiPy* using the `Gmsh2D` object.

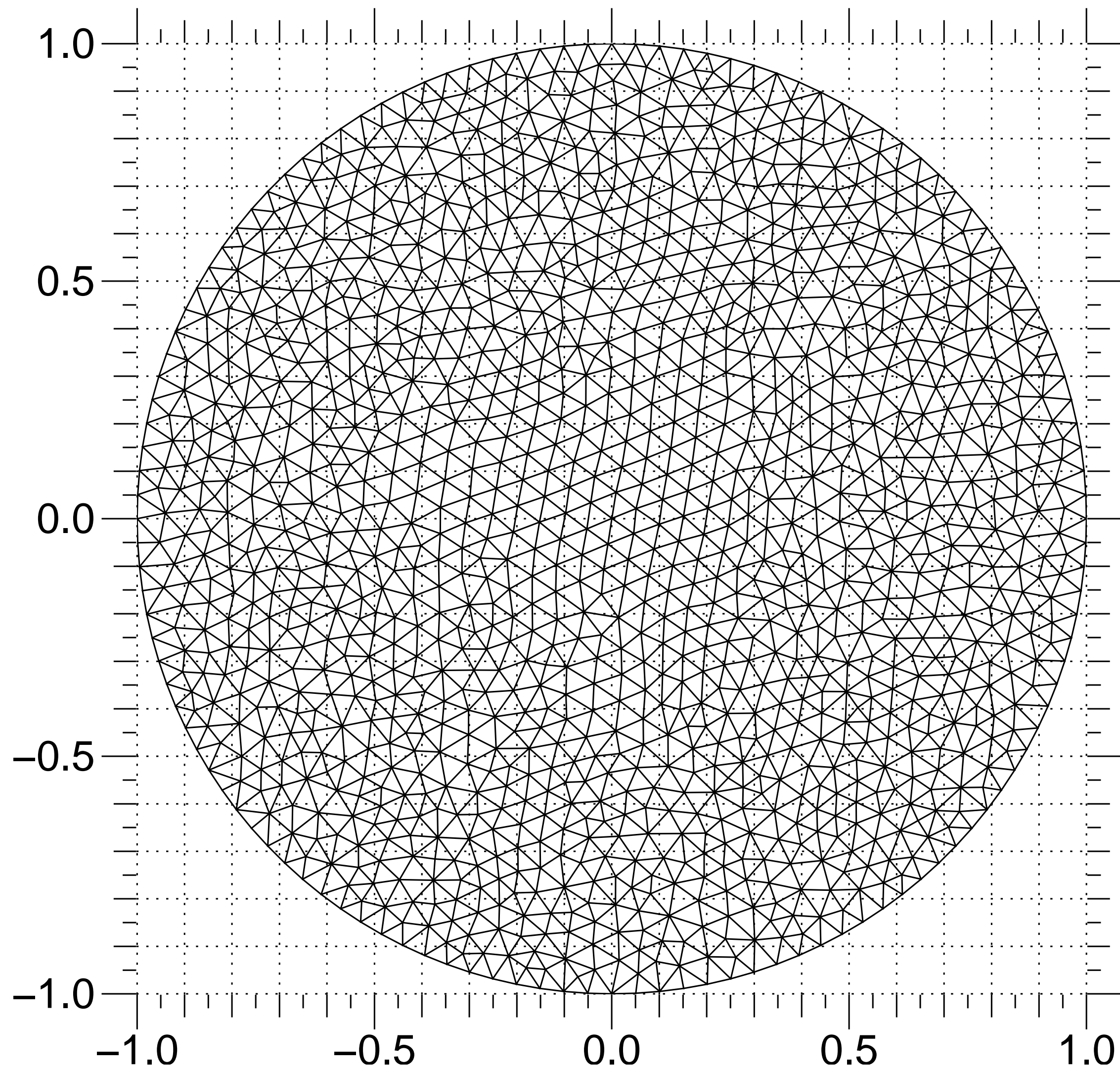
```
>>> from fipy import *
>>> mesh = Gmsh2D('''
...                 cellSize = %(cellSize)g;
...                 radius = %(radius)g;
...                 Point(1) = {0, 0, 0, cellSize};
...                 Point(2) = {-radius, 0, 0, cellSize};
...                 Point(3) = {0, radius, 0, cellSize};
...                 Point(4) = {radius, 0, 0, cellSize};
...                 Point(5) = {0, -radius, 0, cellSize};
...                 Circle(6) = {2, 1, 3};
...                 Circle(7) = {3, 1, 4};
...                 Circle(8) = {4, 1, 5};
...                 Circle(9) = {5, 1, 2};
...                 Line Loop(10) = {6, 7, 8, 9};
...                 Plane Surface(11) = {10};
...                 ''' % locals())
```

Using this mesh, we can construct a solution variable

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

We can now create a `Viewer` to see the mesh

```
>>> viewer = None
>>> 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, Matplotlib2DVie
```



We set up a transient diffusion equation

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

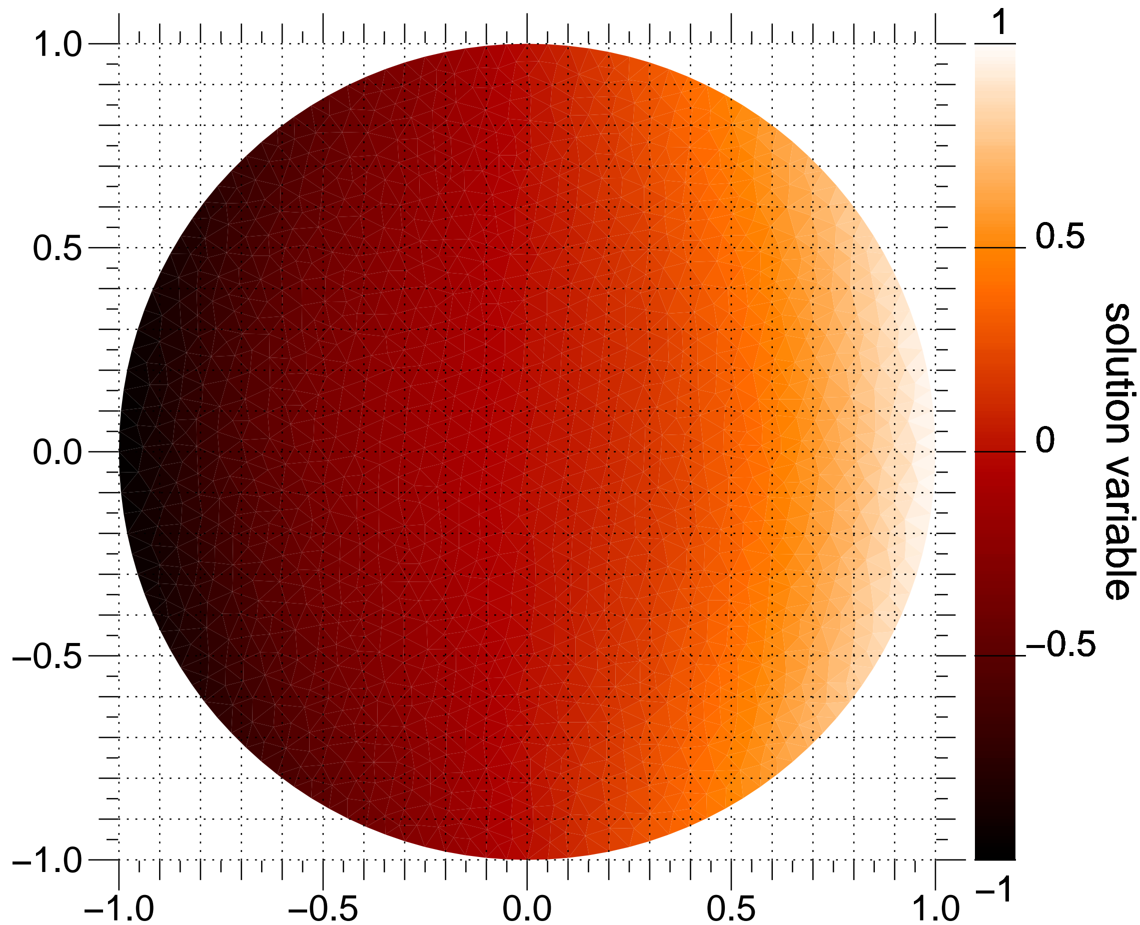
The following line extracts the x coordinate values on the exterior faces. These are used as the boundary condition fixed values.

```
>>> X, Y = mesh.faceCenters
```

```
>>> phi.constrain(X, mesh.exteriorFaces)
```

We first step through the transient problem

```
>>> timeStepDuration = 10 * 0.9 * cellSize**2 / (2 * D)
>>> steps = 10
>>> for step in range(steps):
...     eq.solve(var=phi,
...              dt=timeStepDuration)
...     if viewer is not None:
...         viewer.plot()
```



If we wanted to plot or analyze the results of this calculation with another application, we could export tab-separated-values with

```
TSVViewer(vars=(phi, phi.grad)).plot(filename="myTSV.tsv")
```

```
x         y         solution variable        solution variable_grad_x        solution variable_grad_y
0.975559734792414        0.0755414402612554        0.964844363287199        -0.229687917881182
0.0442864953037566        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.cellCenters
>>> t = timeStepDuration * steps

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

>>> x0 = radius * numerix.cos(numerix.arcsin(y))
>>> try:
...     from scipy.special import erf
...     ## This function can sometimes throw nans on OS X
...     ## see http://projects.scipy.org/scipy/scipy/ticket/325
...     phiAnalytical.setValue(x0 * (erf((x0+x) / (2 * numerix.sqrt(D * t)))
...                                  - erf((x0-x) / (2 * numerix.sqrt(D * t)))))
... except ImportError:
...     print "The SciPy library is not available to test the solution to \
... the transient diffusion equation"

>>> print phi.allclose(phiAnalytical, atol = 7e-2)
1

>>> 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.

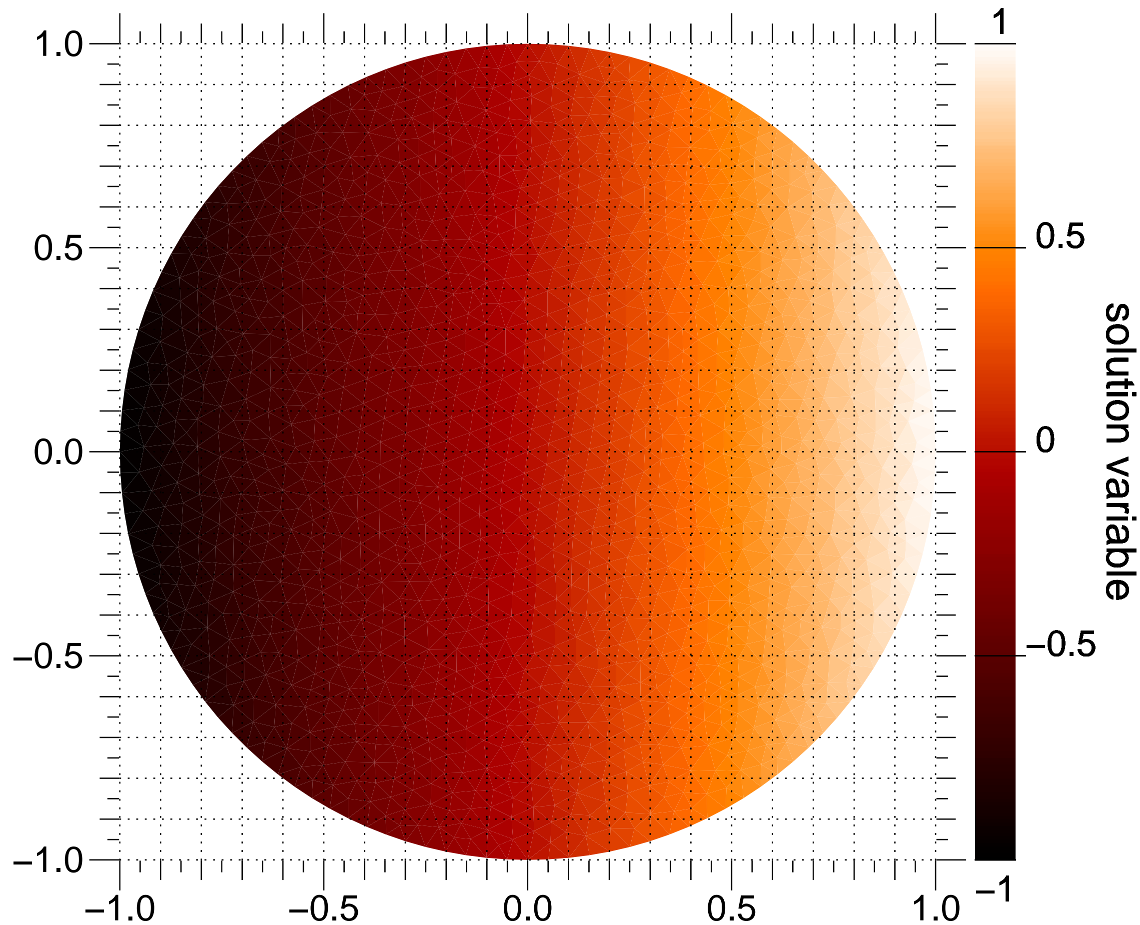
```
>>> DiffusionTerm(coeff=D).solve(var=phi)
```

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

```
>>> print phi.allclose(x, atol = 0.03)
1
```

Display the results if run as a script.

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



10.5 examples.diffusion.electrostatics

Solve the Poisson equation in one dimension.

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

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

```
>>> 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 jth component with valence z_j (we consider only a single component C_{e}^{-} with valence with $z_{\text{e}^-} = -1$)

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

and the charge density ρ,

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

The dimensionless Poisson equation is

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

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

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

```
>>> potential.constrain(0., mesh.facesLeft)
```

First, we obtain a uniform charge distribution by setting a uniform concentration of electrons $C_{\mathrm{e}^-} = 1$.

```
>>> electrons.setValue(1.)
```

and we solve for the electrostatic potential

```
>>> potential.equation.solve(var=potential)
```

This problem has the analytical solution

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

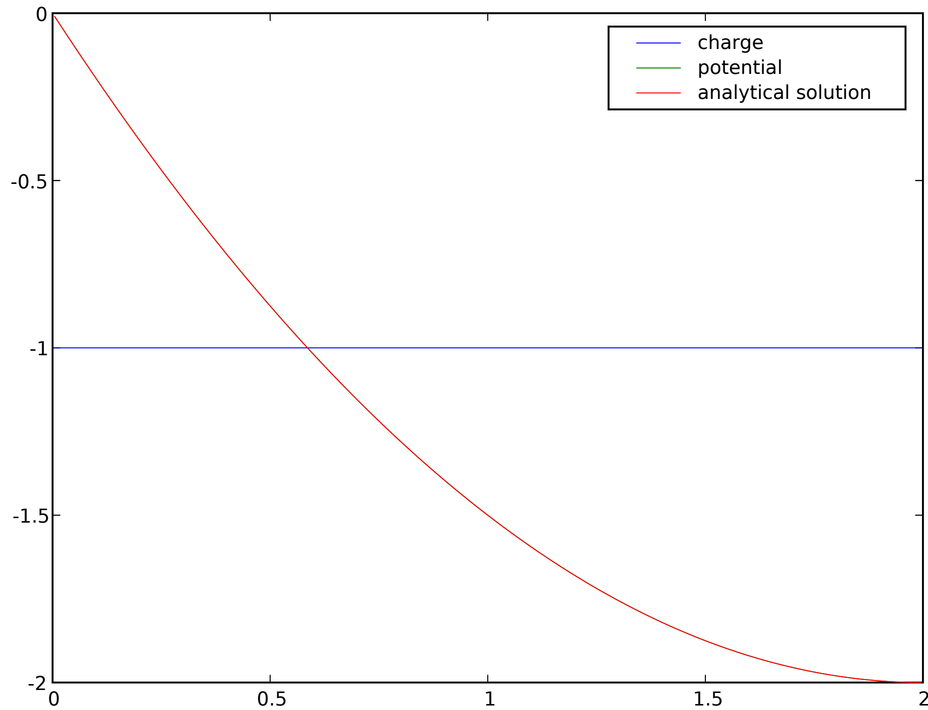
```
>>> x = mesh.cellCenters[0]
>>> analytical = CellVariable(mesh=mesh, name="analytical solution",
...                           value=(x**2)/2 - 2*x)
```

which has been satisifactorily 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

$$C_{\text{e}^-} = \begin{cases} 0 & \text{for } x \le L/2, \\ 1 & \text{for } x > L/2. \end{cases}$$

```
>>> x = mesh.cellCenters[0]
>>> electrons.setValue(0.)
>>> electrons.setValue(1., where=x > L / 2.)
```

and again solve for the electrostatic potential

```
>>> potential.equation.solve(var=potential)
```

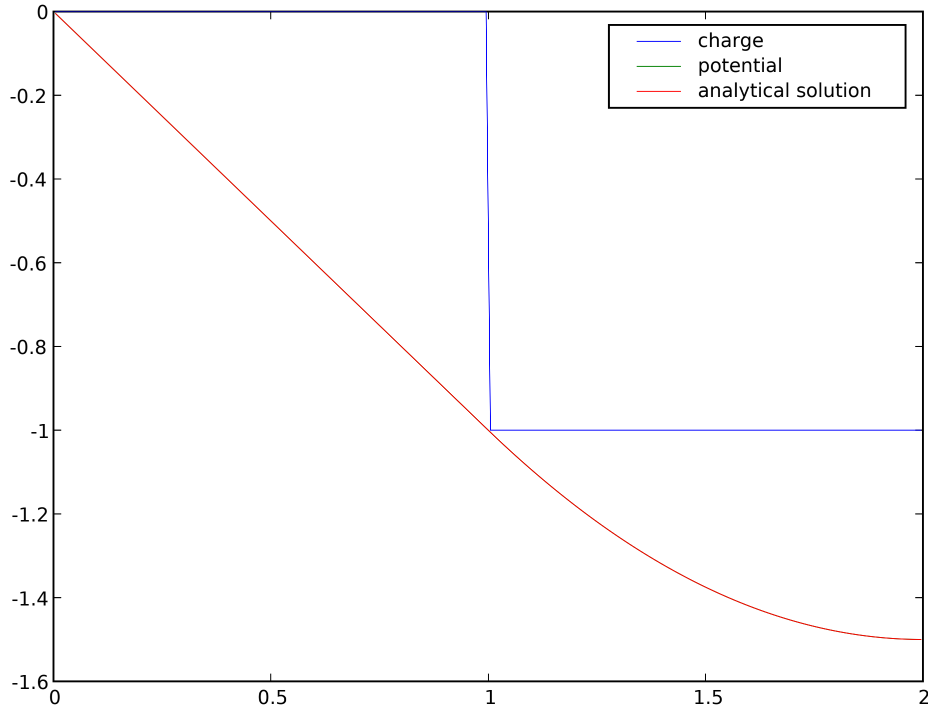
which now has the analytical solution

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

```
>>> 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

$$C_{\mathrm{e}^-} = \begin{cases} 1 & \text{for } x \le L/2, \\ 0 & \text{for } x > L/2. \end{cases}$$

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

and again solve for the electrostatic potential

```
>>> potential.equation.solve(var=potential)
```

which has the analytical solution

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

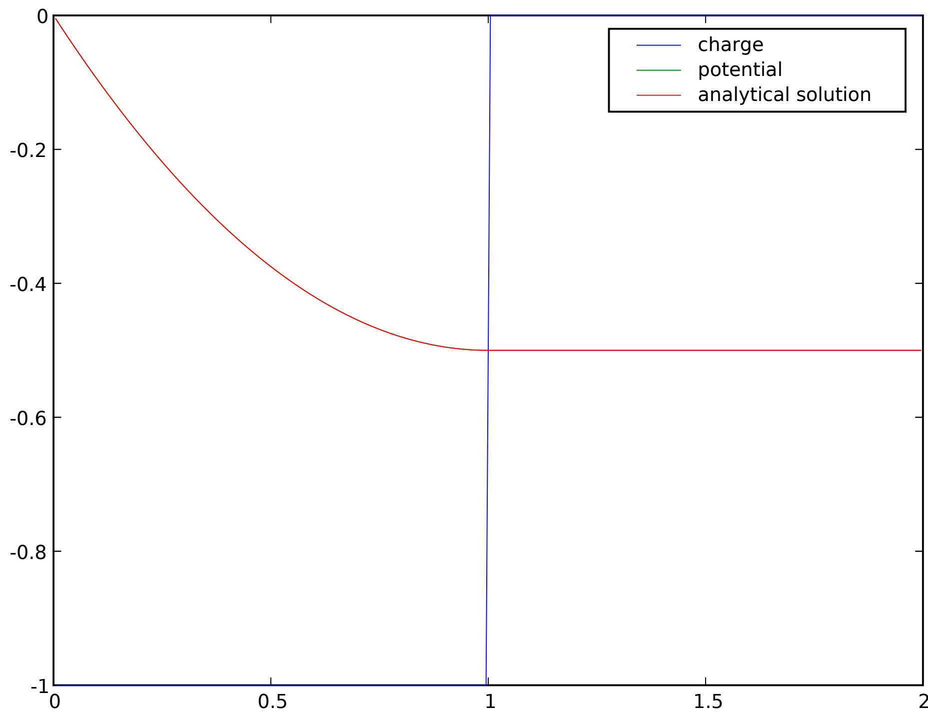
We again verify that the correct equilibrium is attained

```
>>> 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()
```



10.6 examples.diffusion.nthOrder.input4thOrder1D

Solve a fourth-order diffusion problem.

This example uses the DiffusionTerm class to solve the equation

$$\frac{\partial^4 \phi}{\partial x^4} = 0$$

on a 1D mesh of length

```
>>> L = 1000.
```

We create an appropriate mesh

```
>>> from fipy import *

>>> nx = 500
>>> dx = L / nx
>>> mesh = Grid1D(dx=dx, nx=nx)
```

and initialize the solution variable to 0

```
>>> var = CellVariable(mesh=mesh, name='solution variable')
```

For this problem, we impose the boundary conditions:

$$\begin{aligned}
\phi &= \alpha_1 \quad \text{at } x = 0 \\
\frac{\partial \phi}{\partial x} &= \alpha_2 \quad \text{at } x = L \\
\frac{\partial^2 \phi}{\partial x^2} &= \alpha_3 \quad \text{at } x = 0 \\
\frac{\partial^3 \phi}{\partial x^3} &= \alpha_4 \quad \text{at } x = L.
\end{aligned}$$

or

```
>>> alpha1 = 2.
>>> alpha2 = 1.
>>> alpha3 = 4.
>>> alpha4 = -3.
```

```
>>> BCs = (NthOrderBoundaryCondition(faces=mesh.facesLeft, value=alpha3, order=2),
...        NthOrderBoundaryCondition(faces=mesh.facesRight, value=alpha4, order=3))
>>> var.faceGrad.constrain([alpha2], mesh.facesRight)
>>> var.constrain(alpha1, mesh.facesLeft)
```

We initialize the steady-state equation

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

and use the `LinearLUSolver` for stability.

We perform one implicit timestep to achieve steady state

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

The analytical solution is:

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

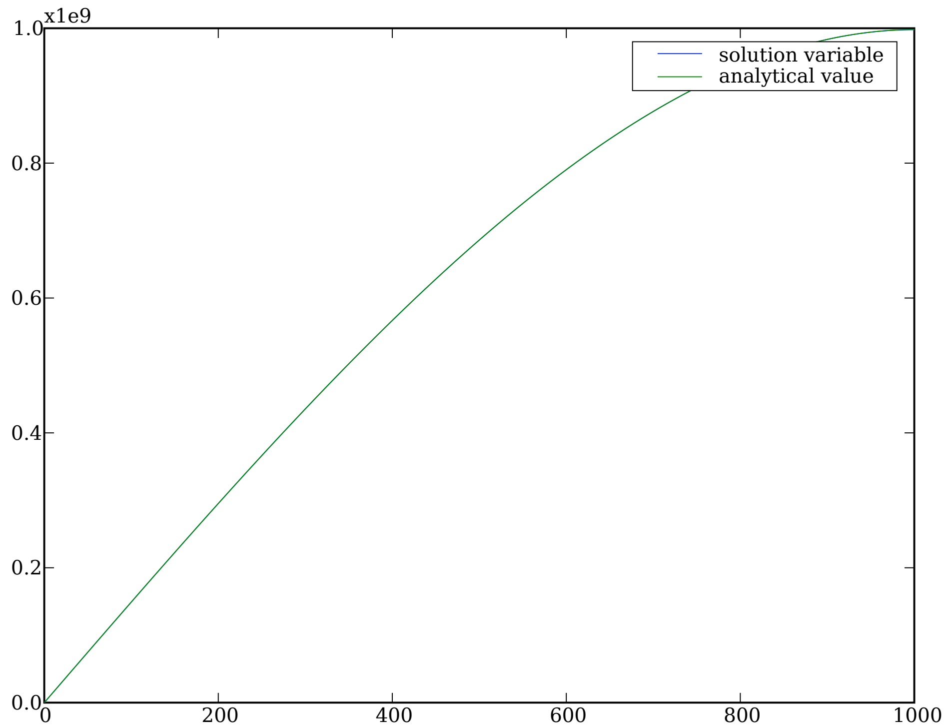
or

```
>>> analytical = CellVariable(mesh=mesh, name='analytical value')
>>> x = mesh.cellCenters[0]
>>> analytical.setValue(alpha4 / 6. * x**3 + alpha3 / 2. * x**2 + \
...                     (alpha2 - alpha4 / 2. * L**2 - alpha3 * L) * x + alpha1)
```

```
>>> print var.allclose(analytical, rtol=1e-4)
1
```

If the problem is run interactively, we can view the result:

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



10.7 examples.diffusion.anisotropy

Solve the diffusion equation with an anisotropic diffusion coefficient.

We wish to solve the problem

$$\frac{\partial \phi}{\partial t} = \partial_j \Gamma_{ij} \partial_i \phi$$

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

$$\Gamma' = \begin{pmatrix} 0.2 & 0 \\ 0 & 1 \end{pmatrix}$$

A new matrix is formed by rotating Γ' such that

$$R = \begin{pmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{pmatrix}$$

and

$$\Gamma = R \Gamma' R^T$$

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

$$\phi\left(X, Y, t\right) = Q \frac{\exp\left(-\frac{1}{4t}\left(\frac{X^2}{\Gamma'_{00}} + \frac{Y^2}{\Gamma'_{11}}\right)\right)}{4 \pi t \sqrt{\Gamma'_{00} \Gamma'_{11}}}$$

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

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

Import a mesh previously created using *Gmsh*.

```
>>> import os
>>> mesh = Gmsh2D(os.path.splitext(__file__)[0] + '.msh', communicator=serial)
```

Set the centermost cell to have a value.

```
>>> var = CellVariable(mesh=mesh, hasOld=1)
>>> x, y = mesh.cellCenters
>>> var[numerix.argmin(x**2 + y**2)] = 1.
```

Choose an orientation for the anisotropy.

```
>>> theta = numerix.pi / 4.
>>> rotationMatrix = numerix.array(((numerix.cos(theta), numerix.sin(theta)), \
...                                 (-numerix.sin(theta), numerix.cos(theta))))
>>> gamma_prime = numerix.array(((0.2, 0.), (0., 1.)))
>>> DOT = numerix.NUMERIX.dot
>>> gamma = DOT(DOT(rotationMatrix, gamma_prime), numerix.transpose(rotationMatrix))
```

Make the equation, viewer and solve.

```
>>> eqn = TransientTerm() == DiffusionTermCorrection((gamma,))

>>> if __name__ == '__main__':
...     viewer = Viewer(var, datamin=0.0, datamax=0.001)

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

>>> for i in range(20):
...     var.updateOld()
...     res = 1.
...
...     while res > 1e-2:
...         res = eqn.sweep(var, dt=dt)
...
...     if __name__ == '__main__':
...         viewer.plot()
...     time += dt
```

Compare with the analytical solution (within 5% accuracy).

```
>>> X, Y = numerix.dot(mesh.cellCenters, CellVariable(mesh=mesh, rank=2, value=rotationMatrix))
>>> solution = mass * numerix.exp(-(X**2 / gamma_prime[0][0] + Y**2 / gamma_prime[1][1]) / (4 * time)
>>> print max(abs((var - solution) / max(solution))) < 0.08
True
```

Chapter 11

Convection Examples

examples.convection.exponential1D.mesh1D	Solve the steady-state convection-diffusion equation in one di
examples.convection.exponential1DSource.mesh1D	Solve the steady-state convection-diffusion equation with a co
examples.convection.robin	Solve an advection-diffusion equation with a Robin boundary
examples.convection.source	Solve a convection problem with a source.

11.1 examples.convection.exponential1D.mesh1D

Solve the steady-state convection-diffusion equation in one dimension.

This example solves the steady-state convection-diffusion equation given by

$$\nabla \cdot (D \nabla \phi + \vec{u} \phi) = 0$$

with coefficients $D = 1$ and $\vec{u} = 10\hat{ß}$, or

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

We define a 1D mesh

```
>>> from fipy import *

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

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

The solution variable is initialized to `valueLeft`:

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

and impose the boundary conditions

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

with

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

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

and test the solution against the analytical result

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

or

```
>>> axis = 0
>>> x = mesh.cellCenters[axis]
>>> CC = 1. - numerix.exp(-convCoeff[axis] * x / diffCoeff)
>>> DD = 1. - numerix.exp(-convCoeff[axis] * L / diffCoeff)
>>> analyticalArray = CC / DD
>>> print var.allclose(analyticalArray)
1
```

If the problem is run interactively, we can view the result:

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

11.2 examples.convection.exponential1DSource.mesh1D

Solve the steady-state convection-diffusion equation with a constant source.

Like `examples.convection.exponential1D.mesh1D` this example solves a steady-state convection-diffusion equation, but adds a constant source, $S_0 = 1$, such that

$$\nabla \cdot (D \nabla \phi + \vec{u} \phi) + S_0 = 0.$$

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

We define a 1D mesh

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

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

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

The solution variable is initialized to `valueLeft`:

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

and impose the boundary conditions

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

with

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

We define the convection-diffusion equation with source

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

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

and test the solution against the analytical result:

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

or

```
>>> axis = 0
>>> x = mesh.cellCenters[axis]
>>> AA = -sourceCoeff * x / convCoeff[axis]
>>> BB = 1. + sourceCoeff * L / convCoeff[axis]
>>> CC = 1. - numerix.exp(-convCoeff[axis] * x / diffCoeff)
>>> DD = 1. - numerix.exp(-convCoeff[axis] * L / diffCoeff)
>>> analyticalArray = AA + BB * CC / DD
>>> print var.allclose(analyticalArray, rtol=1e-4, atol=1e-4)
1
```

If the problem is run interactively, we can view the result:

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

11.3 examples.convection.robin

Solve an advection-diffusion equation with a Robin boundary condition.

This example demonstrates how to apply a Robin boundary condition to an advection-diffusion equation. The equation

we wish to solve is given by,

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

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

>>> C.faceGrad.constrain([-P + P * C.faceValue], mesh.facesLeft)
>>> C.faceGrad.constrain([0], mesh.facesRight)

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

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

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

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

>>> if __name__ == '__main__':
...     restol = 1e-5
...     anstol = 1e-3
... else:
...     restol = 0.5
...     anstol = 0.15

>>> res = 1e+10
```

```
>>> while res > restol:
...     res = eq.sweep(var=C)
...     if __name__ == '__main__':
...         viewer.plot()

>>> print C.allclose(CAnalytical, rtol=anstol, atol=anstol)
True
```

11.4 examples.convection.source

Solve a convection problem with a source.

This example solves the equation

$$\frac{\partial \phi}{\partial x} - \alpha \phi = 0$$

with $\phi(0) = 1$ at $x = 0$. The boundary condition at $x = L$ 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 * numerix.exp(-alpha * mesh.cell

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

>>> phi.constrain(phi0, mesh.facesLeft)
>>> ## fake outflow condition
>>> phi.faceGrad.constrain([0], mesh.facesRight)

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

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

Chapter 12

Phase Field Examples

examples.phase.simple	Solve a phase-field (Allen-Cahn) problem in one-dimension.
examples.phase.binaryCoupled	Simultaneously solve a phase-field evolution and solute diffusion problem in
examples.phase.quaternary	Solve a phase-field evolution and diffusion of four species in one-dimension.
examples.phase.anisotropy	Solve a dendritic solidification problem.
examples.phase.impingement.mesh40x1	Solve for the impingement of two grains in one dimension.
examples.phase.impingement.mesh20x20	Solve for the impingement of four grains in two dimensions.
examples.phase.polyxtal	Solve the dendritic growth of nuclei and subsequent grain impingement.
examples.phase.polyxtalCoupled	Simultaneously solve the dendritic growth of nuclei and subsequent grain im

12.1 examples.phase.simple

Solve a phase-field (Allen-Cahn) problem in one-dimension.

To run this example from the base FiPy directory, type `python examples/phase/simple/input.py` at the command line. A viewer object should appear and, after being prompted to step through the different examples, the word `finished` in the terminal.

This example takes the user through assembling a simple problem with FiPy. It describes a steady 1D phase field problem with no-flux boundary conditions such that,

$$\frac{1}{M_\phi}\frac{\partial \phi}{\partial t} = \kappa_\phi \nabla^2 \phi - \frac{\partial f}{\partial \phi} \tag{12.1}$$

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)
```

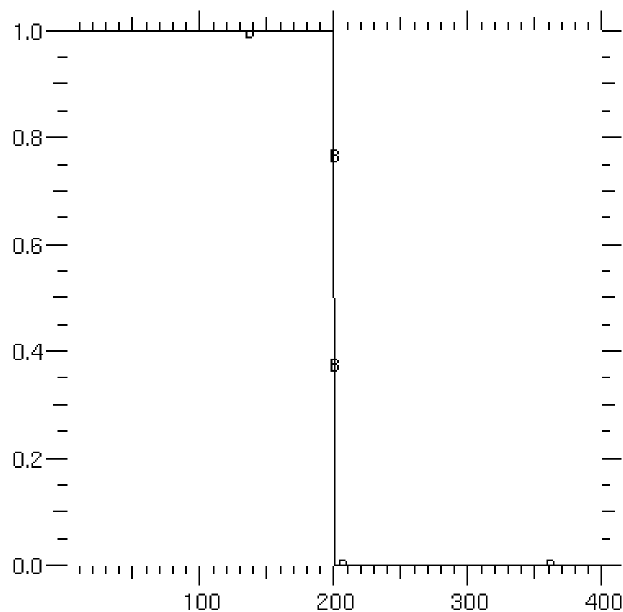
and set a step-function initial condition

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

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

If we are running interactively, we'll want a viewer to see the results

```
>>> 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

$$\phi = \frac{1}{2}\left[1 - \tanh \frac{x - L/2}{2\sqrt{\kappa/W}}\right] \tag{12.2}$$

or

```
>>> x = mesh.cellCenters[0]
>>> analyticalArray = 0.5*(1 - numerix.tanh((x - L/2)/(2*numerix.sqrt(kappa/W))))
```

We treat the diffusion term $\kappa_\phi \nabla^2 \phi$ implicitly,

Note: "Diffusion" in *FiPy* is not limited to the movement of atoms, but rather refers to the spontaneous spreading of any quantity (e.g., solute, temperature, or in this case "phase") by flow "down" the gradient of that quantity.

The source term is

$$\begin{aligned} S = -\frac{\partial f}{\partial \phi} &= -\frac{W}{2} g'(\phi) - L \frac{T - T_M}{T_M} p'(\phi) \\ &= -\left[W\phi(1-\phi)(1-2\phi) + L\frac{T - T_M}{T_M} 30\phi^2(1-\phi)^2 \right] \\ &= m_\phi \phi(1-\phi) \end{aligned}$$

where $m_\phi \equiv -[W(1-2\phi) + 30\phi(1-\phi)L\frac{T-T_M}{T_M}]$.

The simplest approach is to add this source explicitly

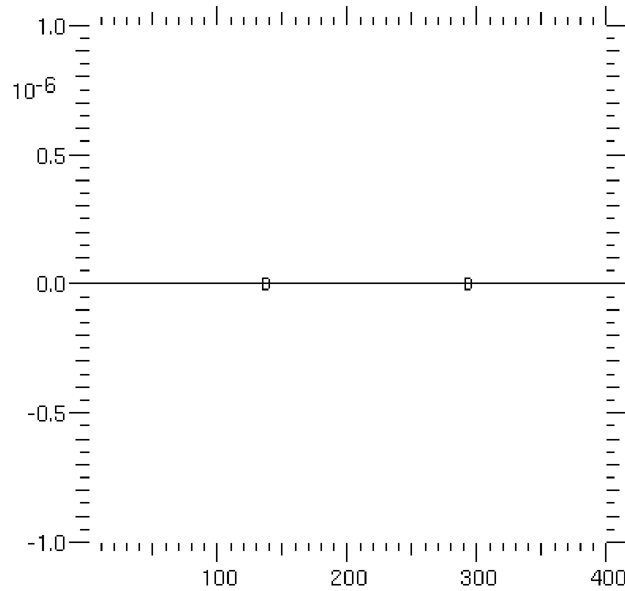
```
>>> mPhi = -((1 - 2 * phase) * W + 30 * phase * (1 - phase) * enthalpy)
>>> S0 = mPhi * phase * (1 - phase)
>>> eq = S0 + DiffusionTerm(coeff=kappa)
```

After solving this equation

```
>>> eq.solve(var = phase, solver=DummySolver())
```

we obtain the surprising result that ϕ is zero everywhere.

```
>>> print phase.allclose(analyticalArray, rtol = 1e-4, atol = 1e-4)
0
>>> if __name__ == '__main__':
...     viewer.plot()
...     raw_input("Fully explicit source. Press <return> to proceed...")
```



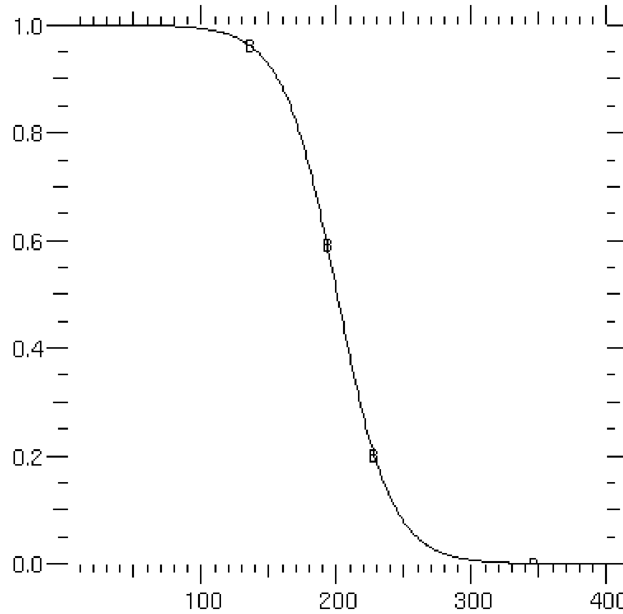
On inspection, we can see that this occurs because, for our step-function initial condition, $m_\phi = 0$ everwhere, hence we are actually only solving the simple implicit diffusion equation $\kappa_\phi \nabla^2 \phi = 0$, which has exactly the uninteresting solution we obtained.

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 (12.1)

```
>>> 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, dt=1.)
...     if __name__ == '__main__':
...         viewer.plot()
```

After 13 time steps, the solution has converged to the analytical solution

```
>>> print phase.allclose(analyticalArray, rtol = 1e-4, atol = 1e-4)
1
>>> 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 (12.2) 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 $\mathsf{M}\vec{\phi} - \vec{b} = 0$.

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 $S = S_\text{old} + \left.\frac{\partial S}{\partial \phi}\right|_\text{old} (\phi - \phi_\text{old}) = (S - \frac{\partial S}{\partial \phi} \phi)_\text{old} + \left.\frac{\partial S}{\partial \phi}\right|_\text{old} \phi$. Now, if our source term is represented by $S = S_0 + S_1 \phi$, then $S_1 = \left.\frac{\partial S}{\partial \phi}\right|_\text{old}$ and $S_0 = (S - \frac{\partial S}{\partial \phi} \phi)_\text{old} = S_\text{old} - S_1 \phi_\text{old}$. 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

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

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.cellCenters[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)

>>> if __name__ == '__main__':
...     displacement = L * 0.1
... else:
...     displacement = L * 0.025

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

and make a new viewer

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

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

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

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

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

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

and then quench by 1 K

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

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

```
>>> velocity = beta * abs(Tm - T()) # cm / s
>>> timeStep = .1 * dx / velocity # s
>>> elapsed = 0
>>> while elapsed < displacement / velocity:
...     phase.updateOld()
...     res = 1e+10
...     while res > 1e-5:
...         res = eq.sweep(var=phase, dt=timeStep)
...     elapsed += timeStep
...     if __name__ == '__main__':
...         viewer2.plot()
```

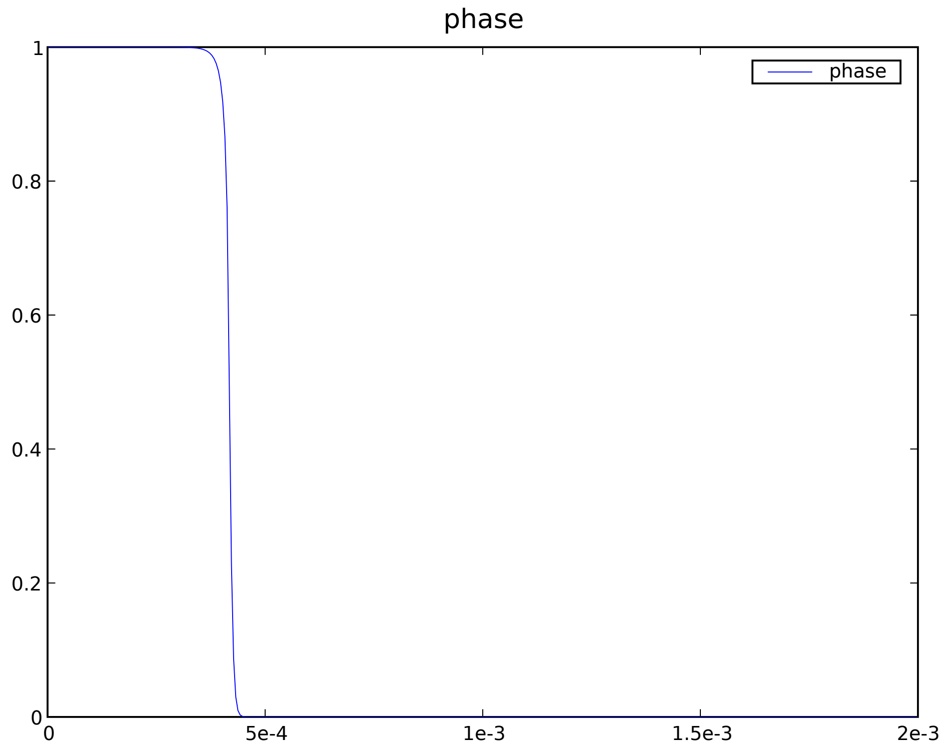
A hyperbolic tangent is not an exact steady-state solution given the quintic polynomial we chose for the p function, but it gives a reasonable approximation.

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

If we had made another common choice of $p(\phi) = \phi^2(3 - 2\phi)$, we would have found much better agreement, as that case does give an exact $\tanh$ solution in steady state. If SciPy is available, another way to compare against the expected result is to do a least-squared fit to determine the interface velocity and thickness

```
>>> try:
...     def tanhResiduals(p, y, x, t):
...         V, d = p
...         return y - 0.5 * (1 - numerix.tanh((x - V * t - L / 2.) / (2*d)))
...     from scipy.optimize import leastsq
...     x =  mesh.cellCenters[0]
...     (V_fit, d_fit), msg = leastsq(tanhResiduals, [L/2., delta],
...                                   args=(phase.globalValue, x.globalValue, elapsed))
... except ImportError:
...     V_fit = d_fit = 0
...     print "The SciPy library is unavailable to fit the interface \
... thickness and velocity"
```

```
>>> print abs(1 - V_fit / velocity) < 4.1e-2
True
>>> print abs(1 - d_fit / delta) < 2e-2
True

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



12.2 examples.phase.binaryCoupled

Simultaneously solve a phase-field evolution and solute diffusion problem in one-dimension.

It is straightforward to extend a phase field model to include binary alloys. As in `examples.phase.simple`, we will examine a 1D problem

```
>>> from fipy import *

>>> 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 [BoettingerReview: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

$$\frac{\partial \phi}{\partial t} = M_\phi \left(\kappa_\phi \nabla^2 \phi - \frac{\partial f}{\partial \phi} \right) \tag{12.3}$$

for phase and

$$\frac{\partial C}{\partial t} = \nabla \cdot \left(M_C \nabla \frac{\partial f}{\partial C} \right) \tag{12.4}$$

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,

$$\begin{aligned} f(\phi, C, T) &= (1 - C) f_A(\phi, T) + C f_B(\phi, T) \\ &\quad + RT \left[(1 - C) \ln(1 - C) + C \ln C \right] + C(1 - C) \left[\Omega_S p(\phi) + \Omega_L (1 - p(\phi)) \right] \end{aligned}$$

where

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

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

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

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

[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`

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

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

$$f_A^L(T) = 0$$
$$f_A^S(T) - f_A^L(T) = \frac{L_A \left(T - T_M^A\right)}{T_M^A}$$

and likewise for component B.

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

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

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

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

With these assumptions

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

and

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

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 of p and g with respect to ϕ

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

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

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

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

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

```
>>> deltaA = deltaB = 1.5 * dx
>>> sigmaA = 3.7e-5 # J / cm**2
>>> sigmaB = 2.9e-5 # J / cm**2
>>> betaA = 0.33 # cm / (K s)
```

```
>>> betaB = 0.39 # cm / (K s)
>>> kappaA = 6 * sigmaA * deltaA # J / cm
>>> kappaB = 6 * sigmaB * deltaB # J / cm
>>> WA = 6 * sigmaA / deltaA # J / cm**3
>>> WB = 6 * sigmaB / deltaB # J / cm**3
```

and define the averages

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

We can now linearize the source exactly as before

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

Using the same gradient energy coefficient and phase field mobility

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

we define the phase field equation

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

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

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

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

for j in range(cells):
    diffusion = (flux[j+.5] - flux[j-.5]) / dx
```

where we neglect the details of the outer boundaries (`j = 0` and `j = N`) or exactly how to translate `j+.5` or `j-.5` into an array index, much less the complexities of higher dimensions. FiPy can handle all of these issues automatically, so we could just write:

```
chemPotA = Vm * (enthalpyA * p(phase) + WA * g(phase)) + R * T * log(C)
chemPotB = Vm * (enthalpyB * p(phase) + WB * g(phase)) + R * T * log(1-C)
flux = Mc * (chemPotB - chemPotA).faceGrad
eq = TransientTerm() == flux.divergence
```

Although the second syntax would essentially work as written, such an explicit implementation would be very slow. In order to take advantage of *FiPy*'s implicit solvers, it is necessary to reduce Eq. (12.4) to the canonical form of Eq. (**??**), hence we must expand Eq. (12.2) as

$$\frac{\partial f}{\partial C} = \left[\frac{L_B\left(T - T_M^B\right)}{T_M^B} - \frac{L_A\left(T - T_M^A\right)}{T_M^A}\right] p(\phi) + \frac{RT}{V_m}\left[\ln C - \ln(1-C)\right] + \frac{W_B - W_A}{2} g(\phi)$$

In either bulk phase, $\nabla p(\phi) = \nabla g(\phi) = 0$, so we can then reduce Eq. (12.4) to

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

and, by comparison with Fick's second law

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

we can associate the mobility M_C with the intrinsic diffusivity D_C by $M_C \equiv D_C C(1-C)V_m/RT$ and write Eq. (12.4) as

$$\begin{aligned}\frac{\partial C}{\partial t} &= \nabla \cdot \left(D_C \nabla C \right) \\ &\quad + \nabla \cdot \left(\frac{D_C C(1-C)V_m}{RT} \left\{ \left[\frac{L_B\left(T - T_M^B\right)}{T_M^B} - \frac{L_A\left(T - T_M^A\right)}{T_M^A} \right] \nabla p(\phi) + \frac{W_B - W_A}{2} \nabla g(\phi) \right\} \right). \\ &= \nabla \cdot \left(D_C \nabla C \right) \\ &\quad + \nabla \cdot \left(\frac{D_C C(1-C)V_m}{RT} \left\{ \left[\frac{L_B\left(T - T_M^B\right)}{T_M^B} - \frac{L_A\left(T - T_M^A\right)}{T_M^A} \right] p'(\phi) + \frac{W_B - W_A}{2} g'(\phi) \right\} \nabla \phi \right).\end{aligned}$$

The first term is clearly a `DiffusionTerm` in C. The second is a `DiffusionTerm` in ϕ with a diffusion coefficent

$$D_\phi(C, \phi) = \frac{D_C C(1-C)V_m}{RT} \left\{ \left[\frac{L_B\left(T - T_M^B\right)}{T_M^B} - \frac{L_A\left(T - T_M^A\right)}{T_M^A} \right] p'(\phi) + \frac{W_B - W_A}{2} g'(\phi) \right\},$$

such that

$$\frac{\partial C}{\partial t} = \nabla \cdot \left(D_C \nabla C \right) + \nabla \cdot \left(D_\phi \nabla \phi \right)$$

or

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

>>> Dphi = ((Dc * C.harmonicFaceValue * (1 - C.harmonicFaceValue) * Vm / (R * T))
...         * ((enthalpyB - enthalpyA) * pPrime(phase.arithmeticFaceValue)
...            + 0.5 * (WB - WA) * gPrime(phase.arithmeticFaceValue)))

>>> diffusionEq = (TransientTerm(var=C)
...                == DiffusionTerm(coeff=Dc, var=C)
...                + DiffusionTerm(coeff=Dphi, var=phase))

>>> eq = phaseEq & diffusionEq
```

We initialize the phase field to a step function in the middle of the domain

```
>>> phase.setValue(1.)
>>> phase.setValue(0., where=mesh.cellCenters[0] > L/2.)
```

and start with a uniform composition field $C = 1/2$

```
>>> C.setValue(0.5)
```

In equilibrium, $\mu_A(0, C_L, T) = \mu_A(1, C_S, T)$ and $\mu_B(0, C_L, T) = \mu_B(1, C_S, T)$ and, for ideal solutions, we can deduce the liquidus and solidus compositions as

$$C_L = \frac{1 - \exp\left(-\frac{L_A\left(T-T_M^A\right)}{T_M^A}\frac{V_m}{RT}\right)}{\exp\left(-\frac{L_B\left(T-T_M^B\right)}{T_M^B}\frac{V_m}{RT}\right) - \exp\left(-\frac{L_A\left(T-T_M^A\right)}{T_M^A}\frac{V_m}{RT}\right)}$$

$$C_S = \exp\left(-\frac{L_B\left(T-T_M^B\right)}{T_M^B}\frac{V_m}{RT}\right) C_L$$

```
>>> Cl = (1. - numerix.exp(-enthalpyA * Vm / (R * T))) \
...    / (numerix.exp(-enthalpyB * Vm / (R * T)) - numerix.exp(-enthalpyA * Vm / (R * T)))
>>> Cs = numerix.exp(-enthalpyB * Vm / (R * T)) * Cl
```

The phase fraction is predicted by the lever rule

```
>>> Cavg = C.cellVolumeAverage
>>> fraction = (Cl - Cavg) / (Cl - Cs)
```

For the special case of `fraction = Cavg = 0.5`, a little bit of algebra reveals that the temperature that leaves the phase fraction unchanged is given by

```
>>> T.setValue((LA + LB) * TmA * TmB / (LA * TmB + LB * TmA))
```

In this simple, binary, ideal solution case, we can derive explicit expressions for the solidus and liquidus compositions. In general, this may not be possible or practical. In that event, the root-finding facilities in SciPy can be used.

We'll need a function to return the two conditions for equilibrium

$$0 = \mu_A(1, C_S, T) - \mu_A(0, C_L, T) = \frac{L_A\left(T - T_M^A\right)}{T_M^A} V_m + RT\ln(1 - C_S) - RT\ln(1 - C_L)$$

$$0 = \mu_B(1, C_S, T) - \mu_B(0, C_L, T) = \frac{L_B\left(T - T_M^B\right)}{T_M^B} V_m + RT\ln C_S - RT\ln C_L$$

```
>>> def equilibrium(C):
...     return [numerix.array(enthalpyA * Vm
...                           + R * T * numerix.log(1 - C[0])
...                           - R * T * numerix.log(1 - C[1])),
...             numerix.array(enthalpyB * Vm
...                           + R * T * numerix.log(C[0])
...                           - R * T * numerix.log(C[1]))]
```

and we'll have much better luck if we also supply the Jacobian

$$\begin{bmatrix} \frac{\partial(\mu_A^S - \mu_A^L)}{\partial C_S} & \frac{\partial(\mu_A^S - \mu_A^L)}{\partial C_L} \\ \frac{\partial(\mu_B^S - \mu_B^L)}{\partial C_S} & \frac{\partial(\mu_B^S - \mu_B^L)}{\partial C_L} \end{bmatrix} = RT \begin{bmatrix} -\frac{1}{1-C_S} & \frac{1}{1-C_L} \\ \frac{1}{C_S} & -\frac{1}{C_L} \end{bmatrix}$$

```
>>> def equilibriumJacobian(C):
...     return R * T * numerix.array([[-1. / (1 - C[0]), 1. / (1 - C[1])],
...                                   [ 1. / C[0],       -1. / C[1]]])
```

```
>>> try:
...     from scipy.optimize import fsolve
...     CsRoot, ClRoot = fsolve(func=equilibrium, x0=[0.5, 0.5],
...                             fprime=equilibriumJacobian)
... except ImportError:
...     ClRoot = CsRoot = 0
...     print "The SciPy library is not available to calculate the solidus and \
... liquidus concentrations"
```

```
>>> print Cl.allclose(ClRoot)
1
>>> print Cs.allclose(CsRoot)
1
```

We plot the result against the sharp interface solution

```
>>> sharp = CellVariable(name="sharp", mesh=mesh)
>>> x = mesh.cellCenters[0]
>>> sharp.setValue(Cs, where=x < L * fraction)
>>> sharp.setValue(Cl, where=x >= L * fraction)
```

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

Because the phase field interface will not move, and because we've seen in earlier examples that the diffusion problem is unconditionally stable, we need take only one very large timestep to reach equilibrium

```
>>> dt = 1.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 `LinearLUSolver` for this equation.

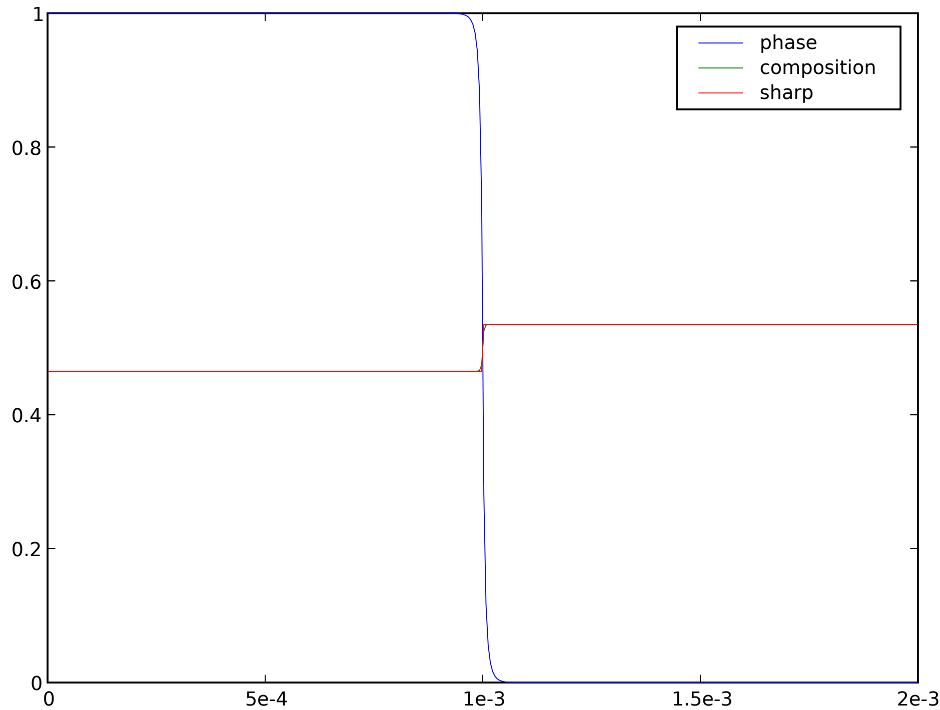
We now use the "`sweep()`" method instead of "`solve()`" because we require the residual.

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

```
>>> phase.updateOld()
>>> C.updateOld()
>>> res = 1.
>>> initialRes = None
```

```
>>> while res > 1e-4:
...     res = eq.sweep(dt=dt, solver=solver)
...     if initialRes is None:
...         initialRes = res
...     res = res / initialRes
```

```
>>> 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.faceCenters[0]
>>> print Cs.allclose(C.faceValue[X.value==0], atol=1e-2)
True
>>> print Cl.allclose(C.faceValue[X.value==L], atol=1e-2)
True
```

and that the phase fraction remains unchanged

```
>>> print fraction.allclose(phase.cellVolumeAverage, atol=2e-4)
1
```

while conserving mass overall

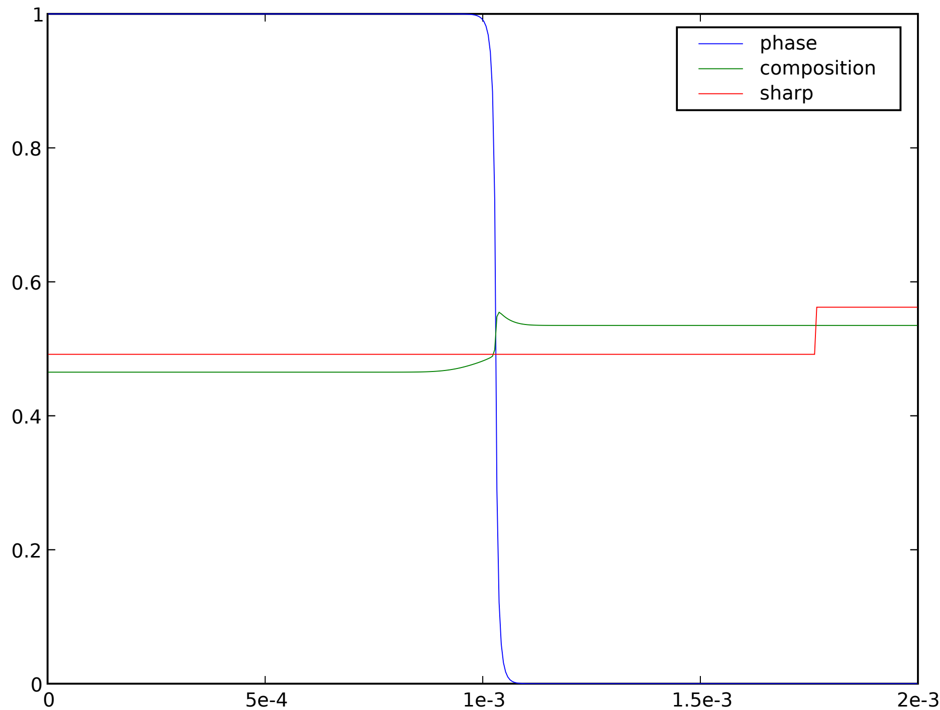
```
>>> print Cavg.allclose(0.5, atol=1e-8)
1
```

We now quench by ten degrees

```
>>> T.setValue(T() - 10.) # K

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

Because this lower temperature will induce the phase interface to move (solidify), we will need to take much smaller timesteps (the time scales of diffusion and of phase transformation compete with each other).

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

>>> if __name__ == '__main__':
...     timesteps = 100
... else:
...     timesteps = 10

>>> for i in range(timesteps):
...     phase.updateOld()
...     C.updateOld()
...     res = 1e+10
...     while res > 1e-3:
...         res = eq.sweep(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.

12.3 examples.phase.quaternary

Solve a phase-field evolution and diffusion of four species in one-dimension.

The same procedure used to construct the two-component phase field diffusion problem in `examples.phase.binary` can be used to build up a system of multiple components. Once again, we'll focus on 1D.

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

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

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

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

interstitial components $C_0 \ldots C_M$

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

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

```
>>> 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

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

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 $\bar{V}_0 = \cdots = \bar{V}_M = 0$ for the "interstitials" and $\bar{V}_{M+1} = \cdots = \bar{V}_N = 1$ 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

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

and

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

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 of p and g with respect to ϕ

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

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

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

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

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

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

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

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

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

We create the phase equation

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

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

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

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

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

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

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

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

The canonical form of the substitutional diffusion equations is

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

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

We start with a sharp phase boundary

$$\xi = \begin{cases} 1 & \text{for } x \le L/2, \\ 0 & \text{for } x > L/2, \end{cases}$$

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

and with uniform concentration fields, initially equal to the average of the solidus and liquidus concentrations

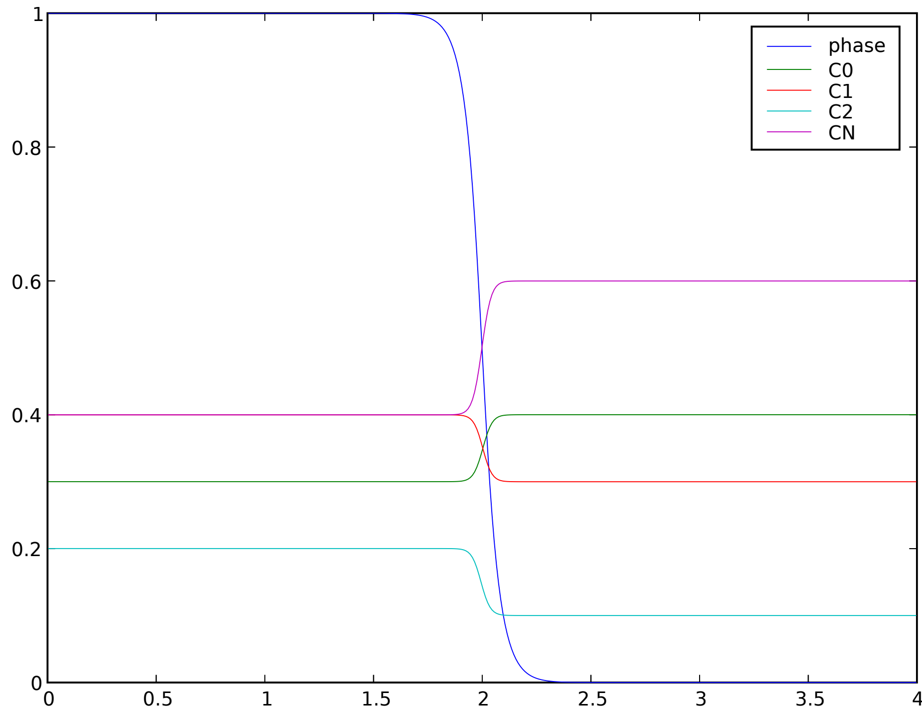
```
>>> for Cj in interstitials + substitutionals:
...     Cj.setValue((Cj.S + Cj.L) / 2.)
```

If we're running interactively, we create a viewer

```
>>> if __name__ == '__main__':
...     viewer = Viewer(vars=([phase]
...                           + interstitials + substitutionals
...                           + [solvent]),
...                     datamin=0, datamax=1)
...     viewer.plot()
```

and again iterate to equilibrium

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

>>> dt = 10000
>>> for i in range(5):
...     for field in [phase] + substitutionals + interstitials:
...         field.updateOld()
...     phase.equation.solve(var = phase, dt = dt)
...     for field in substitutionals + interstitials:
...         field.equation.solve(var = field,
...                              dt = dt,
...                              solver = solver)
...     if __name__ == '__main__':
...         viewer.plot()
```



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

```
>>> X = mesh.faceCenters[0]
>>> print numerix.allclose(phase.faceValue[X.value==0], 1.0, rtol = 1e-5, atol = 1e-5)
True
>>> print numerix.allclose(phase.faceValue[X.value==L], 0.0, rtol = 1e-5, atol = 1e-5)
True
```

and that the concentration fields have appropriately segregated into their equilibrium values in each phase

```
>>> equilibrium = True
>>> for Cj in interstitials + substitutionals:
...     equilibrium &= numerix.allclose(Cj.faceValue[X.value==0], Cj.S, rtol = 3e-3, atol = 3e-3).val
...     equilibrium &= numerix.allclose(Cj.faceValue[X.value==L], Cj.L, rtol = 3e-3, atol = 3e-3).val
>>> print equilibrium
True
```

12.4 examples.phase.anisotropy

Solve a dendritic solidification problem.

To convert a liquid material to a solid, it must be cooled to a temperature below its melting point (known as "undercooling" or "supercooling"). The rate of solidification is often assumed (and experimentally found) to be proportional to the undercooling. Under the right circumstances, the solidification front can become unstable, leading to dendritic patterns. Warren, Kobayashi, Lobkovsky and Carter [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.old) / dt)
```

The governing equation for the phase field is

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

where

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

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

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

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

```
>>> alpha = 0.015
>>> c = 0.02
>>> N = 6.
>>> theta = numerix.pi / 8.
>>> psi = theta + numerix.arctan2(phase.faceGrad[1],
...                               phase.faceGrad[0])
```

```
>>> Phi = numerix.tan(N * psi / 2)
>>> PhiSq = Phi**2
>>> beta = (1. - PhiSq) / (1. + PhiSq)
>>> DbetaDpsi = -N * 2 * Phi / (1 + PhiSq)
>>> Ddia = (1.+ c * beta)
>>> Doff = c * DbetaDpsi
>>> I0 = Variable(value=((1,0), (0,1)))
>>> I1 = Variable(value=((0,-1), (1,0)))
>>> D = alpha**2 * (1.+ c * beta) * (Ddia * I0 + Doff * I1)
```

With these expressions defined, we can construct the phase field equation as

```
>>> tau = 3e-4
>>> kappa1 = 0.9
>>> kappa2 = 20.
>>> phaseEq = (TransientTerm(tau)
...             == DiffusionTerm(D)
...             + ImplicitSourceTerm((phase - 0.5 - kappa1 / numerix.pi * numerix.arctan(kappa2 * dT))
...                                  * (1 - phase)))
```

We seed a circular solidified region in the center

```
>>> radius = dx * 5.
>>> C = (nx * dx / 2, ny * dy / 2)
>>> x, y = mesh.cellCenters
>>> phase.setValue(1., where=((x - C[0])**2 + (y - C[1])**2) < radius**2)
```

and quench the entire simulation domain below the melting point

```
>>> dT.setValue(-0.5)
```

In a real solidification process, dendritic branching is induced by small thermal fluctuations along an otherwise smooth surface, but the granularity of the `Mesh` is enough "noise" in this case, so we don't need to explicitly introduce randomness, the way we did in the Cahn-Hilliard problem.

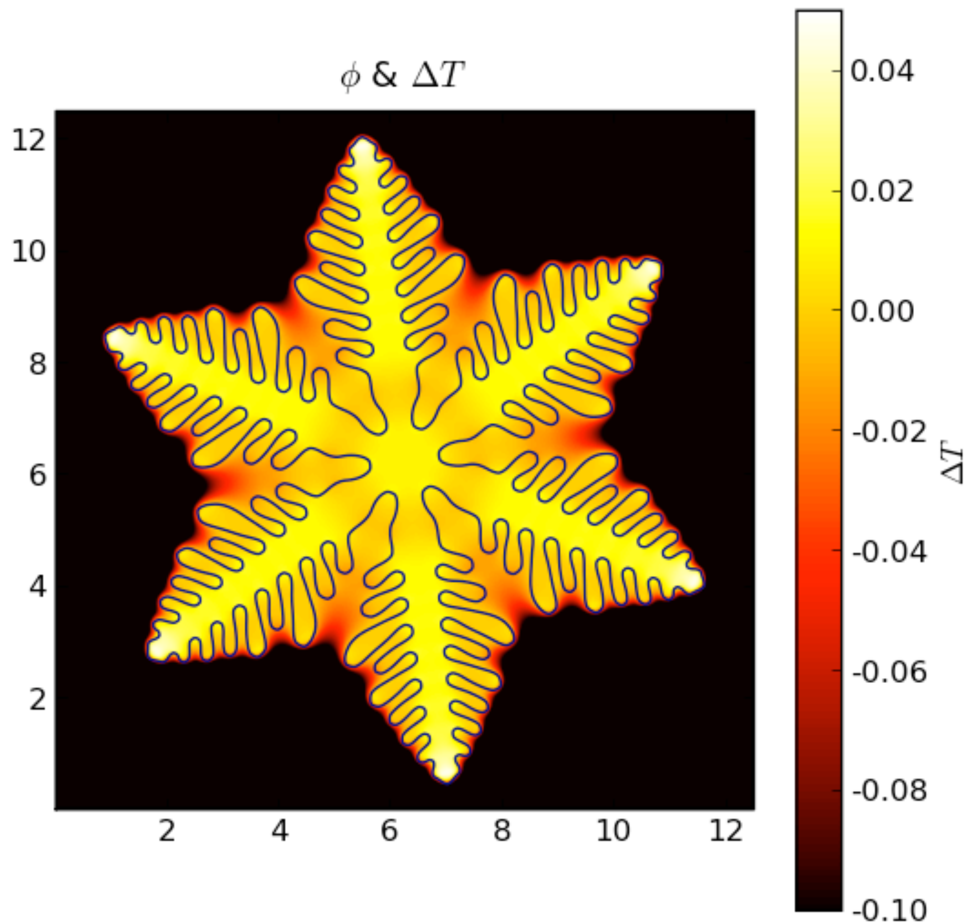
FiPy's viewers are utilitarian, striving to let the user see *something*, regardless of their operating system or installed packages, so you won't be able to simultaneously view two fields "out of the box", but, because all of Python is accessible and FiPy is object oriented, it is not hard to adapt one of the existing viewers to create a specialized display:

```
>>> if __name__ == "__main__":
...     try:
...         import pylab
...         class DendriteViewer(Matplotlib2DGridViewer):
...             def __init__(self, phase, dT, title=None, limits={}, **kwlimits):
...                 self.phase = phase
...                 self.contour = None
...                 Matplotlib2DGridViewer.__init__(self, vars=(dT,), title=title,
...                                                 cmap=pylab.cm.hot,
...                                                 limits=limits, **kwlimits)
...
...             def _plot(self):
...                 Matplotlib2DGridViewer._plot(self)
...
...                 if self.contour is not None:
...                     for c in self.contour.collections:
...                         c.remove()
...
...                 mesh = self.phase.mesh
...                 shape = mesh.shape
```

```
...                 x, y = mesh.cellCenters
...                 z = self.phase.value
...                 x, y, z = [a.reshape(shape, order="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].

12.5 examples.phase.impingement.mesh40x1

Solve for the impingement of two grains in one dimension.

In this example we solve a coupled phase and orientation equation on a one dimensional grid. This is another aspect of the model of Warren, Kobayashi, Lobkovsky and Carter [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

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

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 $0 \le x \le L_x$ and

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

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-peridocity, we must use `ModularVariable` instead of a `CellVariable`. A `ModularVariable` confines `theta` to $-\pi < \theta \le \pi$ by adding or subtracting 2π where necessary and by defining a new subtraction operator between two angles.

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

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

```
>>> theta.setValue(0., where=mesh.cellCenters[0] > Lx / 2.)
```

The `phase` equation is built in the following way.

```
>>> mPhiVar = phase - 0.5 + temperature * phase * (1 - phase)
```

The source term is linearized in the manner demonstrated in `examples.phase.simple` (Kobayashi, semi-implicit).

```
>>> thetaMag = theta.old.grad.mag
>>> implicitSource = mPhiVar * (phase - (mPhiVar < 0))
>>> implicitSource += (2 * s + epsilon**2 * thetaMag) * thetaMag
```

The `phase` equation is constructed.

```
>>> phaseEq = TransientTerm(phaseTransientCoeff) \
...   == ExplicitDiffusionTerm(alpha**2) \
...      - ImplicitSourceTerm(implicitSource) \
...      + (mPhiVar > 0) * mPhiVar * phase
```

The `theta` equation is built in the following way. The details for this equation are fairly involved, see J.A. Warren *et al.*. The main detail is that a source must be added to correct for the discretization of `theta` on the circle.

```
>>> phaseMod = phase + ( phase < thetaSmallValue ) * thetaSmallValue
>>> phaseModSq = phaseMod * phaseMod
>>> expo = epsilon * beta * theta.grad.mag
>>> expo = (expo < 100.) * (expo - 100.) + 100.
>>> pFunc = 1. + numerix.exp(-expo) * (mu / epsilon - 1.)
```

```
>>> phaseFace = phase.arithmeticFaceValue
>>> phaseSq = phaseFace * phaseFace
>>> gradMag = theta.faceGrad.mag
>>> eps = 1. / gamma / 10.
>>> gradMag += (gradMag < eps) * eps
>>> IGamma = (gradMag > 1. / gamma) * (1 / gradMag - gamma) + gamma
>>> diffusionCoeff = phaseSq * (s * IGamma + epsilon**2)
```

The source term requires the evaluation of the face gradient without the modular operator. `thetagetFaceGradNoMod()` evelautes the gradient without modular arithmetic.

```
>>> thetaGradDiff = theta.faceGrad - theta.faceGradNoMod
>>> sourceCoeff = (diffusionCoeff * thetaGradDiff).divergence
```

Finally the `theta` equation can be constructed.

```
>>> thetaEq = TransientTerm(thetaTransientCoeff * phaseModSq * pFunc) == \
...           DiffusionTerm(diffusionCoeff) \
...           + sourceCoeff
```

If the example is run interactively, we create viewers for the phase and orientation variables.

```
>>> if __name__ == '__main__':
...     phaseViewer = Viewer(vars=phase, datamin=0., datamax=1.)
...     thetaProductViewer = Viewer(vars=theta,
...                                 datamin=-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()
...     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 = numerix.loadtxt(os.path.splitext(__file__)[0] + '.gz')
>>> testData = CellVariable(mesh=mesh, value=testData)
>>> print theta.allclose(testData)
1
```

12.6 examples.phase.impingement.mesh20x20

Solve for the impingement of four grains in two dimensions.

In the following examples, we solve the same set of equations as in `examples.phase.impingement.mesh40x1` with different initial conditions and a 2D mesh:

```
>>> from fipy.tools.parser import parse

>>> numberOfElements = parse('--numberOfElements', action = 'store',
...                          type = 'int', default = 400)
>>> numberOfSteps = parse('--numberOfSteps', action = 'store',
...                       type = 'int', default = 10)

>>> from fipy import *

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

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

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

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

The parameters for this example are

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

The system is held isothermal at

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

and is initialized to liquid everywhere

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

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

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

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

```
>>> x, y = mesh.cellCenters
>>> for a, b, thetaValue in ((0., 0.,  2. * numerix.pi / 3.),
...                          (L, 0., -2. * numerix.pi / 3.),
...                          (0., L, -2. * numerix.pi / 3. + 0.3),
...                          (L, L,  2. * numerix.pi / 3.)):
...     segment = (x - a)**2 + (y - b)**2 < (L / 2.)**2
...     phase.setValue(1., where=segment)
...     theta.setValue(thetaValue, where=segment)
```

The `phase` equation is built in the following way. The source term is linearized in the manner demonstrated in `examples.phase.simple` (Kobayashi, semi-implicit). Here we use a function to build the equation, so that it can be reused later.

```
>>> def buildPhaseEquation(phase, theta):
...
...     mPhiVar = phase - 0.5 + temperature * phase * (1 - phase)
...     thetaMag = theta.old.grad.mag
...     implicitSource = mPhiVar * (phase - (mPhiVar < 0))
...     implicitSource += (2 * s + epsilon**2 * thetaMag) * thetaMag
...
...     return TransientTerm(phaseTransientCoeff) == \
...               ExplicitDiffusionTerm(alpha**2) \
...               - ImplicitSourceTerm(implicitSource) \
...               + (mPhiVar > 0) * mPhiVar * phase
```

```
>>> phaseEq = buildPhaseEquation(phase, theta)
```

The theta equation is built in the following way. The details for this equation are fairly involved, see J.A. Warren *et al.*. The main detail is that a source must be added to correct for the discretization of theta on the circle. The source term requires the evaluation of the face gradient without the modular operators.

```
>>> def buildThetaEquation(phase, theta):
...
...     phaseMod = phase + ( phase < thetaSmallValue ) * thetaSmallValue
...     phaseModSq = phaseMod * phaseMod
...     expo = epsilon * beta * theta.grad.mag
...     expo = (expo < 100.) * (expo - 100.) + 100.
...     pFunc = 1. + numerix.exp(-expo) * (mu / epsilon - 1.)
...
...     phaseFace = phase.arithmeticFaceValue
...     phaseSq = phaseFace * phaseFace
...     gradMag = theta.faceGrad.mag
...     eps = 1. / gamma / 10.
...     gradMag += (gradMag < eps) * eps
...     IGamma = (gradMag > 1. / gamma) * (1 / gradMag - gamma) + gamma
...     diffusionCoeff = phaseSq * (s * IGamma + epsilon**2)
...
...     thetaGradDiff = theta.faceGrad - theta.faceGradNoMod
...     sourceCoeff = (diffusionCoeff * thetaGradDiff).divergence
...
...     return TransientTerm(thetaTransientCoeff * phaseModSq * pFunc) == \
...                DiffusionTerm(diffusionCoeff) \
...                + sourceCoeff

>>> thetaEq = buildThetaEquation(phase, theta)
```

If the example is run interactively, we create viewers for the phase and orientation variables. Rather than viewing the raw orientation, which is not meaningful in the liquid phase, we weight the orientation by the phase

```
>>> if __name__ == '__main__':
...     phaseViewer = Viewer(vars=phase, datamin=0., datamax=1.)
...     thetaProd = -numerix.pi + phase * (theta + numerix.pi)
...     thetaProductViewer = Viewer(vars=thetaProd,
...                                 datamin=-numerix.pi, datamax=numerix.pi)
...     phaseViewer.plot()
...     thetaProductViewer.plot()
```

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

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

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

```
>>> for i in range(steps):
...     theta.updateOld()
...     thetaEq.solve(theta, dt=timeStepDuration, solver=GeneralSolver(iterations=2000, tolerance=1e-
...     phaseEq.solve(phase, dt=timeStepDuration, solver=GeneralSolver(iterations=2000, tolerance=1e-
...     if __name__ == '__main__':
...         phaseViewer.plot()
...         thetaProductViewer.plot()
```

The solution is compared against Ryo Kobayashi's test data

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

The following code shows how to restart a simulation from some saved data. First, reset the variables to their original values.

```
>>> phase.setValue(0)
>>> theta.setValue(-numerix.pi + 0.0001)
>>> x, y = mesh.cellCenters
>>> for a, b, thetaValue in ((0., 0.,  2. * numerix.pi / 3.),
...                          (L, 0., -2. * numerix.pi / 3.),
...                          (0., L, -2. * numerix.pi / 3. + 0.3),
...                          (L, L,  2. * numerix.pi / 3.)):
...     segment = (x - a)**2 + (y - b)**2 < (L / 2.)**2
...     phase.setValue(1., where=segment)
...     theta.setValue(thetaValue, where=segment)
```

Step through half the time steps.

```
>>> for i in range(steps // 2):
...     theta.updateOld()
...     thetaEq.solve(theta, dt=timeStepDuration, solver=GeneralSolver(iterations=2000, tolerance=1e-
...     phaseEq.solve(phase, dt=timeStepDuration, solver=GeneralSolver(iterations=2000, tolerance=1e-
```

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

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

We save the variables to disk.

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

and then recall them to test the data pickling mechanism

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

and finish the iterations,

```
>>> for i in range(steps // 2):
...     newTheta.updateOld()
...     newThetaEq.solve(newTheta, dt=timeStepDuration, solver=GeneralSolver(iterations=2000, tolerar
...     newPhaseEq.solve(newPhase, dt=timeStepDuration, solver=GeneralSolver(iterations=2000, tolerar
```

The solution is compared against Ryo Kobayashi's test data

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

12.7 examples.phase.polyxtal

Solve the dendritic growth of nuclei and subsequent grain impingement.

To convert a liquid material to a solid, it must be cooled to a temperature below its melting point (known as "undercooling" or "supercooling"). The rate of solidification is often assumed (and experimentally found) to be proportional to the undercooling. Under the right circumstances, the solidification front can become unstable, leading to dendritic patterns. Warren, Kobayashi, Lobkovsky and Carter [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 = 200
... else:
...     nx = ny = 200
>>> mesh = Grid2D(dx=dx, dy=dy, nx=nx, ny=ny)
```

and we'll take fixed timesteps

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

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

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

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

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

and an orientation $-\pi < \theta \leq \pi$

```
>>> theta = ModularVariable(name=r'$\theta$', mesh=mesh, hasOld=True)
>>> theta.value = -numerix.pi + 0.0001
```

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

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

$$\frac{\partial \Delta T}{\partial t} = D_T \nabla^2 \Delta T + \frac{\partial \phi}{\partial t} + c\left(T_0 - T\right)$$

```
>>> DT = 2.25
>>> q = Variable(0.)
>>> T_0 = -0.1
>>> heatEq = (TransientTerm()
...           == DiffusionTerm(DT)
...           + (phase - phase.old) / dt
...           + q * T_0 - ImplicitSourceTerm(q))
```

The governing equation for the phase field is

$$\tau_\phi \frac{\partial \phi}{\partial t} = \nabla \cdot \mathsf{D} \nabla \phi + \phi(1-\phi) m(\phi, \Delta T) - 2 s \phi |\nabla \theta| - \epsilon^2 \phi |\nabla \theta|^2$$

where

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

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

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

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

```
>>> alpha = 0.015
>>> c = 0.02
>>> N = 4.

>>> psi = theta.arithmeticFaceValue + numerix.arctan2(phase.faceGrad[1],
...                                                    phase.faceGrad[0])
>>> Phi = numerix.tan(N * psi / 2)
>>> PhiSq = Phi**2
>>> beta = (1. - PhiSq) / (1. + PhiSq)
>>> DbetaDpsi = -N * 2 * Phi / (1 + PhiSq)
>>> Ddia = (1.+ c * beta)

>>> Doff = c * DbetaDpsi
>>> I0 = Variable(value=((1,0), (0,1)))
>>> I1 = Variable(value=((0,-1), (1,0)))
>>> D = alpha**2 * Ddia * (Ddia * I0 + Doff * I1)
```

With these expressions defined, we can construct the phase field equation as

```
>>> tau_phase = 3e-4
>>> kappa1 = 0.9
>>> kappa2 = 20.
>>> epsilon = 0.008
>>> s = 0.01
>>> thetaMag = theta.grad.mag
>>> phaseEq = (TransientTerm(tau_phase)
...            == DiffusionTerm(D)
...            + ImplicitSourceTerm((phase - 0.5 - kappa1 / numerix.pi * numerix.arctan(kappa2 * dT))
...                                 * (1 - phase)
...                                 - (2 * s + epsilon**2 * thetaMag) * thetaMag))
```

The governing equation for orientation is given by

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

where

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

The `theta` equation is built in the following way. The details for this equation are fairly involved, see J.A. Warren *et al.*. The main detail is that a source must be added to correct for the discretization of `theta` on the circle.

```
>>> tau_theta = 3e-3
>>> mu = 1e3
>>> gamma = 1e3
>>> thetaSmallValue = 1e-6
>>> phaseMod = phase + ( phase < thetaSmallValue ) * thetaSmallValue
>>> beta_theta = 1e5
>>> expo = epsilon * beta_theta * theta.grad.mag
>>> expo = (expo < 100.) * (expo - 100.) + 100.
>>> Pfunc = 1. + numerix.exp(-expo) * (mu / epsilon - 1.)
```

```
>>> gradMagTheta = theta.faceGrad.mag
>>> eps = 1. / gamma / 10.
>>> gradMagTheta += (gradMagTheta < eps) * eps
>>> IGamma = (gradMagTheta > 1. / gamma) * (1 / gradMagTheta - gamma) + gamma
>>> v_theta = phase.arithmeticFaceValue * (s * IGamma + epsilon**2)
>>> D_theta = phase.arithmeticFaceValue**2 * (s * IGamma + epsilon**2)
```

The source term requires the evaluation of the face gradient without the modular operator. `thetagetFaceGradNoMod()` evaluates the gradient without modular arithmetic.

```
>>> thetaEq = (TransientTerm(tau_theta * phaseMod**2 * Pfunc)
...            == DiffusionTerm(D_theta)
...            + (D_theta * (theta.faceGrad - theta.faceGradNoMod)).divergence)
```

We seed a circular solidified region in the center

```
>>> x, y = mesh.cellCenters
>>> numSeeds = 10
>>> numerix.random.seed(12345)
>>> for Cx, Cy, orientation in numerix.random.random([numSeeds, 3]):
...     radius = dx * 5.
...     seed = ((x - Cx * nx * dx)**2 + (y - Cy * ny * dy)**2) < radius**2
...     phase[seed] = 1.
...     theta[seed] = numerix.pi * (2 * orientation - 1)
```

and quench the entire simulation domain below the melting point

```
>>> dT.setValue(-0.5)
```

In a real solidification process, dendritic branching is induced by small thermal fluctuations along an otherwise smooth surface, but the granularity of the `Mesh` is enough "noise" in this case, so we don't need to explicitly introduce randomness, the way we did in the Cahn-Hilliard problem.

FiPy's viewers are utilitarian, striving to let the user see *something*, regardless of their operating system or installed packages, so you the default color scheme of grain orientation won't be very informative "out of the box". Because all of Python is accessible and FiPy is object oriented, it is not hard to adapt one of the existing viewers to create a specialized display:

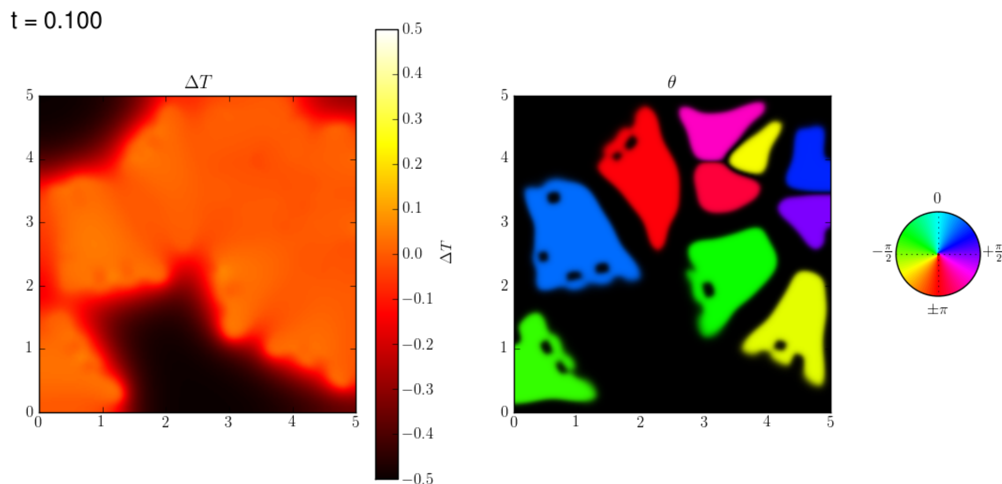
```
>>> if __name__ == "__main__":
...     try:
...         class OrientationViewer(Matplotlib2DGridViewer):
...             def __init__(self, phase, orientation, title=None, limits={}, **kwlimits):
...                 self.phase = phase
...                 Matplotlib2DGridViewer.__init__(self, vars=(orientation,), title=title,
...                                                 limits=limits, colorbar=None, **kwlimits)
...
...                 # make room for non-existent colorbar
...                 # stolen from matplotlib.colorbar.make_axes
...                 # https://github.com/matplotlib/matplotlib/blob
...                 #   /ec1cd2567521c105a451ce15e06de10715f8b54d/lib
...                 #   /matplotlib/colorbar.py#L838
...                 fraction = 0.15
...                 pb = self.axes.get_position(original=True).frozen()
...                 pad = 0.05
...                 x1 = 1.0-fraction
...                 pb1, pbx, pbcb = pb.splitx(x1-pad, x1)
...                 panchor = (1.0, 0.5)
...                 self.axes.set_position(pb1)
...                 self.axes.set_anchor(panchor)
```

```
...
...                 # make the gnomon
...                 fig = self.axes.get_figure()
...                 self.gnomon = fig.add_axes([0.85, 0.425, 0.15, 0.15], polar=True)
...                 self.gnomon.set_thetagrids([180, 270, 0, 90],
...                                            [r"$\pm\pi$", r"$-\frac{\pi}{2}$", "$0$", r"$+\frac{\p
...                                            frac=1.3)
...                 self.gnomon.set_theta_zero_location("N")
...                 self.gnomon.set_theta_direction(-1)
...                 self.gnomon.set_rgrids([1.], [""])
...                 N = 100
...                 theta = numerix.arange(-numerix.pi, numerix.pi, 2 * numerix.pi / N)
...                 radii = numerix.ones((N,))
...                 bars = self.gnomon.bar(theta, radii, width=2 * numerix.pi / N, bottom=0.0)
...                 colors = self._orientation_and_phase_to_rgb(orientation=numerix.array([theta]), p
...                 for c, t, bar in zip(colors[0], theta, bars):
...                     bar.set_facecolor(c)
...                     bar.set_edgecolor(c)
...
...             def _reshape(self, var):
...                 '''return values of var in an 2D array'''
...                 return numerix.reshape(numerix.array(var),
...                                        var.mesh.shape[::-1])[::-1]
...
...             @staticmethod
...             def _orientation_and_phase_to_rgb(orientation, phase):
...                 from matplotlib import colors
...
...                 hsv = numerix.empty(orientation.shape + (3,))
...                 hsv[..., 0] = (orientation / numerix.pi + 1) / 2.
...                 hsv[..., 1] = 1.
...                 hsv[..., 2] = phase
...
...                 return colors.hsv_to_rgb(hsv)
...
...             @property
...             def _data(self):
...                 '''convert phase and orientation to rgb image array
...
...                 orientation (-pi, pi) -> hue (0, 1)
...                 phase (0, 1) -> value (0, 1)
...                 '''
...                 orientation = self._reshape(self.vars[0])
...                 phase = self._reshape(self.phase)
...
...                 return self._orientation_and_phase_to_rgb(orientation, phase)
...
...             def _plot(self):
...                 self.image.set_data(self._data)
...
...         from matplotlib import pyplot
...         pyplot.ion()
...         w, h = pyplot.figaspect(1.)
...         fig = pyplot.figure(figsize=(2*w, h))
...         timer = fig.text(0.1, 0.9, "t = %.3f" % 0, fontsize=18)
...
...         viewer = MultiViewer(viewers=(MatplotlibViewer(vars=dT,
...                                                        cmap=pyplot.cm.hot,
...                                                        datamin=-0.5,
```

```
...                                                           datamax=0.5,
...                                                           axes=fig.add_subplot(121)),
...                                           OrientationViewer(phase=phase,
...                                                             orientation=theta,
...                                                             title=theta.name,
...                                                             axes=fig.add_subplot(122))))
...     except ImportError:
...         viewer = MultiViewer(viewers=(Viewer(vars=dT,
...                                              datamin=-0.5,
...                                              datamax=0.5),
...                                       Viewer(vars=phase,
...                                              datamin=0.,
...                                              datamax=1.),
...                                       Viewer(vars=theta,
...                                              datamin=-numerix.pi,
...                                              datamax=numerix.pi)))
>>> viewer.plot()
```

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

```
>>> if __name__ == "__main__":
...     total_time = 2.
... else:
...     total_time = dt * 10
>>> elapsed = 0.
>>> save_interval = 0.002
>>> save_at = save_interval

>>> while elapsed < total_time:
...     if elapsed > 0.3:
...         q.value = 100
...     phase.updateOld()
...     dT.updateOld()
...     theta.updateOld()
...     thetaEq.solve(theta, dt=dt)
...     phaseEq.solve(phase, dt=dt)
...     heatEq.solve(dT, dt=dt)
...     elapsed += dt
...     if __name__ == "__main__" and elapsed >= save_at:
...         timer.set_text("t = %.3f" % elapsed)
...         viewer.plot()
...         save_at += save_interval
```



The non-uniform temperature results from the release of latent heat at the solidifying interface. The dendrite arms grow fastest where the temperature gradient is steepest.

12.8 examples.phase.polyxtalCoupled

Simultaneously solve the dendritic growth of nuclei and subsequent grain impingement.

To convert a liquid material to a solid, it must be cooled to a temperature below its melting point (known as "undercooling" or "supercooling"). The rate of solidification is often assumed (and experimentally found) to be proportional to the undercooling. Under the right circumstances, the solidification front can become unstable, leading to dendritic patterns. Warren, Kobayashi, Lobkovsky and Carter [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 = 200
... else:
...     nx = ny = 200
>>> mesh = Grid2D(dx=dx, dy=dy, nx=nx, ny=ny)
```

and we'll take fixed timesteps

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

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

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

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

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

and an orientation $-\pi < \theta \leq \pi$

```
>>> theta = ModularVariable(name=r'$\theta$', mesh=mesh, hasOld=True)
>>> theta.value = -numerix.pi + 0.0001
```

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

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

$$\frac{\partial \Delta T}{\partial t} = D_T \nabla^2 \Delta T + \frac{\partial \phi}{\partial t} + c\left(T_0 - T\right)$$

```
>>> DT = 2.25
>>> q = Variable(0.)
>>> T_0 = -0.1
>>> heatEq = (TransientTerm(var=dT)
...           == DiffusionTerm(coeff=DT, var=dT)
...           + TransientTerm(var=phase)
...           + q * T_0 - ImplicitSourceTerm(coeff=q, var=dT))
```

The governing equation for the phase field is

$$\tau_\phi \frac{\partial \phi}{\partial t} = \nabla \cdot \mathsf{D} \nabla \phi + \phi(1-\phi) m(\phi, \Delta T) - 2 s \phi |\nabla \theta| - \epsilon^2 \phi |\nabla \theta|^2$$

where

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

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

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

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

```
>>> alpha = 0.015
>>> c = 0.02
>>> N = 4.

>>> psi = theta.arithmeticFaceValue + numerix.arctan2(phase.faceGrad[1],
...                                                   phase.faceGrad[0])
>>> Phi = numerix.tan(N * psi / 2)
>>> PhiSq = Phi**2
>>> beta = (1. - PhiSq) / (1. + PhiSq)
>>> DbetaDpsi = -N * 2 * Phi / (1 + PhiSq)
>>> Ddia = (1.+ c * beta)

>>> Doff = c * DbetaDpsi
>>> I0 = Variable(value=((1,0), (0,1)))
>>> I1 = Variable(value=((0,-1), (1,0)))
>>> D = alpha**2 * Ddia * (Ddia * I0 + Doff * I1)
```

With these expressions defined, we can construct the phase field equation as

```
>>> tau_phase = 3e-4
>>> kappa1 = 0.9
```

```
>>> kappa2 = 20.
>>> epsilon = 0.008
>>> s = 0.01
>>> thetaMag = theta.grad.mag
>>> phaseEq = (TransientTerm(coeff=tau_phase, var=phase)
...            == DiffusionTerm(coeff=D, var=phase)
...            + ImplicitSourceTerm(coeff=((phase - 0.5 - kappa1 / numerix.pi * numerix.arctan(kappa2
...                                         * (1 - phase)
...                                         - (2 * s + epsilon**2 * thetaMag) * thetaMag),
...                                  var=phase))
```

The governing equation for orientation is given by

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

where

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

The `theta` equation is built in the following way. The details for this equation are fairly involved, see J.A. Warren *et al.*. The main detail is that a source must be added to correct for the discretization of `theta` on the circle.

```
>>> tau_theta = 3e-3
>>> mu = 1e3
>>> gamma = 1e3
>>> thetaSmallValue = 1e-6
>>> phaseMod = phase + ( phase < thetaSmallValue ) * thetaSmallValue
>>> beta_theta = 1e5
>>> expo = epsilon * beta_theta * theta.grad.mag
>>> expo = (expo < 100.) * (expo - 100.) + 100.
>>> Pfunc = 1. + numerix.exp(-expo) * (mu / epsilon - 1.)

>>> gradMagTheta = theta.faceGrad.mag
>>> eps = 1. / gamma / 10.
>>> gradMagTheta += (gradMagTheta < eps) * eps
>>> IGamma = (gradMagTheta > 1. / gamma) * (1 / gradMagTheta - gamma) + gamma
>>> v_theta = phase.arithmeticFaceValue * (s * IGamma + epsilon**2)
>>> D_theta = phase.arithmeticFaceValue**2 * (s * IGamma + epsilon**2)
```

The source term requires the evaluation of the face gradient without the modular operator. `thetagetFaceGradNoMod()` evaluates the gradient without modular arithmetic.

```
>>> thetaEq = (TransientTerm(coeff=tau_theta * phaseMod**2 * Pfunc, var=theta)
...            == DiffusionTerm(coeff=D_theta, var=theta)
...            + PowerLawConvectionTerm(coeff=v_theta * (theta.faceGrad - theta.faceGradNoMod), var=p
```

We seed a circular solidified region in the center

```
>>> x, y = mesh.cellCenters
>>> numSeeds = 10
>>> numerix.random.seed(12345)
>>> for Cx, Cy, orientation in numerix.random.random([numSeeds, 3]):
...     radius = dx * 5.
...     seed = ((x - Cx * nx * dx)**2 + (y - Cy * ny * dy)**2) < radius**2
...     phase[seed] = 1.
...     theta[seed] = numerix.pi * (2 * orientation - 1)
```

and quench the entire simulation domain below the melting point

```
>>> dT.setValue(-0.5)
```

In a real solidification process, dendritic branching is induced by small thermal fluctuations along an otherwise smooth surface, but the granularity of the Mesh is enough "noise" in this case, so we don't need to explicitly introduce randomness, the way we did in the Cahn-Hilliard problem.

FiPy's viewers are utilitarian, striving to let the user see *something*, regardless of their operating system or installed packages, so you the default color scheme of grain orientation won't be very informative "out of the box". Because all of Python is accessible and FiPy is object oriented, it is not hard to adapt one of the existing viewers to create a specialized display:

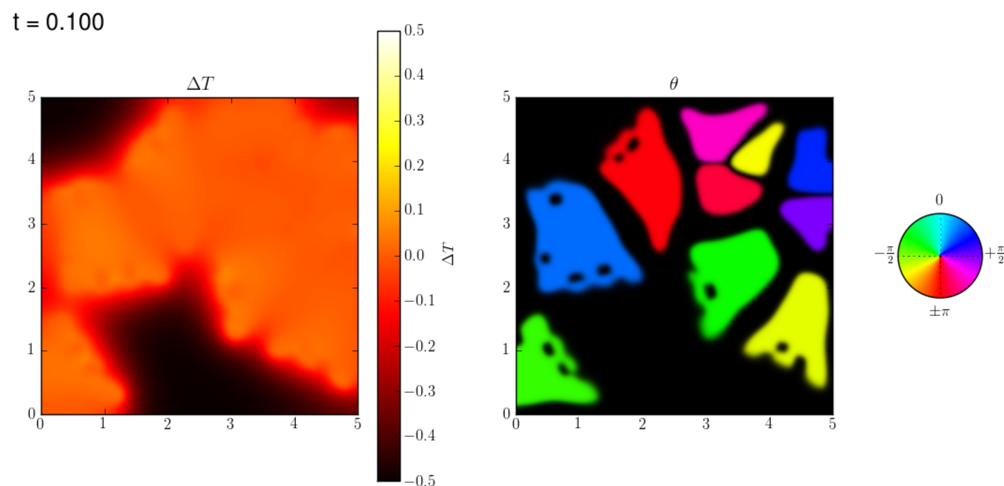
```
>>> if __name__ == "__main__":
...     try:
...         class OrientationViewer(Matplotlib2DGridViewer):
...             def __init__(self, phase, orientation, title=None, limits={}, **kwlimits):
...                 self.phase = phase
...                 Matplotlib2DGridViewer.__init__(self, vars=(orientation,), title=title,
...                                                 limits=limits, colorbar=None, **kwlimits)
...
...                 # make room for non-existent colorbar
...                 # stolen from matplotlib.colorbar.make_axes
...                 # https://github.com/matplotlib/matplotlib/blob
...                 #   /ec1cd2567521c105a451ce15e06de10715f8b54d/lib
...                 #   /matplotlib/colorbar.py#L838
...                 fraction = 0.15
...                 pb = self.axes.get_position(original=True).frozen()
...                 pad = 0.05
...                 x1 = 1.0-fraction
...                 pb1, pbx, pbcb = pb.splitx(x1-pad, x1)
...                 panchor = (1.0, 0.5)
...                 self.axes.set_position(pb1)
...                 self.axes.set_anchor(panchor)
...
...                 # make the gnomon
...                 fig = self.axes.get_figure()
...                 self.gnomon = fig.add_axes([0.85, 0.425, 0.15, 0.15], polar=True)
...                 self.gnomon.set_thetagrids([180, 270, 0, 90],
...                                            [r"$\pm\pi$", r"$-\frac{\pi}{2}$", "$0$", r"$+\frac{\p
...                                            frac=1.3)
...                 self.gnomon.set_theta_zero_location("N")
...                 self.gnomon.set_theta_direction(-1)
...                 self.gnomon.set_rgrids([1.], [""])
...                 N = 100
...                 theta = numerix.arange(-numerix.pi, numerix.pi, 2 * numerix.pi / N)
...                 radii = numerix.ones((N,))
...                 bars = self.gnomon.bar(theta, radii, width=2 * numerix.pi / N, bottom=0.0)
...                 colors = self._orientation_and_phase_to_rgb(orientation=numerix.array([theta]), p
...                 for c, t, bar in zip(colors[0], theta, bars):
...                     bar.set_facecolor(c)
...                     bar.set_edgecolor(c)
...
...             def _reshape(self, var):
...                 '''return values of var in an 2D array'''
...                 return numerix.reshape(numerix.array(var),
...                                        var.mesh.shape[::-1])[::-1]
...
...             @staticmethod
...             def _orientation_and_phase_to_rgb(orientation, phase):
```

```
...                 from matplotlib import colors
...
...                 hsv = numerix.empty(orientation.shape + (3,))
...                 hsv[..., 0] = (orientation / numerix.pi + 1) / 2.
...                 hsv[..., 1] = 1.
...                 hsv[..., 2] = phase
...
...                 return colors.hsv_to_rgb(hsv)
...
...             @property
...             def _data(self):
...                 '''convert phase and orientation to rgb image array
...
...                 orientation (-pi, pi) -> hue (0, 1)
...                 phase (0, 1) -> value (0, 1)
...                 '''
...                 orientation = self._reshape(self.vars[0])
...                 phase = self._reshape(self.phase)
...
...                 return self._orientation_and_phase_to_rgb(orientation, phase)
...
...             def _plot(self):
...                 self.image.set_data(self._data)
...
...         from matplotlib import pyplot
...         pyplot.ion()
...         w, h = pyplot.figaspect(1.)
...         fig = pyplot.figure(figsize=(2*w, h))
...         timer = fig.text(0.1, 0.9, "t = %.3f" % 0, fontsize=18)
...
...         viewer = MultiViewer(viewers=(MatplotlibViewer(vars=dT,
...                                                        cmap=pyplot.cm.hot,
...                                                        datamin=-0.5,
...                                                        datamax=0.5,
...                                                        axes=fig.add_subplot(121)),
...                                       OrientationViewer(phase=phase,
...                                                         orientation=theta,
...                                                         title=theta.name,
...                                                         axes=fig.add_subplot(122))))
...     except ImportError:
...         viewer = MultiViewer(viewers=(Viewer(vars=dT,
...                                              datamin=-0.5,
...                                              datamax=0.5),
...                                       Viewer(vars=phase,
...                                              datamin=0.,
...                                              datamax=1.),
...                                       Viewer(vars=theta,
...                                              datamin=-numerix.pi,
...                                              datamax=numerix.pi)))
>>> viewer.plot()
```

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

```
>>> eq = thetaEq & phaseEq & heatEq

>>> if __name__ == "__main__":
...     total_time = 2.
... else:
```

```
...     total_time = dt * 10
>>> elapsed = 0.
>>> save_interval = 0.002
>>> save_at = save_interval

>>> while elapsed < total_time:
...     if elapsed > 0.3:
...         q.value = 100
...     phase.updateOld()
...     dT.updateOld()
...     theta.updateOld()
...     eq.solve(dt=dt)
...     elapsed += dt
...     if __name__ == "__main__" and elapsed >= save_at:
...         timer.set_text("t = %.3f" % elapsed)
...         viewer.plot()
...         save_at += save_interval
```



The non-uniform temperature results from the release of latent heat at the solidifying interface. The dendrite arms grow fastest where the temperature gradient is steepest.

Chapter 13

Level Set Examples

`examples.levelSet.distanceFunction.mesh1D`	Create a level set variable in one dimension.
`examples.levelSet.distanceFunction.circle`	Solve the level set equation in two dimensions for a circle.
`examples.levelSet.advection.mesh1D`	Solve the distance function equation in one dimension and then advec
`examples.levelSet.advection.circle`	Solve a circular distance function equation and then advect it.

13.1 examples.levelSet.distanceFunction.mesh1D

Create a level set variable in one dimension.

The level set variable calculates its value over the domain to be the distance from the zero level set. This can be represented succinctly in the following equation with a boundary condition at the zero level set such that,

$$\frac{\partial \phi}{\partial x} = 1$$

with the boundary condition, $\phi = 0$ at $x = L/2$.

The solution to this problem will be demonstrated in the following script. Firstly, setup the parameters.

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

```
>>> dx = 0.5
>>> nx = 10
```

Construct the mesh.

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

Construct a *distanceVariable* object.

```
>>> var = DistanceVariable(name='level set variable',
...                        mesh=mesh,
...                        value=-1,
...                        hasOld=1)
>>> x = mesh.cellCenters[0]
>>> var.setValue(1, where=x > dx * nx / 2)
```

Once the initial positive and negative regions have been initialized the *calcDistanceFunction()* method can be used to recalculate *var* as a distance function from the zero level set.

```
>>> var.calcDistanceFunction()
```

The problem can then be solved by executing the `solve()` method of the equation.

```
>>> if __name__ == '__main__':
...     viewer = Viewer(vars=var, datamin=-5., datamax=5.)
...     viewer.plot()
```

The result can be tested with the following commands.

```
>>> print numerix.allclose(var, x - dx * nx / 2)
1
```

13.2 examples.levelSet.distanceFunction.circle

Solve the level set equation in two dimensions for a circle.

The 2D level set equation can be written,

$$|\nabla\phi| = 1$$

and the boundary condition for a circle is given by, $\phi = 0$ at $(x - L/2)^2 + (y - L/2)^2 = (L/4)^2$.

The solution to this problem will be demonstrated in the following script. Firstly, setup the parameters.

```
>>> from fipy import *

>>> 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, communicator=serial)
```

Construct a *distanceVariable* object.

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

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

>>> var.calcDistanceFunction()

>>> if __name__ == '__main__':
...     viewer = Viewer(vars=var, datamin=-5., datamax=5.)
...     viewer.plot()
```

The result can be tested with the following commands.

```
>>> dY = dy / 2.
>>> dX = dx / 2.
>>> mm = min (dX, dY)
>>> m1 = dY * dX / numerix.sqrt(dY**2 + dX**2)
>>> def evalCell(phix, phiy, dx, dy):
```

```
...     aa = dy**2 + dx**2
...     bb = -2 * ( phix * dy**2 + phiy * dx**2)
...     cc = dy**2 * phix**2 + dx**2 * phiy**2 - dx**2 * dy**2
...     sqr = numerix.sqrt(bb**2 - 4. * aa * cc)
...     return ((-bb - sqr) / 2. / aa,  (-bb + sqr) / 2. / aa)
>>> v1 = evalCell(-dY, -m1, dx, dy)[0]
>>> v2 = evalCell(-m1, -dX, dx, dy)[0]
>>> v3 = evalCell(m1,  m1,  dx, dy)[1]
>>> v4 = evalCell(v3, dY, dx, dy)[1]
>>> v5 = evalCell(dX, v3, dx, dy)[1]
>>> MASK = -1000.
>>> trialValues = CellVariable(mesh=mesh, value= \
...     numerix.array((
...     MASK,  MASK, MASK, MASK, MASK, MASK, MASK, MASK, MASK, MASK, MASK,
...     MASK,  MASK, MASK, MASK,-3*dY,-3*dY,-3*dY, MASK, MASK, MASK, MASK,
...     MASK,  MASK, MASK,   v1,  -dY,  -dY,  -dY,   v1, MASK, MASK, MASK,
...     MASK,  MASK,   v2,  -m1,   m1,   dY,   m1,  -m1,   v2, MASK, MASK,
...     MASK, -dX*3,  -dX,   m1,   v3,   v4,   v3,   m1,  -dX,-dX*3, MASK,
...     MASK, -dX*3,  -dX,   dX,   v5, MASK,   v5,   dX,  -dX,-dX*3, MASK,
...     MASK, -dX*3,  -dX,   m1,   v3,   v4,   v3,   m1,  -dX,-dX*3, MASK,
...     MASK,  MASK,   v2,  -m1,   m1,   dY,   m1,  -m1,   v2, MASK, MASK,
...     MASK,  MASK, MASK,   v1,  -dY,  -dY,  -dY,   v1, MASK, MASK, MASK,
...     MASK,  MASK, MASK, MASK,-3*dY,-3*dY,-3*dY, MASK, MASK, MASK, MASK,
...     MASK,  MASK, MASK, MASK, MASK, MASK, MASK, MASK, MASK, MASK, MASK), 'd'))

>>> var[numerix.array(trialValues == MASK)] = MASK
>>> print numerix.allclose(var, trialValues)
True
```

13.3 examples.levelSet.advection.mesh1D

Solve the distance function equation in one dimension and then advect it.

This example first solves the distance function equation in one dimension:

$$|\nabla\phi| = 1$$

with $\phi = 0$ at $x = L/5$.

The variable is then advected with,

$$\frac{\partial\phi}{\partial t} + \vec{u} \cdot \nabla\phi = 0$$

The scheme used in the *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, communicator=serial)
```

Construct a *distanceVariable* object.

```
>>> var = DistanceVariable(name='level set variable',
...                        mesh=mesh,
...                        value=-1.,
...                        hasOld=1)
>>> var.setValue(1., where=mesh.cellCenters[0] > interfacePosition)
>>> var.calcDistanceFunction()
```

The *advectionEquation* is constructed.

```
>>> advEqn = buildAdvectionEquation(advectionCoeff=velocity)
```

The problem can then be solved by executing a serious of time steps.

```
>>> if __name__ == '__main__':
...     viewer = Viewer(vars=var, datamin=-10., datamax=10.)
...     viewer.plot()
...     for step in range(steps):
...         var.updateOld()
...         advEqn.solve(var, dt=timeStepDuration)
...         viewer.plot()
```

The result can be tested with the following code:

```
>>> for step in range(steps):
...     var.updateOld()
...     advEqn.solve(var, dt=timeStepDuration)
>>> x = mesh.cellCenters[0]
>>> distanceTravelled = timeStepDuration * steps * velocity
>>> answer = x - interfacePosition - timeStepDuration * steps * velocity
>>> answer = numerix.where(x < distanceTravelled,
...                        x[0] - interfacePosition, answer)
>>> print var.allclose(answer)
1
```

13.4 examples.levelSet.advection.circle

Solve a circular distance function equation and then advect it.

This example first imposes a circular distance function:

$$\phi(x,y) = \left[\left(x - \frac{L}{2}\right)^2 + \left(y - \frac{L}{2}\right)^2\right]^{1/2} - \frac{L}{4}$$

The variable is advected with,

$$\frac{\partial \phi}{\partial t} + \vec{u} \cdot \nabla \phi = 0$$

The scheme used in the `_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.cellCenters
>>> initialArray = numerix.sqrt((x - L / 2.)**2 + (y - L / 2.)**2) - radius
>>> var.setValue(initialArray)
```

The advection equation is constructed.

```
>>> advEqn = buildAdvectionEquation(
...     advectionCoeff=velocity)
```

The problem can then be solved by executing a serious of time steps.

```
>>> if __name__ == '__main__':
...     viewer = Viewer(vars=var, datamin=-radius, datamax=radius)
...     viewer.plot()
...     for step in range(steps):
...         var.updateOld()
...         advEqn.solve(var, dt=timeStepDuration)
...         viewer.plot()
```

The result can be tested with the following commands.

```
>>> for step in range(steps):
...     var.updateOld()
...     advEqn.solve(var, dt=timeStepDuration)
>>> x = numerix.array(mesh.cellCenters[0])
>>> distanceTravelled = timeStepDuration * steps * velocity
>>> answer = initialArray - distanceTravelled
>>> answer = numerix.where(answer < 0., -1001., answer)
>>> solution = numerix.where(answer < 0., -1001., numerix.array(var))
>>> numerix.allclose(answer, solution, atol=4.7e-3)
1
```

If the advection equation is built with the `buildHigherOrderAdvectionEquation()` 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 = numerix.where(answer < 0., -1001., numerix.array(var))
>>> numerix.allclose(answer, solution, atol=1.02e-3)
1
```

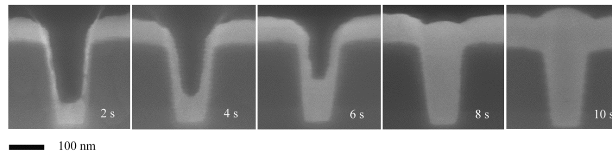
13.5 Superconformal Electrodeposition Examples

13.5.1 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.

13.5.2 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.



13.5.3 The CEAC Mechanism

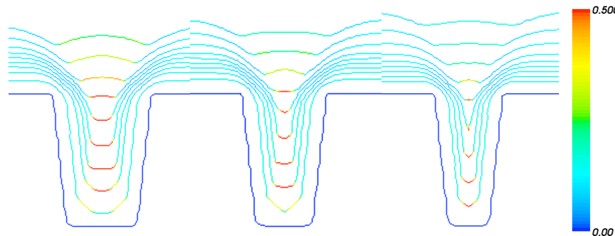
This process has been demonstrated to depend critically on the inclusion of additives in the electrolyte. Recent publications propose Curvature Enhanced Accelerator Coverage (CEAC) as the mechanism behind the superfilling process [NIST:damascene:2001]. In this mechanism, molecules that accelerate local 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].

13.5.4 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.simpleTrenchSystem`	Model electrochemical superfill using the CEAC mechanis
`examples.levelSet.electroChem.gold`	Model electrochemical superfill of gold using the CEAC m
`examples.levelSet.electroChem.leveler`	Model electrochemical superfill of copper with leveler and
`examples.levelSet.electroChem.howToWriteAScript`	Tutorial for writing an electrochemical superfill script.

13.6 examples.levelSet.electroChem.simpleTrenchSystem

Model electrochemical superfill using the CEAC mechanism.

This input file is a demonstration of the use of *FiPy* for modeling electrodeposition using the CEAC mechanism. The material properties and experimental parameters used are roughly those that have been previously published [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. To run with a different number of time steps change the `numberOfSteps` argument as follows,

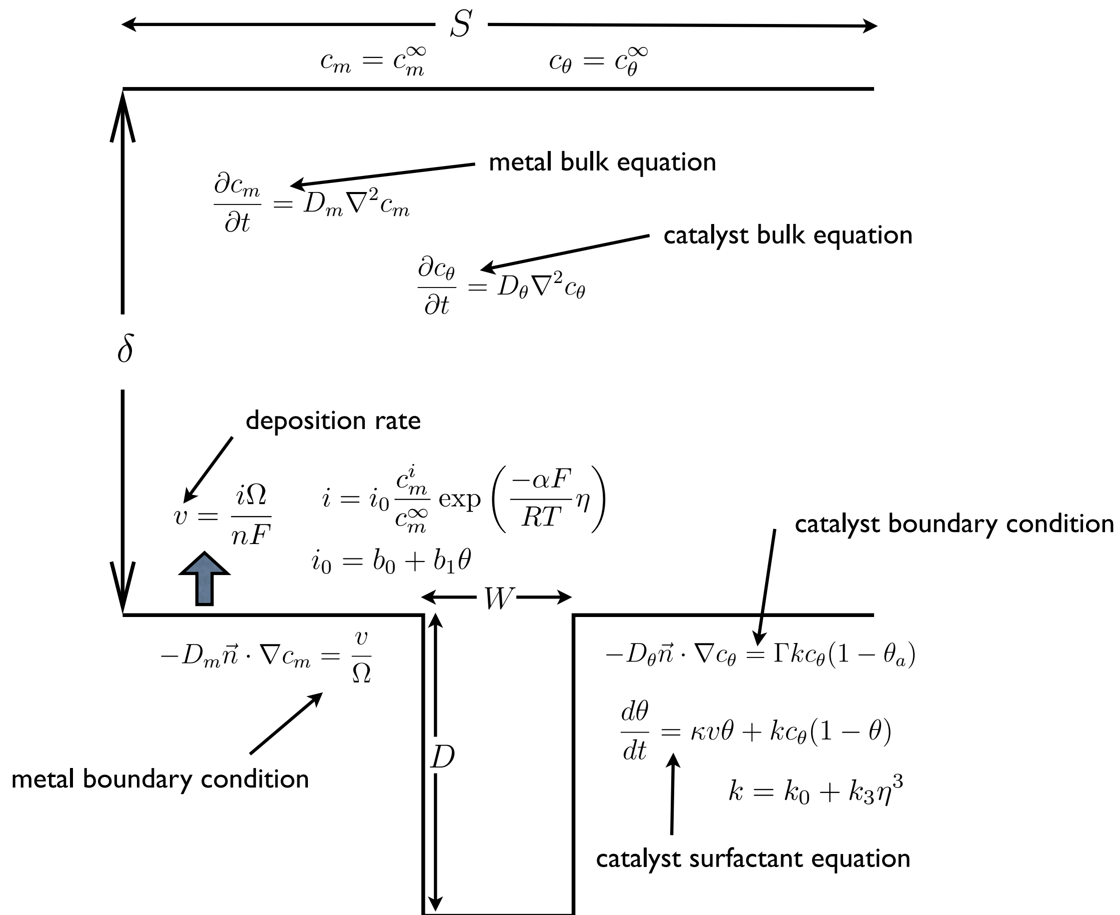
```
>>> runSimpleTrenchSystem(numberOfSteps=2, displayViewers=False)
1
```

Change the `displayViewers` argument to `True` if you wish to see the results displayed on the screen. Example `examples.levelSet.electroChem.simpleTrenchSystem` gives explanation for writing new scripts or modifying existing scripts that are encapsulated by functions.

Any argument parameter can be changed. For example if the initial catalyst coverage is not 0, then it can be reset,

```
>>> runSimpleTrenchSystem(numberOfSteps=2, catalystCoverage=0.1, displayViewers=False)
0
```

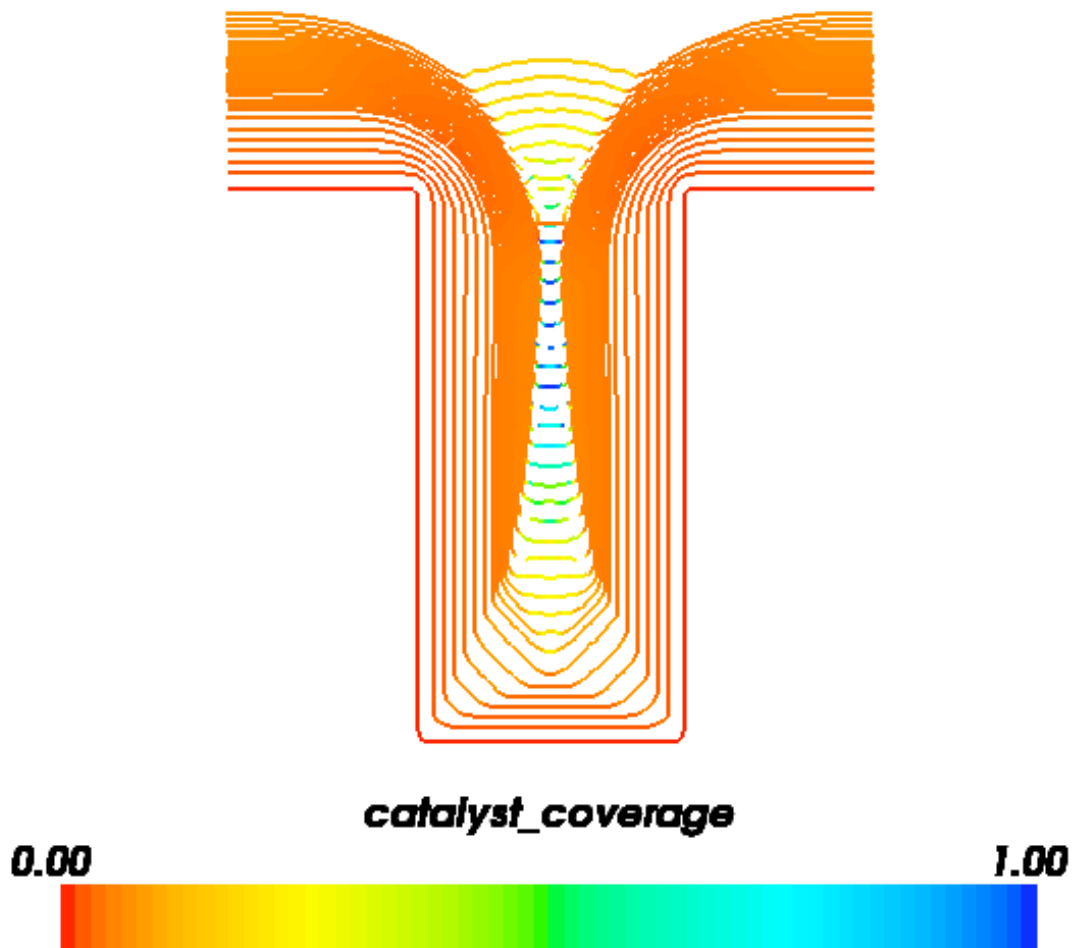
The following image shows a schematic of a trench geometry along with the governing equations for modeling electrodeposition with the CEAC mechanism. All of the given equations are implemented in the `examples.levelSet.electroChem.simpleTrenchSystem.runSimpleTrenchSystem()` function. As stated above, all the parameters in the equations can be changed with function arguments.



The following table shows the symbols used in the governing equations and their corresponding arguments to the `runSimpleTrenchSystem()` function. The boundary layer depth is intentionally small in this example in order not to complicate the mesh. Further examples will simulate more realistic boundary layer depths but will also have more complex meshes requiring the **gmsh** software.

Symbol	Description	Keyword Argument	Value	Unit
	Deposition Rate Parameters			
v	deposition rate			m s^{-1}
i	current density			A m^{-2}
Ω	molar volume	`molarVolume`	7.1×10^{-6}	m^3 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	J K^{-1} 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}	m^2 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}	m^2 s^{-1}
Γ	catalyst site density	`siteDensity`	9.8×10^{-6}	mol m^{-2}
k	rate constant			m^3 mol^{-1} s^{-1}
k_0	rate constant for η^0	`rateConstant0`	1.76	m^3 mol^{-1} s^{-1}
k_3	rate constant for η^3	`rateConstant3`	-245.0×10^{-6}	m^3 mol^{-1} s^{-1} 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.



13.7 examples.levelSet.electroChem.gold

Model electrochemical superfill of gold using the CEAC mechanism.

This input file is a demonstration of the use of *FiPy* for modeling gold superfill. The material properties and experimental parameters used are roughly those that have been previously published [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. To run with a different number of time steps change the `numberOfSteps` argument as follows,

```
>>> runGold(numberOfSteps=10, displayViewers=False)
1
```

Change the `displayViewers` argument to `True` if you wish to see the results displayed on the screen. This example has a more realistic default boundary layer depth and thus requires *gmsh* to construct a more complex mesh.

There are a few differences between the gold superfill model presented in this example and in `examples.levelSet.electroChem.simpleTrenchSystem`. Most default values have changed to account

for a different metal ion (gold) and catalyst (lead). In this system the catalyst is not present in the electrolyte but instead has a non-zero initial coverage. Thus quantities associated with bulk catalyst and catalyst accumulation are not defined. The current density is given by,

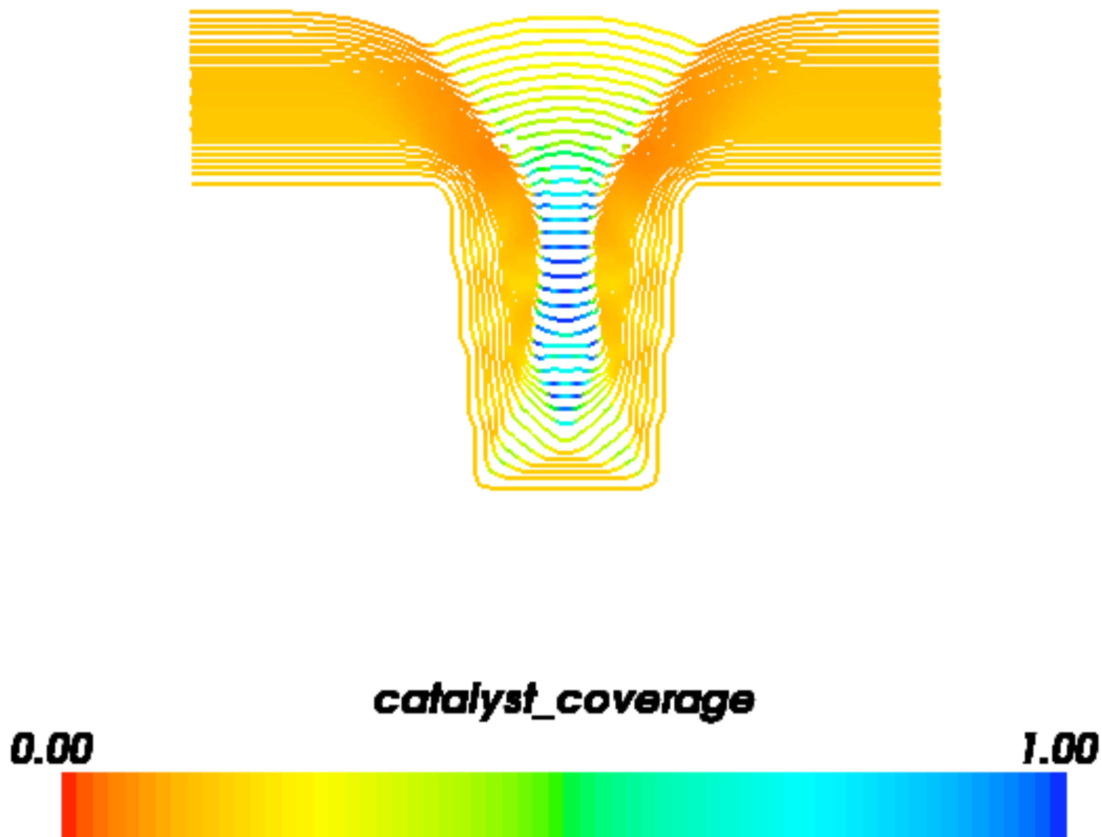
$$i = \frac{c_m}{c_m^\infty} \left(b_0 + b_1 \theta \right).$$

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.



13.8 examples.levelSet.electroChem.leveler

Model electrochemical superfill of copper with leveler and accelerator additives.

This input file is a demonstration of the use of *FiPy* for modeling copper superfill with leveler and accelerator additives. The material properties and experimental parameters used are roughly those that have been previously published [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. To run with a different number of time steps change the `numberOfSteps` argument as follows,

```
>>> runLeveler(numberOfSteps=10, displayViewers=False, cellSize=0.25e-7)
1
```

Change the `displayViewers` argument to `True` if you wish to see the results displayed on the screen. This example requires *gmsh* to construct the mesh.

This example models the case when suppressor, accelerator and leveler additives are present in the electrolyte. The suppressor is assumed to 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 j represents S for suppressor, A for accelerator, L for leveler and V for vacant. This model assumes a linear interpolation between the three cases of complete coverage for each additive or vacant substrate. The governing equations for the surfactants are given by,

$$\begin{aligned}
\dot{\theta_L} &= \kappa v \theta_L + k_l^+ c_L (1 - \theta_L) - k_L^- v \theta_L, \\
\dot{\theta_a} &= \kappa v \theta_A + k_A^+ c_A (1 - \theta_A - \theta_L) - k_L c_L \theta_A - k_A^- \theta_A^{q-1}, \\
\theta_S &= 1 - \theta_A - \theta_L \\
\theta_V &= 0.
\end{aligned}$$

It has been found experimentally that $i_L = i_S$.

If the surface reaches full coverage, the equations do not naturally prevent the coverage rising above full coverage due to the curvature terms. Thus, when $\theta_L + \theta_A = 1$ then the equation for accelerator becomes $\dot{\theta_A} = -\dot{\theta_L}$ and when $\theta_L = 1$, the equation for leveler becomes $\dot{\theta_L} = -k_L^- v \theta_L$.

The parameters k_A^+, k_A^- and q are both functions of η given by,

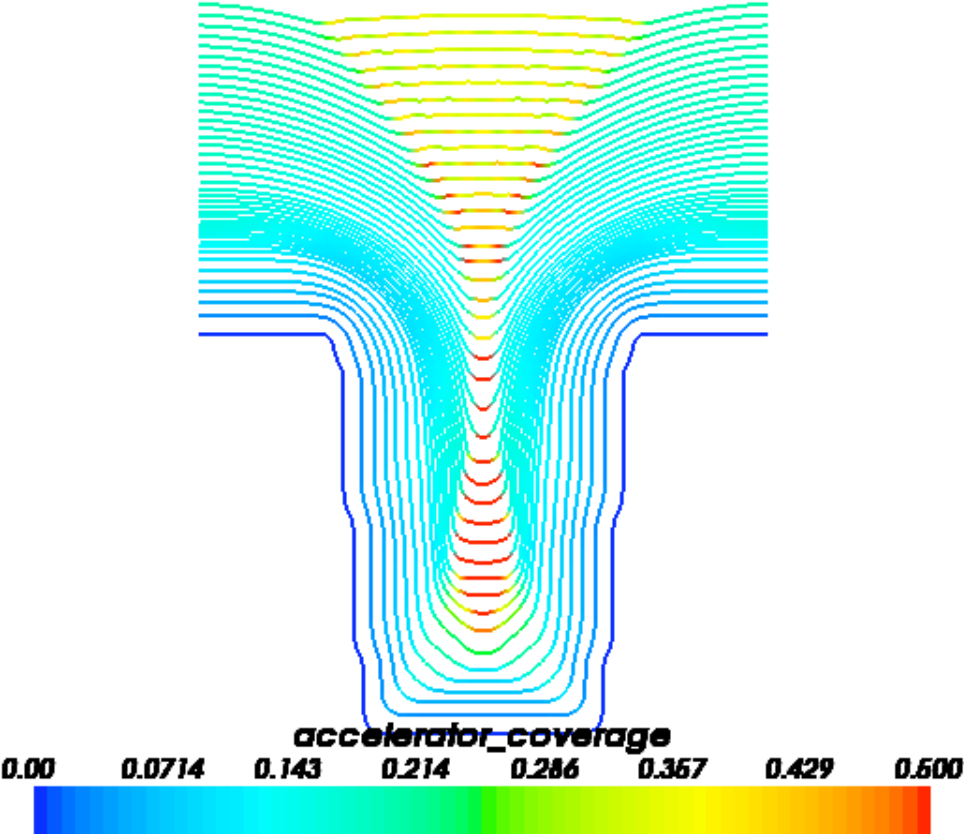
$$\begin{aligned}
k_A^+ &= k_{A0}^+ \exp \frac{-\alpha_k F \eta}{RT}, \\
k_A^- &= B_d + \frac{A}{\exp (B_a (\eta + V_d))} + \exp (B_b (\eta + V_d)) \\
q &= m \eta + b.
\end{aligned}$$

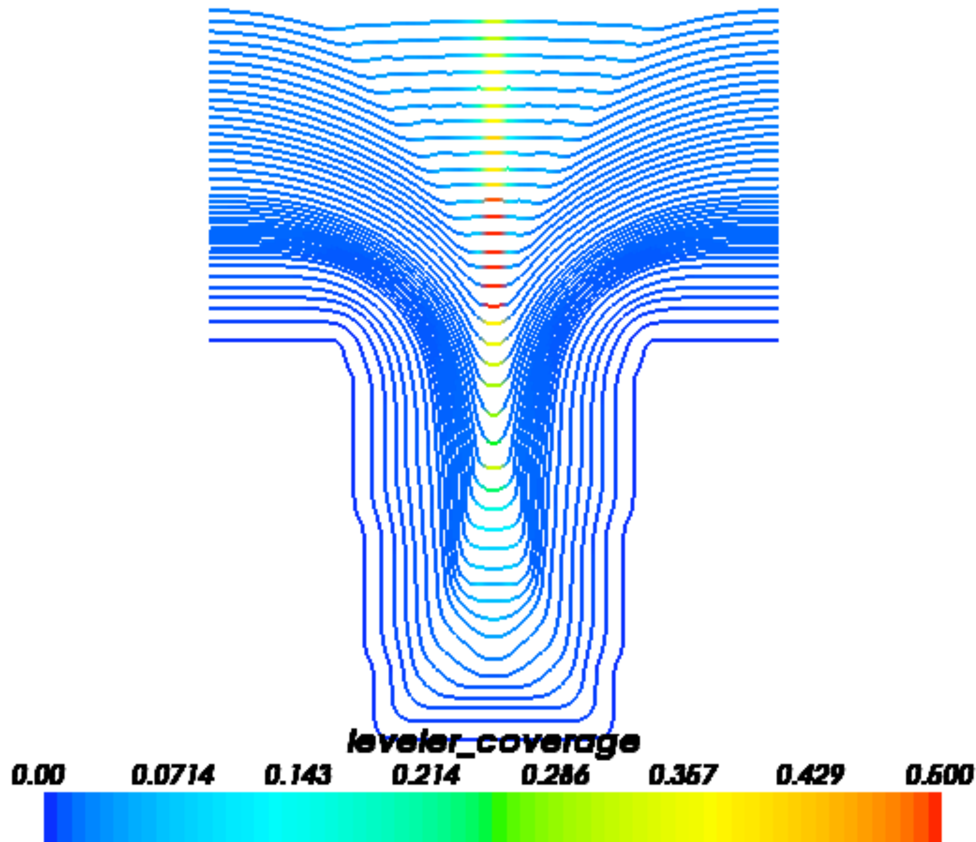
The following table shows the symbols used in the governing equations and their corresponding arguments for the

`runLeveler()` function.

Symbol	Description	Keyword Argument	Value	Unit
	Deposition Rate Parameters			
v	deposition rate			m s^{-1}
i_A	accelerator current density	`i0Accelerator`		A m^{-2}
i_L	leveler current density	`i0Leveler`		A m^{-2}
Ω	molar volume	`molarVolume`	7.1×10^{-6}	m^3 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_A	accelerator transfer coefficient	`alphaAccelerator`	0.4	
α_S	leveler transfer coefficient	`alphaLeveler`	0.5	
η	overpotential	`overpotential`	-0.3	V
R	gas constant	`gasConstant`	8.314	J K mol^{-1}
T	temperature	`temperature`	298.0	K
	Ion Parameters			
c_I	ion concentration	`ionConcentration`	250.0	mol m^{-3}
c_I^∞	far field ion concentration	`ionConcentration`	250.0	mol m^{-3}
D_I	ion diffusion coefficient	`ionDiffusion`	5.6×10^{-10}	m^2 s^{-1}
	Accelerator Parameters			
θ_A	accelerator coverage	`acceleratorCoverage`	0.0	
c_A	accelerator concentartion	`acceleratorConcentration`	5.0×10^{-3}	mol m^{-3}
c_A^∞	far field accelerator concentration	`acceleratorConcentration`	5.0×10^{-3}	mol m^{-3}
D_A	catalyst diffusion coefficient	`catalystDiffusion`	1.0×10^{-9}	m^2 s^{-1}
Γ_A	accelerator site density	`siteDensity`	9.8×10^{-6}	mol m^{-2}
k_A^+	accelerator adsorption			m^3 mol^{-1} s^{-1}
k_{A0}^+	accelerator adsorption coeff	`kAccelerator0`	2.6×10^{-4}	m^3 mol^{-1} s^{-1}
α_k	accelerator adsorption coeff	`alphaAdsorption`	0.62	
k_A^-	accelerator consumption coeff			
B_a	experimental parameter	`Bd`	-40.0	
B_b	experimental parameter	`Bd`	60.0	
V_d	experimental parameter	`Bd`	9.8×10^{-2}	
B_d	experimental parameter	`Bd`	8.0×10^{-4}	
	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`	

The following images show accelerator and leveler contour plots that can be obtained by running this example.


accelerator_coverage
0.00
0.0714
0.143
0.214
0.286
0.357
0.429
0.500



13.9 examples.levelSet.electroChem.howToWriteAScript

Tutorial for writing an electrochemical superfill script.

This input file demonstrates how to create a new superfill script if the existing suite of scripts do not meet the required needs. It provides the functionality of `examples.levelSet.electroChem.simpleTrenchSystem`.

To run this example from the base fipy directory type:

```
$ python examples/levelSet/electroChem/howToWriteAScript.py --numberOfElements=10000 --numberOfSteps=
```

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
>>> numberOfCellsInNarrowBand = 10
>>> cellsBelowTrench = 10
>>> cellSize = 0.1e-7
```

- parameters required for a trench geometry,

```
>>> trenchDepth = 0.5e-6
>>> aspectRatio = 2.
>>> trenchSpacing = 0.6e-6
>>> boundaryLayerDepth = 0.3e-6
```

The hydrodynamic boundary layer depth (`boundaryLayerDepth`) is intentionally small in this example to keep the mesh at a reasonable size.

Build the mesh:

```
>>> from fipy.tools.parser import parse
>>> numberOfElements = parse('--numberOfElements', action='store',
...     type='int', default=-1)
>>> numberOfSteps = parse('--numberOfSteps', action='store',
...     type='int', default=2)

>>> 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,
...               communicator=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.cellCenters
>>> 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.interfaceVar
>>> exchangeCurrentDensity = currentDensity0 + tmp
>>> expo = numerix.exp(expoConstant * overpotential)
>>> currentDensity = expo * exchangeCurrentDensity * metalVar \
...                  / bulkMetalConcentration
>>> depositionRateVariable = currentDensity * molarVolume \
...                          / (charge * faradaysConstant)
```

Build the extension velocity variable v_{ext}. The extension velocity uses the `extensionEquation` to spread the velocity at the interface to the rest of the domain.

```
>>> extensionVelocityVariable = CellVariable(
...     name='extension velocity',
...     mesh=mesh,
...     value=depositionRateVariable)
```

Using the variables created above the governing equations will be built. The governing equation for surfactant conservation is given by,

$$\dot{\theta} = Jv\theta + kc_\theta^i(1-\theta)$$

where θ is the coverage of catalyst at the interface, J is the curvature of the interface, v is the normal velocity of the interface, c_θ^i is the concentration of catalyst in the bulk at the interface. The value k is given by an empirical function of overpotential,

$$k = k_0 + k_3\eta^3$$

The above equation is represented by the `AdsorbingSurfactantEquation` in *FiPy*:

```
>>> surfactantEquation = AdsorbingSurfactantEquation(
...     surfactantVar=catalystVar,
...     distanceVar=distanceVar,
...     bulkVar=bulkCatalystVar,
...     rateConstant=rateConstant0 \
...                  + rateConstant3 * overpotential**3)
```

The variable ϕ is advected by the `advectionEquation` given by,

$$\frac{\partial \phi}{\partial t} + v_{\text{ext}} |\nabla \phi| = 0$$

and is set up with the following commands:

```
>>> advectionEquation = 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 metal ion diffusion equation is set up with the following commands.

```
>>> metalEquation = buildMetalIonDiffusionEquation(
...     ionVar=metalVar,
...     distanceVar=distanceVar,
...     depositionRate=depositionRateVariable,
...     diffusionCoeff=metalDiffusionCoefficient,
...     metalIonMolarVolume=molarVolume,
... )

>>> metalVar.constrain(bulkMetalConcentration, mesh.facesTop)
```

The surfactant bulk diffusion equation solves the bulk diffusion of a species with a source term for the jump from the bulk to an interface. The governing equation is given by,

$$\frac{\partial c}{\partial t} = \nabla \cdot D \nabla c$$

where,

$$D = \begin{cases} D_\theta & \text{when } \phi > 0 \\ 0 & \text{when } \phi \leq 0 \end{cases}$$

The jump condition at the interface is defined by Langmuir adsorption. Langmuir adsorption essentially states that the ability for a species to jump from an electrolyte to an interface is proportional to the concentration in the electrolyte, available site density and a jump coefficient. The boundary condition at $\phi = 0$ is given by,

$$D \hat{n} \cdot \nabla c = -kc(1 - \theta).$$

The surfactant bulk diffusion equation is set up with the following commands.

```
>>> bulkCatalystEquation = buildSurfactantBulkDiffusionEquation(
...     bulkVar=bulkCatalystVar,
...     distanceVar=distanceVar,
...     surfactantVar=catalystVar,
...     diffusionCoeff=catalystDiffusion,
...     rateConstant=rateConstant0 * siteDensity
... )
```

```
>>> bulkCatalystVar.constrain(catalystConcentration, mesh.facesTop)
```

If running interactively, create viewers.

```
>>> if __name__ == '__main__':
...     try:
...         viewer = MayaviSurfactantViewer(distanceVar,
...                                         catalystVar.interfaceVar,
...                                         zoomFactor=1e6,
...                                         datamax=1.0,
...                                         datamin=0.0,
...                                         smooth=1)
...     except:
...         viewer = MultiViewer(viewers=(
...             Viewer(distanceVar, datamin=-1e-9, datamax=1e-9),
...             Viewer(catalystVar.interfaceVar)))
... else:
...     viewer = None
```

The `levelSetUpdateFrequency` defines how often to call the `distanceEquation` to reinitialize the `distanceVariable` to a distance function.

```
>>> levelSetUpdateFrequency = int(0.8 * narrowBandWidth \
...                               / (cellSize * cflNumber * 2))
```

The following loop runs for `numberOfSteps` time steps. The time step is calculated with the CFL number and the maximum extension velocity. v to v_ext throughout the whole domain using $\nabla\phi\cdot\nabla v_\text{ext} = 0$.

```
>>> for step in range(numberOfSteps):
...
...     if viewer is not None:
...         viewer.plot()
...
...     if step % levelSetUpdateFrequency == 0:
...         distanceVar.calcDistanceFunction()
...
...     extensionVelocityVariable.setValue(depositionRateVariable())
...
...     distanceVar.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)
...     bulkCatalystEquation.solve(var=bulkCatalystVar, dt=dt, solver=GeneralSolver())
```

The following is a short test case. It uses saved data from a simulation with 5 time steps. It is not a test for accuracy but a way to tell if something has changed or been broken.

```
>>> import os
>>> filepath = os.path.join(os.path.split(__file__)[0],
```

```
...                         "simpleTrenchSystem.gz")

>>> print catalystVar.allclose(numerix.loadtxt(filepath), rtol=1e-4)
1

>>> if __name__ == '__main__':
...     raw_input('finished')
```

Chapter 14

Cahn Hilliard Examples

examples.cahnHilliard.mesh2DCoupled	Solve the Cahn-Hilliard problem in two dimensions.
examples.cahnHilliard.sphere	Solves the Cahn-Hilliard problem on the surface of a sphere.

14.1 examples.cahnHilliard.mesh2DCoupled

Solve the Cahn-Hilliard problem in two dimensions.

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 = 20
... else:
...     nx = ny = 10
>>> mesh = Grid2D(nx=nx, ny=ny, dx=0.25, dy=0.25)
>>> phi = CellVariable(name=r"$\phi$", mesh=mesh)
>>> psi = CellVariable(name=r"$\psi$", mesh=mesh)
```

We start the problem with random fluctuations about $\phi = 1/2$

```
>>> noise = GaussianNoiseVariable(mesh=mesh,
...                               mean=0.5,
...                               variance=0.01).value
```

```
>>> phi[:] = noise
```

FiPy doesn't plot or output anything unless you tell it to:

```
>>> if __name__ == "__main__":
...     viewer = Viewer(vars=(phi, psi)) # , datamin=0., datamax=1.)
```

We factor the Cahn-Hilliard equation into two 2nd-order PDEs and place them in canonical form for *FiPy* to solve them as a coupled set of equations.

$$\frac{\partial \phi}{\partial t} = \nabla \cdot D \nabla \psi$$
$$\psi = \frac{\partial^2 f}{\partial \phi^2}(\phi - \phi^{\text{old}}) + \frac{\partial f}{\partial \phi} - \epsilon^2 \nabla^2 \phi$$

We need to perform the partial derivatives

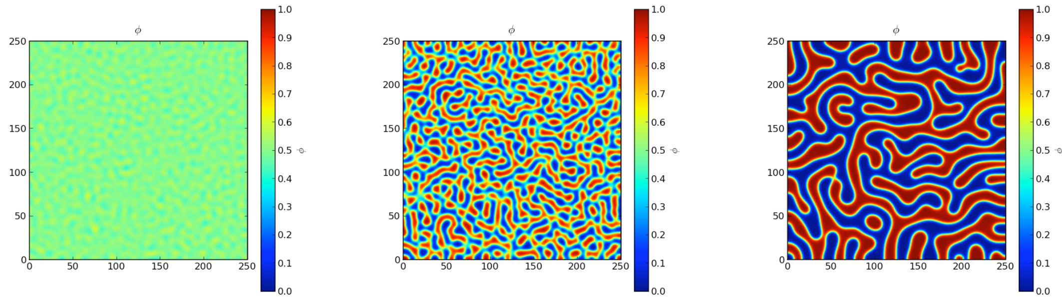
$$\frac{\partial f}{\partial \phi} = (a^2/2) 2\phi(1-\phi)(1-2\phi)$$
$$\frac{\partial^2 f}{\partial \phi^2} = (a^2/2) 2 \left[1 - 6\phi(1-\phi)\right]$$

manually.

```
>>> D = a = epsilon = 1.
>>> dfdphi = a**2 * 2 * phi * (1 - phi) * (1 - 2 * phi)
>>> dfdphi_ = a**2 * 2 * (1 - phi) * (1 - 2 * phi)
>>> d2fdphi2 = a**2 * 2 * (1 - 6 * phi * (1 - phi))
>>> eq1 = (TransientTerm(var=phi) == DiffusionTerm(coeff=D, var=psi))
>>> eq2 = (ImplicitSourceTerm(coeff=1., var=psi)
...        == ImplicitSourceTerm(coeff=-d2fdphi2, var=phi) - d2fdphi2 * phi + dfdphi
...        - DiffusionTerm(coeff=epsilon**2, var=phi))
>>> eq3 = (ImplicitSourceTerm(coeff=1., var=psi)
...        == ImplicitSourceTerm(coeff=dfdphi_, var=phi)
...        - DiffusionTerm(coeff=epsilon**2, var=phi))

>>> eq = eq1 & eq2
```

Because the evolution of a spinodal microstructure slows with time, we use exponentially increasing time steps to keep the simulation "interesting". The *FiPy* user always has direct control over the evolution of their problem.

```
>>> dexp = -5
>>> elapsed = 0.
>>> if __name__ == "__main__":
...     duration = .5e-1
... else:
...     duration = .5e-1

>>> while elapsed < duration:
...     dt = min(100, numerix.exp(dexp))
...     elapsed += dt
...     dexp += 0.01
...     eq.solve(dt=dt)
...     if __name__ == "__main__":
...         viewer.plot()

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



(a) $t = 30$ (b) $t = 100$ (c) $t = 1000$

These equations can also be solved in *FiPy* using a vector equation. The variables ϕ and ψ are now stored in a single variable

```
>>> var = CellVariable(mesh=mesh, elementshape=(2,))
>>> var[0] = noise

>>> if __name__ == "__main__":
...     viewer = Viewer(vars=(var[0], var[1]))

>>> D = a = epsilon = 1.
>>> v0 = var[0]
>>> dfdphi = a**2 * 2 * v0 * (1 - v0) * (1 - 2 * v0)
>>> dfdphi_ = a**2 * 2 * (1 - v0) * (1 - 2 * v0)
>>> d2fdphi2 = a**2 * 2 * (1 - 6 * v0 * (1 - v0))
```

The source terms have to be shaped correctly for a vector. The implicit source coefficient has to have a shape of *(2, 2)* while the explicit source has a shape *(2,)*

```
>>> source = (- d2fdphi2 * v0 + dfdphi) * (0, 1)
>>> impCoeff = -d2fdphi2 * ((0, 0),
...                          (1., 0)) + ((0, 0),
...                                      (0, -1.))
```

This is the same equation as the previous definition of *eq*, but now in a vector format.

```
>>> eq = TransientTerm(((1., 0.),
...                     (0., 0.))) == DiffusionTerm([((0.,          D),
...                                                   (-epsilon**2, 0.))]) + ImplicitSourceTerm(impCo

>>> dexp = -5
>>> elapsed = 0.

>>> while elapsed < duration:
...     dt = min(100, numerix.exp(dexp))
...     elapsed += dt
...     dexp += 0.01
...     eq.solve(var=var, dt=dt)
...     if __name__ == "__main__":
...         viewer.plot()

>>> print numerix.allclose(var, (phi, psi))
True
```

14.2 examples.cahnHilliard.sphere

Solves the Cahn-Hilliard problem on the surface of a sphere.

This phenomenon canoccur 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 = Gmsh2DIn3DSpace('''
...     radius = 5.0;
...     cellSize = 0.3;
...
...     // create inner 1/8 shell
...     Point(1) = {0, 0, 0, cellSize};
...     Point(2) = {-radius, 0, 0, cellSize};
...     Point(3) = {0, radius, 0, cellSize};
...     Point(4) = {0, 0, radius, cellSize};
...     Circle(1) = {2, 1, 3};
...     Circle(2) = {4, 1, 2};
...     Circle(3) = {4, 1, 3};
...     Line Loop(1) = {1, -3, 2} ;
...     Ruled Surface(1) = {1};
...
...     // create remaining 7/8 inner shells
...     t1[] = Rotate {{0,0,1},{0,0,0},Pi/2} {Duplicata{Surface{1};}};
...     t2[] = Rotate {{0,0,1},{0,0,0},Pi} {Duplicata{Surface{1};}};
...     t3[] = Rotate {{0,0,1},{0,0,0},Pi*3/2} {Duplicata{Surface{1};}};
...     t4[] = Rotate {{0,1,0},{0,0,0},-Pi/2} {Duplicata{Surface{1};}};
...     t5[] = Rotate {{0,0,1},{0,0,0},Pi/2} {Duplicata{Surface{t4[0]};}};
...     t6[] = Rotate {{0,0,1},{0,0,0},Pi} {Duplicata{Surface{t4[0]};}};
...     t7[] = Rotate {{0,0,1},{0,0,0},Pi*3/2} {Duplicata{Surface{t4[0]};}};
...
...     // create entire inner and outer shell
...     Surface Loop(100)={1,t1[0],t2[0],t3[0],t7[0],t4[0],t5[0],t6[0]};
... ''', order=2).extrude(extrudeFunc=lambda r: 1.1 * r)
>>> phi = CellVariable(name=r"$\phi$", mesh=mesh)
```

We start the problem with random fluctuations about $\phi = 1/2$

```
>>> phi.setValue(GaussianNoiseVariable(mesh=mesh,
...                                    mean=0.5,
...                                    variance=0.01))
```

FiPy doesn't plot or output anything unless you tell it to: If `MayaviClient` is available, we can customize the view with a sublcass 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:

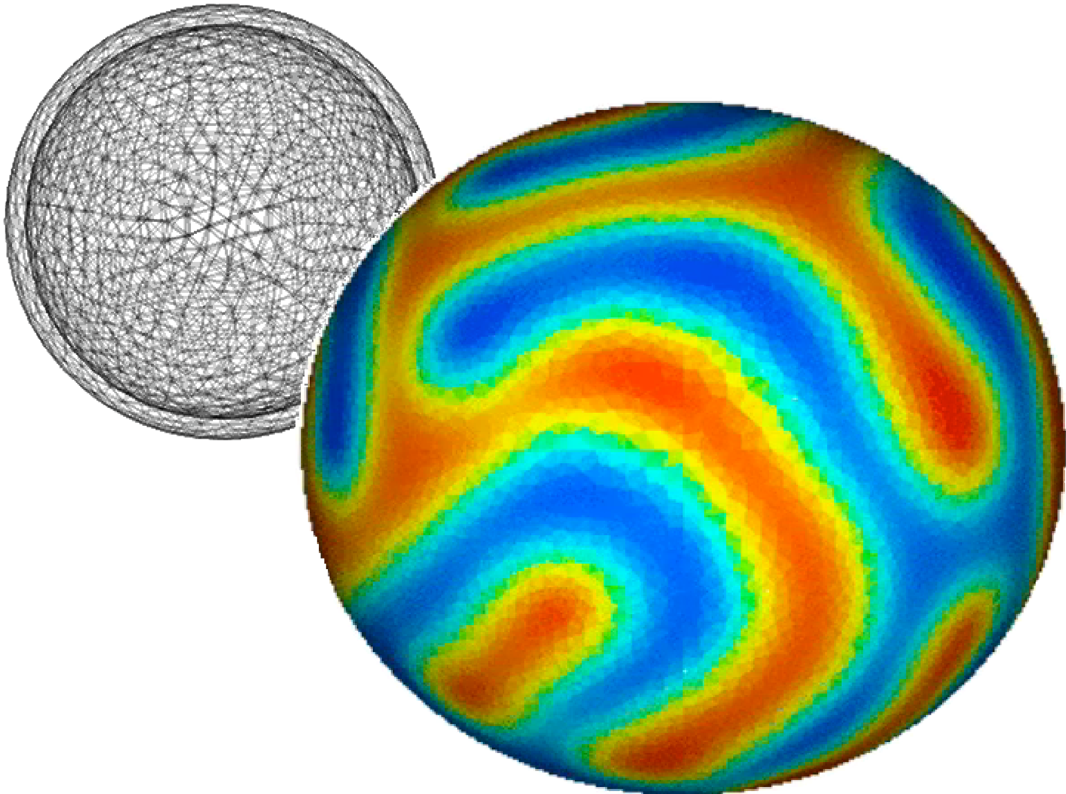
$$\frac{\partial \phi}{\partial t} = \nabla \cdot D a^2 \left[1 - 6\phi\left(1 - \phi\right)\right] \nabla \phi - \nabla \cdot D \nabla \epsilon^2 \nabla^2 \phi.$$

FiPy would automatically interpolate `D * a**2 * (1 - 6 * phi * (1 - phi))` onto the faces, where the diffusive flux is calculated, but we obtain somewhat more accurate results by performing a linear interpolation from `phi` at cell centers to `PHI` at face centers. Some problems benefit from non-linear interpolations, such as harmonic or geometric means, and *FiPy* makes it easy to obtain these, too.

```
>>> PHI = phi.arithmeticFaceValue
>>> D = a = epsilon = 1.
>>> eq = (TransientTerm()
...       == DiffusionTerm(coeff=D * a**2 * (1 - 6 * PHI * (1 - PHI)))
...       - DiffusionTerm(coeff=(D, epsilon**2)))
```

Because the evolution of a spinodal microstructure slows with time, we use exponentially increasing time steps to keep the simulation “interesting”. The *FiPy* user always has direct control over the evolution of their problem.

```
>>> dexp = -5
>>> elapsed = 0.
>>> if __name__ == "__main__":
...     duration = 1000.
... else:
...     duration = 1e-2
>>> while elapsed < duration:
...     dt = min(100, numerix.exp(dexp))
...     elapsed += dt
...     dexp += 0.01
...     eq.solve(phi, dt=dt, solver=DefaultSolver(precon=None))
...     if __name__ == "__main__":
...         viewer.plot()
```



Chapter 15

Fluid Flow Examples

examples.flow.stokesCavity	Solve the Navier-Stokes equation in the viscous limit.

15.1 examples.flow.stokesCavity

Solve the Navier-Stokes equation in the viscous limit.

Many thanks to Benny Malengier <bm@cage.ugent.be> for reworking this example and actually making it work correctly...see changeset:3799

This example is an implementation of a rudimentary Stokes solver on a collocated grid. It solves the Navier-Stokes equation in the viscous limit,

$$\nabla \mu \cdot \nabla \vec{u} = \nabla p$$

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 for collocated grids to solve the pressure-momentum coupling. Some of the details of the algorithm will be highlighted below but a good reference for this material is Ferziger and Peric [ferziger] and Rossow [rossow:2003]. The solution has a high degree of error close to the corners of the domain for the pressure but does a reasonable job of predicting the velocities away from the boundaries. A number of aspects of *FiPy* need to be improved to have a first class flow solver. These include, higher order spatial diffusion terms, proper wall boundary conditions, improved mass flux evaluation and extrapolation of cell values to the boundaries using gradients.

In the table below a comparison is made with the Dolfyn open source code on a 100 by 100 grid. The table shows the frequency of values that fall within the given error confidence bands. Dolfyn has the added features described above. When these features are switched off the results of Dolfyn and *FiPy* are identical.

% frequency of cells	x-velocity error (%)	y-velocity error (%)	pressure error (%)
90	< 0.1	< 0.1	< 5
5	0.1 to 0.6	0.1 to 0.3	5 to 11
4	0.6 to 7	0.3 to 4	11 to 35
1	7 to 96	4 to 80	35 to 179
0	> 96	> 80	> 179

To start, some parameters are declared.

```
>>> from fipy import *
>>> #from fipy.meshes.grid2D import Grid2D
```

```
>>> L = 1.0
>>> N = 50
>>> dL = L / N
>>> viscosity = 1
>>> U = 1.
>>> #0.8 for pressure and 0.5 for velocity are typical relaxation values for SIMPLE
>>> pressureRelaxation = 0.8
>>> velocityRelaxation = 0.5
>>> if __name__ == '__main__':
...     sweeps = 300
... else:
...     sweeps = 5
```

Build the mesh.

```
>>> mesh = Grid2D(nx=N, ny=N, dx=dL, dy=dL)
```

Declare the variables.

```
>>> pressure = CellVariable(mesh=mesh, name='pressure')
>>> pressureCorrection = CellVariable(mesh=mesh)
>>> xVelocity = CellVariable(mesh=mesh, name='X velocity')
>>> yVelocity = CellVariable(mesh=mesh, name='Y velocity')
```

The velocity is required as a rank-1 `FaceVariable` for calculating the mass flux. This is required by the Rhie-Chow correction to avoid pressure/velocity decoupling.

```
>>> velocity = FaceVariable(mesh=mesh, rank=1)
```

Build the Stokes equations in the cell centers.

```
>>> xVelocityEq = DiffusionTerm(coeff=viscosity) - pressure.grad.dot([1.,0.])
>>> yVelocityEq = DiffusionTerm(coeff=viscosity) - pressure.grad.dot([0.,1.])
```

In this example the SIMPLE algorithm is used to couple the pressure and momentum equations. Let us assume we have solved the discretized momentum equations using a guessed pressure field p^* to obtain a velocity field $\vec{u}^*$. That is $\vec{u}^*$ is found from

$$a_P \vec{u}_P^* = \sum_f a_A \vec{u}_A^* - V_P (\nabla p^*)_P$$

We would like to somehow correct these initial fields to satisfy both the discretized momentum and continuity equations. We now try to correct these initial fields with a correction such that $\vec{u} = \vec{u}^* + \vec{u}'$ and $p = p^* + p'$, where $\vec{u}$ and p now satisfy the momentum and continuity equations. Substituting the exact solution into the equations we obtain,

$$\nabla \mu \cdot \nabla \vec{u}' = \vec{p}'$$

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 results in an equation for $p = p'$,

$$a_P \vec{u}_P' = \sum_f a_A \vec{u}_A' - V_P (\nabla p')_P$$

where notation from *Linear Equations* is used. The SIMPLE algorithm drops the second term in the above equation to leave,

$$\vec{u}_P' = -\frac{V_P (\nabla p')_P}{a_P}$$

By substituting the above expression into the continuity equations we obtain the pressure correction equation,

$$\nabla \frac{V_P}{a_P} \cdot \nabla p' = \nabla \cdot \vec{u}^*$$

In the discretized version of the above equation V_P/a_P is approximated at the face by $A_f d_{AP}/(a_P)_f$. In *FiPy* the pressure correction equation can be written as,

```
>>> ap = CellVariable(mesh=mesh, value=1.)
>>> coeff = 1./ ap.arithmeticFaceValue*mesh._faceAreas * mesh._cellDistances
>>> pressureCorrectionEq = DiffusionTerm(coeff=coeff) - velocity.divergence
```

Above would work good on a staggered grid, however, on a colocated grid as *FiPy* uses, the term `velocity.divergence` will cause oscillations in the pressure solution as velocity is a face variable. We can apply the Rhie-Chow correction terms for this. In this an intermediate velocity term $u^\diamond$ is considered which does not contain the pressure corrections:

$$\vec{u}_P^\diamond = \vec{u}_P^* + \frac{V_P}{a_P}(\nabla p^*)_P = \sum_f \frac{a_A}{a_P}\vec{u}_A^*$$

This velocity is interpolated at the edges, after which the pressure correction term is added again, but now considered at the edge:

$$\vec{u}_f = \frac{1}{2}(\vec{u}_L^\diamond + \vec{u}_R^\diamond)) - \left(\frac{V}{a_P}\right)_{\text{avg L,R}} (\nabla p_f^*)$$

where $\left(\frac{V}{a_P}\right)_{\text{avg L,R}}$ is assumed a good approximation at the edge. Here L and R denote the two cells adjacent to the face. Expanding the not calculated terms we arrive at

$$\vec{u}_f = \frac{1}{2}(\vec{u}_L^* + \vec{u}_R^*)) + \frac{1}{2}\left(\frac{V}{a_P}\right)_{\text{avg L,R}} (\nabla p_L^* + \nabla p_R^*) - \left(\frac{V}{a_P}\right)_{\text{avg L,R}} (\nabla p_f^*)$$

where we have replaced the coefficients of the cell pressure gradients by an averaged value over the edge. This formula has the consequence that the velocity on a face depends not only on the pressure of the adjacent cells, but also on the cells further away, which removes the unphysical pressure oscillations. We start by introducing needed terms

```
>>> from fipy.variables.faceGradVariable import _FaceGradVariable
>>> volume = CellVariable(mesh=mesh, value=mesh.cellVolumes, name='Volume')
>>> contrvolume=volume.arithmeticFaceValue
```

And set up the velocity with this formula in the SIMPLE loop. Now, set up the no-slip boundary conditions

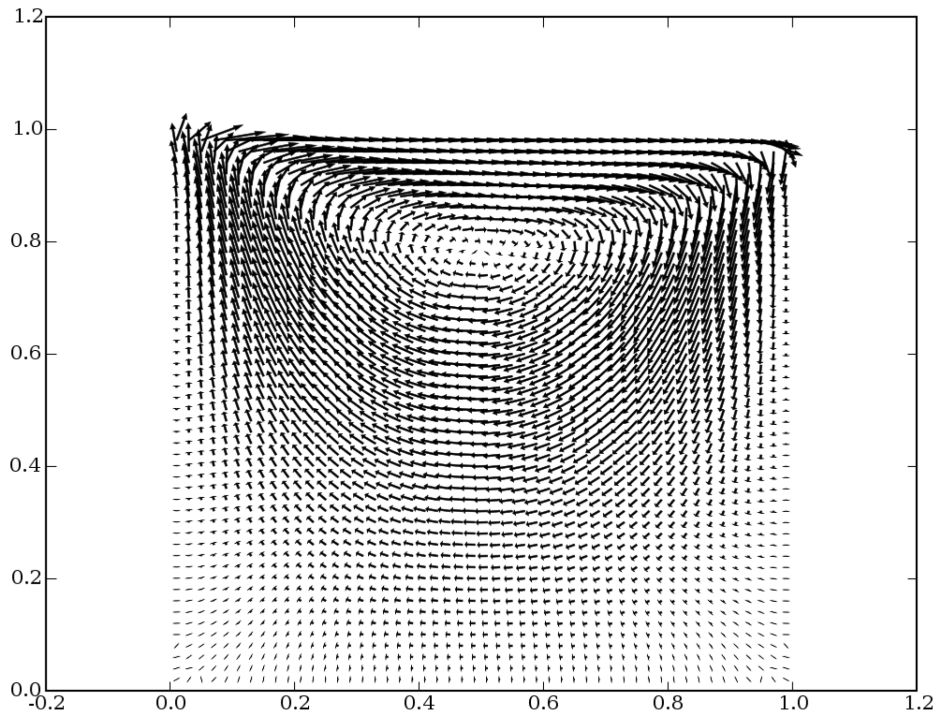
```
>>> xVelocity.constrain(0., mesh.facesRight | mesh.facesLeft | mesh.facesBottom)
>>> xVelocity.constrain(U, mesh.facesTop)
>>> yVelocity.constrain(0., mesh.exteriorFaces)
>>> X, Y = mesh.faceCenters
>>> pressureCorrection.constrain(0., mesh.facesLeft & (Y < dL))
```

Set up the viewers,

```
>>> if __name__ == '__main__':
...     viewer = Viewer(vars=(pressure, xVelocity, yVelocity, velocity),
...                     xmin=0., xmax=1., ymin=0., ymax=1., colorbar=True)
```

Below, we iterate for a set number of sweeps. We use the `sweep()` method instead of `solve()` because we require the residual for output. We also use the `cacheMatrix()`, `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,
...                              underRelaxation=velocityRelaxation)
...     xmat = xVelocityEq.matrix
...
...     yres = yVelocityEq.sweep(var=yVelocity,
...                              underRelaxation=velocityRelaxation)
...
...     ## update the ap coefficient from the matrix diagonal
...     ap[:] = -xmat.takeDiagonal()
...
...     ## update the face velocities based on starred values with the
...     ## Rhie-Chow correction.
...     ## cell pressure gradient
...     presgrad = pressure.grad
...     ## face pressure gradient
...     facepresgrad = _FaceGradVariable(pressure)
...
...     velocity[0] = xVelocity.arithmeticFaceValue \
...          + contrvolume / ap.arithmeticFaceValue * \
...            (presgrad[0].arithmeticFaceValue-facepresgrad[0])
...     velocity[1] = yVelocity.arithmeticFaceValue \
...          + contrvolume / ap.arithmeticFaceValue * \
...            (presgrad[1].arithmeticFaceValue-facepresgrad[1])
...     velocity[..., mesh.exteriorFaces.value] = 0.
...     velocity[0, mesh.facesTop.value] = U
...
...     ## solve the pressure correction equation
...     pressureCorrectionEq.cacheRHSvector()
...     ## left bottom point must remain at pressure 0, so no correction
...     pres = pressureCorrectionEq.sweep(var=pressureCorrection)
...     rhs = pressureCorrectionEq.RHSvector
...
...     ## update the pressure using the corrected value
...     pressure.setValue(pressure + pressureRelaxation * pressureCorrection )
...     ## update the velocity using the corrected pressure
...     xVelocity.setValue(xVelocity - pressureCorrection.grad[0] / \
...                                                 ap * mesh.cellVolumes)
...     yVelocity.setValue(yVelocity - pressureCorrection.grad[1] / \
...                                                 ap * mesh.cellVolumes)
...
...     if __name__ == '__main__':
...         if sweep%10 == 0:
...             print 'sweep:',sweep,', x residual:',xres, \
...                                  ', y residual',yres, \
...                                  ', p residual:',pres, \
...                                  ', continuity:',max(abs(rhs))
...
...             viewer.plot()
```



Test values in the last cell.

```
>>> print numerix.allclose(pressure.globalValue[...,-1], 162.790867927)
1
>>> print numerix.allclose(xVelocity.globalValue[...,-1], 0.265072740929)
1
>>> print numerix.allclose(yVelocity.globalValue[...,-1], -0.150290488304)
1
```

Chapter 16

Reactive Wetting Examples

`examples.reactiveWetting.liquidVapor1D`	Solve a single-component, liquid-vapor, van der Waals system.

16.1 examples.reactiveWetting.liquidVapor1D

Solve a single-component, liquid-vapor, van der Waals system.

This example solves a single-component, liquid-vapor, van der Waals system as described by Wheeler et *al.* [PhysRevE.82.051601]. The free energy for this system takes the form,

$$f = -\frac{e\rho^2}{m^2} + \frac{RT}{m}\left(\ln\frac{\rho}{m - \bar{v}\rho}\right) \tag{16.1}$$

where ρ is the density. This free energy supports a two phase equilibrium with densities given by ρ^l and ρ^v in the liquid and vapor phases, respectively. The densities are determined by solving the following system of equations,

$$P\left(\rho^l\right) = P\left(\rho^v\right) \tag{16.2}$$

and

$$\mu\left(\rho^l\right) = \mu\left(\rho^v\right) \tag{16.3}$$

where μ is the chemical potential,

$$\mu = \frac{\partial f}{\partial \rho} \tag{16.4}$$

and P is the pressure,

$$P = \rho\mu - f \tag{16.5}$$

One choice of thermodynamic parameters that yields a relatively physical two phase system is

```
>>> molarWeight = 0.118
>>> ee = -0.455971
>>> gasConstant = 8.314
>>> temperature = 650.
>>> vbar = 1.3e-05
```

with equilibrium density values of

```
>>> liquidDensity = 7354.3402662299995
>>> vaporDensity = 82.855803327810008
```

The equilibrium densities are verified by substitution into Eqs. (16.2) and (16.3). Firstly, Eqs. (16.1), (16.4) and (16.5) are defined as python functions,

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

```
>>> def f(rho):
...     return ee * rho**2 / molarWeight**2 + gasConstant * temperature * rho / molarWeight * \
...            numerix.log(rho / (molarWeight - vbar * rho))
```

```
>>> def mu(rho):
...     return 2 * ee * rho / molarWeight**2 + gasConstant * temperature / molarWeight * \
...            (numerix.log(rho / (molarWeight - vbar * rho)) + molarWeight / (molarWeight - vbar * r
```

```
>>> def P(rho):
...     return rho * mu(rho) - f(rho)
```

The equilibrium densities values are verified with

```
>>> print numerix.allclose(mu(liquidDensity), mu(vaporDensity))
True
```

and

```
>>> print numerix.allclose(P(liquidDensity), P(vaporDensity))
True
```

In order to derive governing equations, the free energy functional is defined.

$$F = \int \left[f + \frac{\epsilon T}{2} \left(\partial_j \rho \right)^2 \right] dV$$

Using standard dissipation laws, we write the governing equations for mass and momentum conservation,

$$\frac{\partial \rho}{\partial t} + \partial_j \left(\rho u_j \right) = 0 \tag{16.6}$$

and

$$\frac{\partial \left(\rho u_i \right)}{\partial t} + \partial_j \left(\rho u_i u_j \right) = \partial_j \left(\nu \left[\partial_j u_i + \partial_i u_j \right] \right) - \rho \partial_i \mu^{NC} \tag{16.7}$$

where the non-classical potential, μ^{NC}, is given by,

$$\mu^{NC} = \frac{\delta F}{\delta \rho} = \mu - \epsilon T \partial_j^2 \rho \tag{16.8}$$

As usual, to proceed, we define a mesh

```
>>> Lx = 1e-6
>>> nx = 100
>>> dx = Lx / nx
>>> mesh = Grid1D(nx=nx, dx=dx)
```

and the independent variables.

```
>>> density = CellVariable(mesh=mesh, hasOld=True, name=r'$\rho$')
>>> velocity = CellVariable(mesh=mesh, hasOld=True, name=r'$u$')
>>> densityPrevious = density.copy()
>>> velocityPrevious = velocity.copy()
```

The system of equations is solved in a fully coupled manner using a block matrix. Defining μ^{NC} as an independent variable makes it easier to script the equations without using higher order terms.

```
>>> potentialNC = CellVariable(mesh=mesh, name=r'$\mu^{NC}$')
```

```
>>> epsilon = 1e-16
>>> freeEnergy = (f(density) + epsilon * temperature / 2 * density.grad.mag**2).cellVolumeAverage
```

In order to solve the equations numerically, an interpolation method is used to prevent the velocity and density fields decoupling. The following velocity correction equation (expressed in discretized form) prevents decoupling from occuring,

$$u^c_{i,f} = \frac{A_f d_f}{\overline{d}_f} \left(\overline{\rho \partial_i \mu^{NC}}_f - \overline{\rho}_f \partial_{i,f} \mu^{NC} \right) \tag{16.9}$$

where A_f is the face area, d_f is the distance between the adjacent cell centers and $\overline{a}_f$ is the momentum conservation equation's matrix diagonal. The overbar refers to an averaged value between the two adjacent cells to the face. The notation $\partial_{i,f}$ refers to a derivative evaluated directly at the face (not averaged). The variable u_i^c is used to modify the velocity used in Eq. (16.6) such that,

$$\frac{\partial \rho}{\partial t} + \partial_j \left(\rho \left[u_j + u_i^c \right] \right) = 0 \tag{16.10}$$

Equation (16.10) becomes

```
>>> matrixDiagonal = CellVariable(mesh=mesh, name=r'$a_f$', value=1e+20, hasOld=True)
>>> correctionCoeff = mesh._faceAreas * mesh._cellDistances / matrixDiagonal.faceValue
>>> massEqn = TransientTerm(var=density) \
...           + VanLeerConvectionTerm(coeff=velocity.faceValue + correctionCoeff \
...                                         * (density * potentialNC.grad).faceValue, \
...                                   var=density) \
...           - DiffusionTerm(coeff=correctionCoeff * density.faceValue**2, var=potentialNC)
```

where the first term on the LHS of Eq. (16.9) is calculated in an explicit manner in the `VanLeerConvectionTerm` and the second term is calculated implicitly as a `DiffusionTerm` with μ^{NC} as the independent variable.

In order to write Eq. (16.7) as a *FiPy* expression, the last term is rewritten such that,

$$\rho \partial_i \mu^{NC} = \partial_i \left(\rho \mu^{NC} \right) - \mu^{NC} \partial_i \rho$$

which results in

```
>>> viscosity = 1e-3
>>> ConvectionTerm = CentralDifferenceConvectionTerm
>>> momentumEqn = TransientTerm(coeff=density, var=velocity) \
...               + ConvectionTerm(coeff=[[1]] * density.faceValue * velocity.faceValue, var=velocity
...               == DiffusionTerm(coeff=2 * viscosity, var=velocity) \
...               - ConvectionTerm(coeff=density.faceValue * [[1]], var=potentialNC) \
...               + ImplicitSourceTerm(coeff=density.grad[0], var=potentialNC)
```

The only required boundary condition eliminates flow in or out of the domain.

```
>>> velocity.constrain(0, mesh.exteriorFaces)
```

As previously stated, the μ^{NC} variable will be solved implicitly. To do this the Eq. (16.8) is linearized in ρ such that

$$\mu^{NC} = \mu^* + \left(\frac{\partial \mu}{\partial \rho} \right)^* (\rho - \rho^*) - \epsilon T \partial_j^2 \rho \tag{16.11}$$

The * superscript denotes the current held value. In *FiPy*, $\frac{\partial \mu}{\partial \rho}$ is written as,

```
>>> potentialDerivative = 2 * ee / molarWeight**2 + gasConstant * temperature * molarWeight / density
```

and μ^* is simply,

```
>>> potential = mu(density)
```

Eq. (16.11) can be scripted as

```
>>> potentialNCEqn = ImplicitSourceTerm(coeff=1, var=potentialNC) \
...                  == potential \
...                  + ImplicitSourceTerm(coeff=potentialDerivative, var=density) \
...                  - potentialDerivative * density \
...                  - DiffusionTerm(coeff=epsilon * temperature, var=density)
```

Due to a quirk in *FiPy*, the gradient of μ^{NC} needs to be constrained on the boundary. This is because `ConvectionTerm`'s will automatically assume a zero flux, which is not what we need in this case.

```
>>> potentialNC.faceGrad.constrain(value=[0], where=mesh.exteriorFaces)
```

All three equations are defined and an are combined together with

```
>>> coupledEqn = massEqn & momentumEqn & potentialNCEqn
```

The system will be solved as a phase separation problem with an initial density close to the average density, but with some small amplitude noise. Under these circumstances, the final condition should be two separate phases of roughly equal volume. The initial condition for the density is defined by

```
>>> numerix.random.seed(2011)
>>> density[:] = (liquidDensity + vaporDensity) / 2 * \
...    (1  + 0.01 * (2 * numerix.random.random(mesh.numberOfCells) - 1))
```

Viewers are also defined.

```
>>> if __name__ == '__main__':
...     viewers = Viewer(density), Viewer(velocity), Viewer(potentialNC)
...     for viewer in viewers:
...         viewer.plot()
...     raw_input('arrange viewers')
...     for viewer in viewers:
...         viewer.plot()
```

The following section defines the required control parameters. The `cfl` parameter limits the size of the time step so that `dt = cfl * dx / max(velocity)`.

```
>>> cfl = 0.1
>>> tolerance = 1e-1
>>> dt = 1e-14
>>> timestep = 0
>>> relaxation = 0.5
>>> if __name__ == '__main__':
...     totalSteps = 1e10
... else:
...     totalSteps = 10
```

In the following time stepping scheme a time step is recalculated if the residual increases between sweeps or the required tolerance is not attained within 20 sweeps. The major quirk in this scheme is the requirement of updating the `matrixDiagonal` using the entire coupled matrix. This could be achieved more elegantly by calling `cacheMatrix()` only on the necessary part of the equation. This currently doesn't work properly in *FiPy*.

```
>>> while timestep < totalSteps:
...
...     sweep = 0
...     dt *= 1.1
...     residual = 1.
...     initialResidual = None
...
...     density.updateOld()
...     velocity.updateOld()
...     matrixDiagonal.updateOld()
...
...     while residual > tolerance:
...
...         densityPrevious[:] = density
...         velocityPrevious[:] = velocity
...         previousResidual = residual
...
...         dt = min(dt, dx / max(abs(velocity)) * cfl)
...
...         coupledEqn.cacheMatrix()
...         residual = coupledEqn.sweep(dt=dt)
...
...         if initialResidual is None:
...             initialResidual = residual
...
...         residual = residual / initialResidual
...
...         if residual > previousResidual * 1.1 or sweep > 20:
...             density[:] = density.old
...             velocity[:] = velocity.old
...             matrixDiagonal[:] = matrixDiagonal.old
...             dt = dt / 10.
...             if __name__ == '__main__':
...                 print 'Recalculate the time step'
...             timestep -= 1
...             break
...         else:
...             matrixDiagonal[:] = coupledEqn.matrix.takeDiagonal()[mesh.numberOfCells:2 * mesh.numb
...             density[:] = relaxation * density + (1 - relaxation) * densityPrevious
...             velocity[:] = relaxation * velocity + (1 - relaxation) * velocityPrevious
...
...         sweep += 1
...
...     if __name__ == '__main__' and timestep % 10 == 0:
...         print 'timestep: %i, dt: %1.5e, free energy: %1.5e' % (timestep, dt, freeEnergy)
...         for viewer in viewers:
...             viewer.plot()
...
...     timestep += 1

>>> if __name__ == '__main__':
...     raw_input('finished')

>>> print freeEnergy < 1.5e9
True
```

Chapter 17

Updating FiPy

examples.updating.update2_0to3_0	How to update scripts from version 2.0 to 3.0.
examples.updating.update1_0to2_0	How to update scripts from version 1.0 to 2.0.
examples.updating.update0_1to1_0	How to update scripts from version 0.1 to 1.0.

17.1 examples.updating.update2_0to3_0

How to update scripts from version 2.0 to 3.0.

FiPy 3.0 introduces several syntax changes from *FiPy* 2.0. We appreciate that this is very inconvenient for our users, but we hope you'll agree that the new syntax is easier to read and easier to use. We assure you that this is not something we do casually; it has been over two and a half years since our last incompatible change (when *FiPy* 2.0 superceded *FiPy* 1.0).

All examples included with version 3.0 have been updated to use the new syntax, but any scripts you have written for *FiPy* 2.0 will need to be updated. A complete listing of the changes needed to take the *FiPy* examples scripts from version 2.0 to version 3.0 can be found at

> http://www.matforge.org/fipy/wiki/upgrade2_0examplesTo3_0

but we summarize the necessary changes here. If these tips are not sufficient to make your scripts compatible with *FiPy* 3.0, please don't hesitate to ask for help on the mailing list.

The following items **must** be changed in your scripts

- We have reconsidered the change in FiPy 2.0 that included all of the functions of the `numerix` module in the `fipy` namespace. You now must be more explicit when referring to any of these functions:

  ```
  >>> from fipy import *
  >>> y = numerix.exp(x)

  >>> from fipy.tools.numerix import exp
  >>> y = exp(x)
  ```

  We generally use the first, but you may see us import specific functions if we feel it improves readability. You should feel free to use whichever form you find most comfortable.

  Note: the old behavior can be obtained, at least for now, by setting the `FIPY_INCLUDE_NUMERIX_ALL` environment variable.

- If your equation contains a `TransientTerm`, then you must specify the timestep by passing a `dt=` argument when calling `solve()` or `sweep()`.

The remaining changes are not *required*, but they make scripts easier to read and we recommend them. *FiPy* may issue a `DeprecationWarning` for some cases, to indicate that we may not maintain the old syntax indefinitely.

- "getter" and "setter" methods have been replaced with properties, e.g., use

  ```
  >>> x, y = mesh.cellCenters
  ```

  instead of

  ```
  >>> x, y = mesh.getCellCenters()
  ```

- Boundary conditions are better applied with the `constrain()` method than with the old `FixedValue` and `FixedFlux` classes. See *Boundary Conditions*.
- Individual `Mesh` classes should be imported directly from `fipy.meshes` and not `fipy.meshes.numMesh`.
- The *Gmsh* meshes now have simplified names: `Gmsh2D` instead of `GmshImporter2D`, `Gmsh3D` instead of `GmshImporter3D`, and `Gmsh2DIn3DSpace` instead of `GmshImporter2DIn3DSpace`.

17.2 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 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.getFaaceCenters()
>>> FixedValue(faces=mesh.getExteriorFaces() & (X**2 < 1e-6), value=...)
```

instead of

```
>>> exteriorFaces = mesh.getExteriorFaces()
>>> X = exteriorFaces.getCenters()[...,0]
>>> FixedValue(faces=exteriorFaces.where(X**2 < 1e-6), value=...)
```

With the old syntax, a different call to getCenters() had to be made for each set of Face objects. It was also extremely difficult to specify boundary conditions that depended both on position in space and on the current values of any other Variable.

```
>>> FixedValue(faces=(mesh.getExteriorFaces()
...                   & (((X**2 < 1e-6)
...                       & (Y > 3.))
...                      | (phi.getArithmeticFaceValue()
...                        < sin(gamma.getArithmeticFaceValue())))), value=...)
```

although it probably could have been done with a rather convoluted (and slow!) `filter` function passed to `where`. There no longer are any `filter` methods used in *FiPy*. You now would write

```
>>> x, y = mesh.cellCenters
>>> initialArray[(x < dx) | (x > (Lx - dx)) | (y < dy) | (y > (Ly - dy))] = 1.
```

instead of the *much* slower

```
>>> def cellFilter(cell):
...     return ((cell.center[0] < dx)
...             or (cell.center[0] > (Lx - dx))
...             or (cell.center[1] < dy)
...             or (cell.center[1] > (Ly - dy)))

>>> positiveCells = mesh.getCells(filter=cellFilter)
>>> for cell in positiveCells:
...     initialArray[cell.ID] = 1.
```

Although they still exist, we find very little cause to ever call `getCells()` or `fipy.meshes.mesh.Mesh.getFaces()`.

- Some modules, such as `fipy.solvers`, have been significantly rearranged. For example, you need to change

```
>>> from fipy.solvers.linearPCGSolver import LinearPCGSolver
```

to either

```
>>> from fipy import LinearPCGSolver
```

or

```
>>> from fipy.solvers.pysparse.linearPCGSolver import LinearPCGSolver
```

- The `numerix.max()` and `numerix.min()` functions no longer exist. Either call `max()` and `min()` or the `max()` and `min()` methods of a `Variable`.
- The `Numeric` module has not been supported for a long time. Be sure to use

```
>>> from fipy import numerix
```

instead of

```
>>> import Numeric
```

The remaining changes are not *required*, but they make scripts easier to read and we recommend them. *FiPy* may issue a `DeprecationWarning` for some cases, to indicate that we may not maintain the old syntax indefinitely.

- All of the most commonly used classes and functions in *FiPy* are directly accessible in the `fipy` namespace. For brevity, our examples now start with

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

instead of the explicit

```
>>> from fipy.meshes.grid1D import Grid1D
>>> from fipy.terms.powerLawConvectionTerm import PowerLawConvectionTerm
>>> from fipy.variables.cellVariable import CellVariable
```

imports that we used to use. Most of the explicit imports should continue to work, so you do not need to change them if you don't wish to, but we find our own scripts much easier to read without them.

All of the `numerix` module is now imported into the `fipy` namespace, so you can call `numerix` functions a number of different ways, including:

```
>>> from fipy import *
>>> y = exp(x)
```

or

```
>>> from fipy import numerix
>>> y = numerix.exp(x)
```

or

```
>>> from fipy.tools.numerix import exp
>>> y = exp(x)
```

We generally use the first, but you may see us use the others, and should feel free to use whichever form you find most comfortable.

Note: Internally, *FiPy* uses explicit imports, as is considered best Python practice, but we feel that clarity trumps orthodoxy when it comes to the examples.

- The function `fipy.viewers.make()` has been renamed to `fipy.viewers.Viewer()`. All of the `limits` can now be supplied as direct arguments, as well (although this is not required). The result is a more natural syntax:

```
>>> from fipy import Viewer
>>> viewer = Viewer(vars=(alpha, beta, gamma), datamin=0, datamax=1)
```

  instead of

```
>>> from fipy import viewers
>>> viewer = viewers.make(vars=(alpha, beta, gamma),
...                       limits={'datamin': 0, 'datamax': 1})
```

  With the old syntax, there was also a temptation to write

```
>>> from fipy.viewers import make
>>> viewer = make(vars=(alpha, beta, gamma))
```

  which can be very hard to understand after the fact (`make`? `make` what?).

- A `ConvectionTerm` can now calculate its 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 partion 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`.

17.3 examples.updating.update0_1to1_0

How to update scripts from version 0.1 to 1.0.

It seems unlikely that many users are still running *FiPy* 0.1, but for those that are, the syntax of *FiPy* scripts changed considerably between version 0.1 and version 1.0. We incremented the full version-number to stress that previous scripts are incompatible. We strongly believe that these changes are for the better, resulting in easier code to write and read as well as slightly improved efficiency, but we realize that this represents an inconvenience to our users that have already written scripts of their own. We will strive to avoid any such incompatible changes in the future.

Any scripts you have written for *FiPy* 0.1 should be updated in two steps, first to work with *FiPy* 1.0, and then with *FiPy* 2.0. As a tutorial for updating your scripts, we will walk through updating `examples/convection/exponential1D/input.py` from *FiPy* 0.1. If you attempt to run that script with *FiPy* 1.0, the script will fail and you will see the errors shown below:

This example solves the steady-state convection-diffusion equation given by:

$$\nabla \cdot (D \nabla \phi + \vec{u} \phi) = 0$$

with coefficients $D = 1$ and $\vec{u} = (10, 0)$, or

```
>>> diffCoeff = 1.
>>> convCoeff = (10.,0.)
```

We define a 1D mesh

```
>>> L = 10.
>>> nx = 1000
>>> ny = 1
>>> from fipy.meshes.grid2D import Grid2D
>>> mesh = Grid2D(L / nx, L / ny, nx, ny)
```

and impose the boundary conditions

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

or

```
>>> valueLeft = 0.
>>> valueRight = 1.
>>> from fipy.boundaryConditions.fixedValue import FixedValue
>>> from fipy.boundaryConditions.fixedFlux import FixedFlux
>>> boundaryConditions = (
...     FixedValue(mesh.getFacesLeft(), valueLeft),
...     FixedValue(mesh.getFacesRight(), valueRight),
...     FixedFlux(mesh.getFacesTop(), 0.),
...     FixedFlux(mesh.getFacesBottom(), 0.)
...     )
```

The solution variable is initialized to *valueLeft*:

```
>>> from fipy.variables.cellVariable import CellVariable
>>> var = CellVariable(
...     name = "concentration",
...     mesh = mesh,
...     value = valueLeft)
```

The `SteadyConvectionDiffusionScEquation` object is used to create the equation. It needs to be passed a convection term instantiator as follows:

```
>>> from fipy.terms.exponentialConvectionTerm import ExponentialConvectionTerm
>>> from fipy.solvers import *
>>> from fipy.equations.stdyConvDiffScEquation import SteadyConvectionDiffusionScEquation
Traceback (most recent call last):
...
ImportError: No module named equations.stdyConvDiffScEquation
>>> eq = SteadyConvectionDiffusionScEquation(
...      var = var,
...      diffusionCoeff = diffCoeff,
...      convectionCoeff = convCoeff,
...      solver = LinearLUSolver(tolerance = 1.e-15, steps = 2000),
...      convectionScheme = ExponentialConvectionTerm,
...      boundaryConditions = boundaryConditions
...      )
Traceback (most recent call last):
...
NameError: name 'SteadyConvectionDiffusionScEquation' is not defined
```

More details of the benefits and drawbacks of each type of convection term can be found in the numerical section of the manual. Essentially the `ExponentialConvectionTerm` and `PowerLawConvectionTerm` will both handle most types of convection diffusion cases with the `PowerLawConvectionTerm` being more efficient.

We iterate to equilibrium

```
>>> from fipy.iterators.iterator import Iterator
>>> it = Iterator((eq,))
Traceback (most recent call last):
...
NameError: name 'eq' is not defined
>>> it.timestep()
Traceback (most recent call last):
```

```
...
NameError: name 'it' is not defined
```

and test the solution against the analytical result

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

or

```
>>> axis = 0
>>> x = mesh.getCellCenters()[:,axis]
>>> from fipy.tools import numerix
>>> CC = 1. - numerix.exp(-convCoeff[axis] * x / diffCoeff)
>>> DD = 1. - numerix.exp(-convCoeff[axis] * L / diffCoeff)
>>> analyticalArray = CC / DD
>>> numerix.allclose(analyticalArray, var, rtol = 1e-10, atol = 1e-10)
0
```

If the problem is run interactively, we can view the result:

```
>>> if __name__ == '__main__':
...     from fipy.viewers.grid2DGistViewer import Grid2DGistViewer
Traceback (most recent call last):
...
ImportError: No module named grid2DGistViewer

...     viewer = Grid2DGistViewer(var)
...     viewer.plot()
```

We see that a number of errors are thrown:

- `ImportError: No module named equations.stdyConvDiffScEquation`
- `NameError: name 'SteadyConvectionDiffusionScEquation' is not defined`
- `NameError: name 'eq' is not defined`
- `NameError: name 'it' is not defined`
- `ImportError: No module named grid2DGistViewer`

As is usually the case with computer programming, many of these errors are caused by earlier errors. Let us update the script, section by section:

Although no error was generated by the use of `Grid2D`, *FiPy* 1.0 supports a true 1D mesh class, so we instantiate the mesh as

```
>>> L = 10.
>>> nx = 1000
>>> from fipy.meshes.grid1D import Grid1D
>>> mesh = Grid1D(dx = L / nx, nx = nx)
```

The `Grid2D` class with *ny = 1* still works perfectly well for 1D problems, but the `Grid1D` class is slightly more efficient, and it makes the code clearer when a 1D geometry is actually desired.

Because the mesh is now 1D, we must update the convection coefficient vector to be 1D as well

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

The `FixedValue` boundary conditions at the left and right are unchanged, but a *Grid1D* mesh does not even have top and bottom faces:

```
>>> valueLeft = 0.
>>> valueRight = 1.
>>> from fipy.boundaryConditions.fixedValue import FixedValue
>>> boundaryConditions = (
...     FixedValue(mesh.getFacesLeft(), valueLeft),
...     FixedValue(mesh.getFacesRight(), valueRight))
```

The creation of the solution variable is unchanged:

```
>>> from fipy.variables.cellVariable import CellVariable
>>> var = CellVariable(name = "concentration",
...                    mesh = mesh,
...                    value = valueLeft)
```

The biggest change between *FiPy* 0.1 and *FiPy* 1.0 is that `Equation` objects no longer exist at all. Instead, `Term` objects can be simply added, subtracted, and equated to assemble an equation. Where before the assembly of the equation occurred in the black-box of `SteadyConvectionDiffusionScEquation`, we now assemble it directly:

```
>>> from fipy.terms.implicitDiffusionTerm import ImplicitDiffusionTerm
>>> diffTerm = ImplicitDiffusionTerm(coeff = diffCoeff)

>>> from fipy.terms.exponentialConvectionTerm import ExponentialConvectionTerm
>>> eq = diffTerm + ExponentialConvectionTerm(coeff = convCoeff,
...                                           diffusionTerm = diffTerm)
```

One thing that `SteadyConvectionDiffusionScEquation` took care of automatically was that a `ConvectionTerm` must know about any `DiffusionTerm` in the equation in order to calculate a 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.

Part III

`fipy` Package Documentation

Chapter 18

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.

18.1 subpackage Package

18.1.1 `subpackage` Package

Each chapter describes one of the main sub-packages of the `fipy` package. The sub-package `fipy.package` can be found in the directory `fipy/package/`. In a few cases, there will be packages within packages, *e.g.* `fipy.package.subpackage` located in `fipy/package/subpackage/`. These sub-sub-packages will not be given their own chapters; rather, their contents will be described in the chapter for their containing package.

18.1.2 `base` Module

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

```
import package.subpackage.base
```

in which case you refer to it by its full name of `package.subpackage.base`, or:

```
from package.subpackage import base
```

in which case you can refer simply to `base`.

class `package.subpackage.base.`**`Base`**

With very few exceptions, the name of a class will be the capitalized form of the module it resides in. Depending on how you imported the module above, you will refer to either `package.subpackage.object.Object` or `object.Object`. Alternatively, you can use:

```
from package.subpackage.object import Object
```

and then refer simply to `Object`. For many classes, there is a shorthand notation:

```
from fipy import Object
```

Python is an object-oriented language and the FiPy framework is composed of objects or classes. Knowledge of object-oriented programming (OOP) is not necessary to use either Python or FiPy, but a few concepts are useful. OOP involves two main ideas:

encapsulation an object binds data with actions or "methods". In most cases, you will not work with an object's data directly; instead, you will set, retrieve, or manipulate the data using the object's methods.

Methods are functions that are attached to objects and that have direct access to the data of those objects. Rather than passing the object data as an argument to a function:

```
fn(data, arg1, arg2, ...)
```

you instruct an object to invoke an appropriate method:

```
object.meth(arg1, arg2, ...)
```

If you are unfamiliar with object-oriented practices, there probably seems little advantage in this reordering. You will have to trust us that the latter is a much more powerful way to do things.

inheritance specialized objects are derived or inherited from more general objects. Common behaviors or data are defined in base objects and specific behaviors or data are either added or modified in derived objects. Objects that declare the existence of certain methods, without actually defining what those methods do, are called "abstract". These objects exist to define the behavior of a family of objects, but rely on their descendants to actually provide that behavior.

Unlike many object-oriented languages, *Python* does not prevent the creation of abstract objects, but we will include a notice like

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

for abstract classes which should be used for documentation but never actually created in a *FiPy* script.

method1()

This is one thing that you can instruct any object that derives from `Base` to do, by calling `myObjectDerivedFromBase.method1()`

Parameters

- *self*: this special argument refers to the object that is being created.

> **Attention:** *self* is supplied automatically by the *Python* interpreter to all methods. You don't need to (and should not) specify it yourself.

method2()

This is another thing that you can instruct any object that derives from `Base` to do.

18.1.3 `object` Module

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

Bases: `package.subpackage.base.Base`

This method, like all those whose names begin and end with "__" are special. You won't ever need to call these methods directly, but *Python* will invoke them for you under certain circumstances, which are described in the Python Reference Manual: Special Method Names.

As an example, the `__init__()` method is invoked when you create an object, as in:

```
obj = Object(arg1=something, arg3=somethingElse, ...)
```

Parameters

- *arg1*: this argument is required. *Python* supports named arguments, so you must either list the value for *arg1* first:

  ```
  obj = Object(val1, val2)
  ```

  or you can specify the arguments in any order, as long as they are named:

  ```
  obj = Object(arg2=val2, arg1=val1)
  ```

- *arg2*: this argument may be omitted, in which case it will be assigned a default value of `None`. If you do not use named arguments (and we recommend that you do), all required arguments must be specified before any optional arguments.
- *arg3*: this argument may be omitted, in which case it will be assigned a default value of `'string'`.

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

Chapter 19

boundaryConditions Package

19.1 `boundaryConditions` Package

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

Bases: `object`

Object to hold a *Variable* to *value* at *where*

see `constrain()`

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

Bases: `fipy.boundaryConditions.boundaryCondition.BoundaryCondition`

The *FixedFlux* boundary condition adds a contribution, equivalent to a fixed flux (Neumann condition), to the equation's RHS vector. The contribution, given by *value*, is only added to entries corresponding to the specified *faces*, and is weighted by the face areas.

Creates a *FixedFlux* object.

Parameters

- *faces*: A *list* or *tuple* of *Face* objects to which this condition applies.
- *value*: The value to impose.

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

Bases: `fipy.boundaryConditions.boundaryCondition.BoundaryCondition`

The *FixedValue* boundary condition adds a contribution, equivalent to a fixed value (Dirichlet condition), to the equation's RHS vector and coefficient matrix. The contributions are given by $-\mathtt{value} \times G_{\text{face}}$ for the RHS vector and G_{face} for the coefficient matrix. The parameter G_{face} represents the term's geometric coefficient, which depends on the type of term and the mesh geometry.

Contributions are only added to entries corresponding to the specified faces.

Parameters

- *faces*: A *list* or *tuple* of exterior *Face* objects to which this condition applies.
- *value*: The value to impose.

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

Bases: `fipy.boundaryConditions.boundaryCondition.BoundaryCondition`

This boundary condition is generally used in conjunction with a *ImplicitDiffusionTerm* that has multiple coefficients. It does not have any direct effect on the solution matrices, but its derivatives do.

Creates an *NthOrderBoundaryCondition*.

Parameters

- *faces*: 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.

19.2 `boundaryCondition` Module

class `fipy.boundaryConditions.boundaryCondition.`**`BoundaryCondition`**(*faces*, *value*)

Bases: `object`

Generic boundary condition base class.

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

Parameters

- *faces*: A *list* or *tuple* of exterior *Face* objects to which this condition applies.
- *value*: The value to impose.

19.3 `constraint` Module

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

Bases: `object`

Object to hold a *Variable* to *value* at *where*

see `constrain()`

19.4 `fixedFlux` Module

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

Bases: `fipy.boundaryConditions.boundaryCondition.BoundaryCondition`

The *FixedFlux* boundary condition adds a contribution, equivalent to a fixed flux (Neumann condition), to the equation's RHS vector. The contribution, given by *value*, is only added to entries corresponding to the specified *faces*, and is weighted by the face areas.

Creates a *FixedFlux* object.

Parameters

- *faces*: A *list* or *tuple* of *Face* objects to which this condition applies.
- *value*: The value to impose.

19.5 `fixedValue` Module

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

Bases: `fipy.boundaryConditions.boundaryCondition.BoundaryCondition`

The *FixedValue* boundary condition adds a contribution, equivalent to a fixed value (Dirichlet condition), to the equation's RHS vector and coefficient matrix. The contributions are given by $-\mathtt{value} \times G_{\text{face}}$ for the RHS vector and G_{face} for the coefficient matrix. The parameter G_{face} represents the term's geometric coefficient, which depends on the type of term and the mesh geometry.

Contributions are only added to entries corresponding to the specified faces.

Parameters

- *faces*: A *list* or *tuple* of exterior *Face* objects to which this condition applies.
- *value*: The value to impose.

19.6 `nthOrderBoundaryCondition` Module

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

Bases: `fipy.boundaryConditions.boundaryCondition.BoundaryCondition`

This boundary condition is generally used in conjunction with a *ImplicitDiffusionTerm* that has multiple coefficients. It does not have any direct effect on the solution matrices, but its derivatives do.

Creates an *NthOrderBoundaryCondition*.

Parameters

- *faces*: 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.

19.7 `test` Module

Test numeric implementation of the mesh

Chapter 20

matrices Package

20.1 `offsetSparseMatrix` Module

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

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

20.2 `pysparseMatrix` Module

20.3 `scipyMatrix` Module

20.4 `sparseMatrix` Module

20.5 `test` Module

20.6 `trilinosMatrix` Module

Chapter 21

meshes Package

21.1 meshes Package

`fipy.meshes.`**`Grid3D`**(*dx=1.0*, *dy=1.0*, *dz=1.0*, *nx=None*, *ny=None*, *nz=None*, *Lx=None*, *Ly=None*, *Lz=None*, *overlap=2*, *communicator=ParallelCommWrapper()*)

Factory function to select between UniformGrid3D and Grid3D. If *Lx* is specified the length of the domain is always *Lx* regardless of *dx*.

Parameters

- *dx*: grid spacing in the horizontal direction
- *dy*: grid spacing in the vertical direction
- *dz*: grid spacing in the z-direction
- *nx*: number of cells in the horizontal direction
- *ny*: number of cells in the vertical direction
- *nz*: number of cells in the z-direction
- *Lx*: the domain length in the horizontal direction
- *Ly*: the domain length in the vertical direction
- *Lz*: the domain length in the z-direction
- *overlap*: the number of overlapping cells for parallel simulations. Generally 2 is adequate. Higher order equations or discretizations require more.
- *communicator*: either *fipy.tools.parallel* or *fipy.tools.serial*. Select *fipy.tools.serial* to create a serial mesh when running in parallel. Mostly used for test purposes.

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

Factory function to select between UniformGrid2D and Grid2D. If *Lx* is specified the length of the domain is always *Lx* regardless of *dx*.

Parameters

- *dx*: grid spacing in the horizontal direction
- *dy*: grid spacing in the vertical direction
- *nx*: number of cells in the horizontal direction
- *ny*: number of cells in the vertical direction
- *Lx*: the domain length in the horizontal direction
- *Ly*: the domain length in the vertical direction

- *overlap*: the number of overlapping cells for parallel simulations. Generally 2 is adequate. Higher order equations or discretizations require more.
- *communicator*: either *fipy.tools.parallel* or *fipy.tools.serial*. Select *fipy.tools.serial* to create a serial mesh when running in parallel. Mostly used for test purposes.

```
>>> print Grid2D(Lx=3., nx=2).dx
1.5
```

fipy.meshes.**Grid1D**(*dx=1.0*, *nx=None*, *Lx=None*, *overlap=2*, *communicator=ParallelCommWrapper()*)

Factory function to select between UniformGrid1D and Grid1D. If *Lx* is specified the length of the domain is always *Lx* regardless of *dx*.

Parameters

- *dx*: grid spacing in the horizonal direction
- *nx*: number of cells in the horizonal direction
- *Lx*: the domain length in the horizonal direction
- *overlap*: the number of overlapping cells for parallel simulations. Generally 2 is adequate. Higher order equations or discretizations require more.
- *communicator*: either *fipy.tools.parallel* or *fipy.tools.serial*. Select *fipy.tools.serial* to create a serial mesh when running in parallel. Mostly used for test purposes.

fipy.meshes.**CylindricalGrid2D**(*dr=None*, *dz=None*, *nr=None*, *nz=None*, *Lr=None*, *Lz=None*, *dx=1.0*, *dy=1.0*, *nx=None*, *ny=None*, *Lx=None*, *Ly=None*, *origin=((0,), (0,))*, *overlap=2*, *communicator=ParallelCommWrapper()*)

Factory function to select between CylindricalUniformGrid2D and CylindricalGrid2D. If *Lx* is specified the length of the domain is always *Lx* regardless of *dx*.

Parameters

- *dr* or *dx*: grid spacing in the radial direction
- *dz* or *dy*: grid spacing in the vertical direction
- *nr* or *nx*: number of cells in the radial direction
- *nz* or *ny*: number of cells in the vertical direction
- *Lr* or *Lx*: the domain length in the radial direction
- *Lz* or *Ly*: the domain length in the vertical direction
- *origin* : position of the mesh's origin in the form ((x,),(y,))
- *overlap*: the number of overlapping cells for parallel simulations. Generally 2 is adequate. Higher order equations or discretizations require more.
- *communicator*: either *fipy.tools.parallel* or *fipy.tools.serial*. Select *fipy.tools.serial* to create a serial mesh when running in parallel. Mostly used for test purposes.

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

Factory function to select between CylindricalUniformGrid1D and CylindricalGrid1D. If *Lx* is specified the length of the domain is always *Lx* regardless of *dx*.

Parameters

- *dr* or *dx*: grid spacing in the radial direction

- *nr* or *nx*: number of cells in the radial direction
- *Lr* or *Lx*: the domain length in the radial direction
- *origin* : position of the mesh's origin in the form (x,)
- *overlap*: the number of overlapping cells for parallel simulations. Generally 2 is adequate. Higher order equations or discretizations require more.
- *communicator*: either *fipy.tools.parallel* or *fipy.tools.serial*. Select *fipy.tools.serial* to create a serial mesh when running in parallel. Mostly used for test purposes.

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

Bases: fipy.meshes.grid1D.Grid1D

Creates a Periodic grid mesh.

```
>>> mesh = PeriodicGrid1D(dx = (1, 2, 3))

>>> print numerix.allclose(numerix.nonzero(mesh.exteriorFaces)[0],
...                        [3])
True

>>> print numerix.allclose(mesh.faceCellIDs.filled(-999),
...                        [[2, 0, 1, 2],
...                         [0, 1, 2, -999]])
True

>>> print numerix.allclose(mesh._cellDistances,
...                        [ 2., 1.5, 2.5, 1.5])
True

>>> print numerix.allclose(mesh._cellToCellDistances,
...                        [[ 2.,   1.5,   2.5],
...                         [ 1.5,   2.5,   2. ]])
True

>>> print numerix.allclose(mesh._faceNormals,
...                        [[ 1.,   1.,   1.,   1.]])
True

>>> print numerix.allclose(mesh._cellVertexIDs,
...                        [[1, 2, 2],
...                        [0, 1, 0]])
True
```

cellCenters

Defined outside of a geometry class since we need the *CellVariable* version of *cellCenters*; that is, the *cellCenters* defined in fipy.meshes.mesh and not in any geometry (since a *CellVariable* requires a reference to a mesh).

class fipy.meshes.**PeriodicGrid2D**(*dx=1.0*, *dy=1.0*, *nx=None*, *ny=None*, *overlap=2*, *communicator=ParallelCommWrapper()*)

Bases: fipy.meshes.periodicGrid2D._BasePeriodicGrid2D

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
```

```
>>> mesh = PeriodicGrid2D(dx = 1., dy = 0.5, nx = 2, ny = 2)

>>> print numerix.allclose(numerix.nonzero(mesh.exteriorFaces)[0],
...                        [ 4,   5,   8, 11])
True

>>> print numerix.allclose(mesh.faceCellIDs.filled(-1),
...                        [[2, 3, 0, 1, 2, 3, 1, 0, 1, 3, 2, 3],
...                         [0, 1, 2, 3, -1, -1, 0, 1, -1, 2, 3, -1]])
True

>>> print numerix.allclose(mesh._cellDistances,
...                        [ 0.5, 0.5, 0.5, 0.5, 0.25, 0.25, 1., 1., 0.5, 1., 1., 0.5])
True

>>> print numerix.allclose(mesh.cellFaceIDs,
...                        [[0, 1, 2, 3],
...                         [7, 6, 10, 9],
...                         [2, 3, 0, 1],
...                         [6, 7, 9, 10]])
True

>>> print numerix.allclose(mesh._cellToCellDistances,
...                        [[ 0.5, 0.5, 0.5, 0.5],
...                         [ 1., 1., 1., 1. ],
...                         [ 0.5, 0.5, 0.5, 0.5],
...                         [ 1., 1., 1., 1. ]])
True

>>> normals = [[0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1],
...            [1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0]]

>>> print numerix.allclose(mesh._faceNormals, normals)
True

>>> print numerix.allclose(mesh._cellVertexIDs,
...                        [[4, 5, 7, 8],
...                         [3, 4, 6, 7],
...                         [1, 2, 4, 5],
...                         [0, 1, 3, 4]])
True
```

class `fipy.meshes.`**`PeriodicGrid2DLeftRight`**(*dx=1.0*, *dy=1.0*, *nx=None*, *ny=None*, *overlap=2*, *communicator=ParallelCommWrapper()*)

Bases: `fipy.meshes.periodicGrid2D._BasePeriodicGrid2D`

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

Bases: `fipy.meshes.periodicGrid2D._BasePeriodicGrid2D`

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

Bases: `fipy.meshes.mesh2D.Mesh2D`

Creates a 2D grid mesh with horizontal faces numbered first and then vertical faces. The points are skewed by a random amount (between *rand* and *-rand*) in the X and Y directions.

`physicalShape`

Return physical dimensions of Grid2D.

`shape`

class `fipy.meshes.Tri2D`(*dx=1.0*, *dy=1.0*, *nx=1*, *ny=1*, *_RepresentationClass=<class 'fipy.meshes.representations.gridRepresentation._Grid2DRepresentation'>*, *_TopologyClass=<class 'fipy.meshes.topologies.meshTopology._Mesh2DTopology'>*)

Bases: `fipy.meshes.mesh2D.Mesh2D`

This class creates a mesh made out of triangles. It does this by starting with a standard Cartesian mesh (*Grid2D*) and dividing each cell in that mesh (hereafter referred to as a 'box') into four equal parts with the dividing lines being the diagonals.

Creates a 2D triangular mesh with horizontal faces numbered first then vertical faces, then diagonal faces. Vertices are numbered starting with the vertices at the corners of boxes and then the vertices at the centers of boxes. Cells on the right of boxes are numbered first, then cells on the top of boxes, then cells on the left of boxes, then cells on the bottom of boxes. Within each of the 'sub-categories' in the above, the vertices, cells and faces are numbered in the usual way.

Parameters

- *dx, dy*: The X and Y dimensions of each 'box'. If *dx* <> *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*.

physicalShape

Return physical dimensions of Grid2D.

shape

`fipy.meshes.openMSHFile`(*name*, *dimensions=None*, *coordDimensions=None*, *communicator=ParallelCommWrapper()*, *order=1*, *mode='r'*, *background=None*)

Open a Gmsh MSH file

Parameters

- *filename*: a string indicating gmsh output file
- *dimensions*: an integer indicating dimension of mesh
- *coordDimensions*: an integer indicating dimension of shapes
- *order*: ???
- *mode*: a string beginning with 'r' for reading and 'w' for writing. The file will be created if it doesn't exist when opened for writing; it will be truncated when opened for writing. Add a 'b' to the mode for binary files.
- *background*: a *CellVariable* that specifies the desired characteristic lengths of the mesh cells

`fipy.meshes.openPOSFile`(*name*, *communicator=ParallelCommWrapper()*, *mode='w'*)

Open a Gmsh POS post-processing file

class `fipy.meshes.Gmsh2D`(*arg*, *coordDimensions=2*, *communicator=ParallelCommWrapper()*, *order=1*, *background=None*)

Bases: `fipy.meshes.mesh2D.Mesh2D`

Construct a 2D Mesh using Gmsh

```
>>> radius = 5.
>>> side = 4.
>>> squaredCircle = Gmsh2D('''
... // A mesh consisting of a square inside a circle inside a circle
...
... // define the basic dimensions of the mesh
...
```

```
... cellSize = 1;
... radius = %(radius)g;
... side = %(side)g;
...
... // define the compass points of the inner circle
...
... Point(1) = {0, 0, 0, cellSize};
... Point(2) = {-radius, 0, 0, cellSize};
... Point(3) = {0, radius, 0, cellSize};
... Point(4) = {radius, 0, 0, cellSize};
... Point(5) = {0, -radius, 0, cellSize};
...
... // define the compass points of the outer circle
...
... Point(6) = {-2*radius, 0, 0, cellSize};
... Point(7) = {0, 2*radius, 0, cellSize};
... Point(8) = {2*radius, 0, 0, cellSize};
... Point(9) = {0, -2*radius, 0, cellSize};
...
... // define the corners of the square
...
... Point(10) = {side/2, side/2, 0, cellSize/2};
... Point(11) = {-side/2, side/2, 0, cellSize/2};
... Point(12) = {-side/2, -side/2, 0, cellSize/2};
... Point(13) = {side/2, -side/2, 0, cellSize/2};
...
... // define the inner circle
...
... Circle(1) = {2, 1, 3};
... Circle(2) = {3, 1, 4};
... Circle(3) = {4, 1, 5};
... Circle(4) = {5, 1, 2};
...
... // define the outer circle
...
... Circle(5) = {6, 1, 7};
... Circle(6) = {7, 1, 8};
... Circle(7) = {8, 1, 9};
... Circle(8) = {9, 1, 6};
...
... // define the square
...
... Line(9) = {10, 13};
... Line(10) = {13, 12};
... Line(11) = {12, 11};
... Line(12) = {11, 10};
...
... // define the three boundaries
...
... Line Loop(1) = {1, 2, 3, 4};
... Line Loop(2) = {5, 6, 7, 8};
... Line Loop(3) = {9, 10, 11, 12};
...
... // define the three domains
...
... Plane Surface(1) = {2, 1};
... Plane Surface(2) = {1, 3};
... Plane Surface(3) = {3};
```

```
...
... // label the three domains
...
... // attention: if you use any "Physical" labels, you *must* label
... // all elements that correspond to FiPy Cells (Physical Surace in 2D
... // and Physical Volume in 3D) or Gmsh will not include them and FiPy
... // will not be able to include them in the Mesh.
...
... // note: if you do not use any labels, all Cells will be included.
...
... Physical Surface("Outer") = {1};
... Physical Surface("Middle") = {2};
... Physical Surface("Inner") = {3};
...
... // label the "north-west" part of the exterior boundary
...
... // note: you only need to label the Face elements
... // (Physical Line in 2D and Physical Surface in 3D) that correspond
... // to boundaries you are interested in. FiPy does not need them to
... // construct the Mesh.
...
... Physical Line("NW") = {5};
... ''' % locals())
```

It can be easier to specify certain domains and boundaries within Gmsh than it is to define the same domains and boundaries with FiPy expressions.

Here we compare obtaining the same Cells and Faces using FiPy's parametric descriptions and Gmsh's labels.

```
>>> x, y = squaredCircle.cellCenters

>>> middle = ((x**2 + y**2 <= radius**2)
...           & ~((x > -side/2) & (x < side/2)
...               & (y > -side/2) & (y < side/2)))

>>> print (middle == squaredCircle.physicalCells["Middle"]).all()
True

>>> X, Y = squaredCircle.faceCenters

>>> NW = ((X**2 + Y**2 > (1.99*radius)**2)
...       & (X**2 + Y**2 < (2.01*radius)**2)
...       & (X <= 0) & (Y >= 0))

>>> print (NW == squaredCircle.physicalFaces["NW"]).all()
True
```

It is possible to direct Gmsh to give the mesh different densities in different locations

```
>>> geo = '''
... // A mesh consisting of a square
...
... // define the corners of the square
...
... Point(1) = {1, 1, 0, 1};
... Point(2) = {0, 1, 0, 1};
... Point(3) = {0, 0, 0, 1};
... Point(4) = {1, 0, 0, 1};
...
... // define the square
```

```
...
... Line(1) = {1, 2};
... Line(2) = {2, 3};
... Line(3) = {3, 4};
... Line(4) = {4, 1};
...
... // define the boundary
...
... Line Loop(1) = {1, 2, 3, 4};
...
... // define the domain
...
... Plane Surface(1) = {1};
... '''

>>> from fipy import CellVariable, numerix

>>> std = []
>>> bkg = None
>>> for refine in range(4):
...     square = Gmsh2D(geo, background=bkg)
...     x, y = square.cellCenters
...     bkg = CellVariable(mesh=square, value=abs(x / 4) + 0.01)
...     std.append(numerix.std(numerix.sqrt(2 * square.cellVolumes) / bkg))
```

Check that the mesh is monotonically approaching the desired density

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

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

```
>>> print std[-1] < 0.2
True
```

The initial mesh doesn't have to be from Gmsh

```
>>> from fipy import Tri2D

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

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

>>> print std1 > std2
True
```

Parameters

- ***arg*: a string giving (i) the path to an MSH file, (ii) a path to a** Gmsh geometry (".geo") file, or (iii) a Gmsh geometry script
- *coordDimensions*: an integer indicating dimension of shapes
- *order*: ???

- *background*: a *CellVariable* that specifies the desired characteristic lengths of the mesh cells

class `fipy.meshes.`**`Gmsh2DIn3DSpace`**(*arg*, *communicator=ParallelCommWrapper()*, *order=1*, *background=None*)

Bases: `fipy.meshes.gmshMesh.Gmsh2D`

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

Parameters

- ***arg*: a string giving (i) the path to an MSH file, (ii) a path to a** Gmsh geometry (".geo") file, or (iii) a Gmsh geometry script
- *coordDimensions*: an integer indicating dimension of shapes
- *order*: ???
- *background*: a *CellVariable* that specifies the desired characteristic lengths of the mesh cells

class `fipy.meshes.`**`Gmsh3D`**(*arg*, *communicator=ParallelCommWrapper()*, *order=1*, *background=None*)

Bases: `fipy.meshes.mesh.Mesh`

Create a 3D Mesh using Gmsh

Parameters

- ***arg*: a string giving (i) the path to an MSH file, (ii) a path to a** Gmsh geometry (".geo") file, or (iii) a Gmsh geometry script
- *order*: ???
- *background*: a *CellVariable* that specifies the desired characteristic lengths of the mesh cells

class `fipy.meshes.`**`GmshGrid2D`**(*dx=1.0*, *dy=1.0*, *nx=1*, *ny=None*, *coordDimensions=2*, *communicator=ParallelCommWrapper()*, *order=1*)

Bases: `fipy.meshes.gmshMesh.Gmsh2D`

Should serve as a drop-in replacement for Grid2D.

class `fipy.meshes.`**`GmshGrid3D`**(*dx=1.0*, *dy=1.0*, *dz=1.0*, *nx=1*, *ny=None*, *nz=None*, *communicator=ParallelCommWrapper()*, *order=1*)

Bases: `fipy.meshes.gmshMesh.Gmsh3D`

Should serve as a drop-in replacement for Grid3D.

class `fipy.meshes.`**`GmshImporter2D`**(*arg*, *coordDimensions=2*)

Bases: `fipy.meshes.gmshMesh.Gmsh2D`

class `fipy.meshes.`**`GmshImporter2DIn3DSpace`**(*arg*)

Bases: `fipy.meshes.gmshMesh.Gmsh2DIn3DSpace`

class `fipy.meshes.`**`GmshImporter3D`**(*arg*)

Bases: `fipy.meshes.gmshMesh.Gmsh3D`

21.2 `abstractMesh` Module

class `fipy.meshes.abstractMesh.`**`AbstractMesh`**(*communicator*, *_RepresentationClass=<class 'fipy.meshes.representations.abstractRepresentation._AbstractRepresenta*, *_TopologyClass=<class 'fipy.meshes.topologies.abstractTopology._AbstractTopology'>*)

Bases: `object`

A class encapsulating all commonalities among meshes in FiPy.

VTKCellDataSet
Returns a TVTK *DataSet* representing the cells of this mesh

VTKFaceDataSet
Returns a TVTK *DataSet* representing the face centers of this mesh

aspect2D
The physical y:x aspect ratio of a 2D mesh

cellCenters

cellDistanceVectors

cellFaceIDs
Topology properties

cellToFaceDistanceVectors

cellVolumes

exteriorFaces

faceCenters

facesBack
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.)
>>> print numerix.allequal((6, 7, 8, 9, 10, 11),
...                        numerix.nonzero(mesh.facesBack)[0])
True
```

facesBottom
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.)
>>> print numerix.allequal((12, 13, 14),
...                        numerix.nonzero(mesh.facesBottom)[0])
1
>>> x, y, z = mesh.faceCenters
>>> print numerix.allequal((12, 13),
...                        numerix.nonzero(mesh.facesBottom & (x < 1))[0])
1
```

facesDown
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.)
>>> print numerix.allequal((12, 13, 14),
...                        numerix.nonzero(mesh.facesBottom)[0])
1
>>> x, y, z = mesh.faceCenters
>>> print numerix.allequal((12, 13),
...                        numerix.nonzero(mesh.facesBottom & (x < 1))[0])
1
```

facesFront
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.)
>>> print numerix.allequal((0, 1, 2, 3, 4, 5),
...                        numerix.nonzero(mesh.facesFront)[0])
True
```

facesLeft

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.)
>>> print numerix.allequal((21, 25),
...                        numerix.nonzero(mesh.facesLeft)[0])
True
>>> mesh = Grid2D(nx = 3, ny = 2, dx = 0.5, dy = 2.)
>>> print numerix.allequal((9, 13),
...                        numerix.nonzero(mesh.facesLeft)[0])
True
```

facesRight

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.)
>>> print numerix.allequal((24, 28),
...                        numerix.nonzero(mesh.facesRight)[0])
True
>>> mesh = Grid2D(nx = 3, ny = 2, dx = 0.5, dy = 2.)
>>> print numerix.allequal((12, 16),
...                        numerix.nonzero(mesh.facesRight)[0])
True
```

facesTop

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.)
>>> print numerix.allequal((18, 19, 20),
...                        numerix.nonzero(mesh.facesTop)[0])
True
>>> mesh = Grid2D(nx = 3, ny = 2, dx = 0.5, dy = 2.)
>>> print numerix.allequal((6, 7, 8),
...                        numerix.nonzero(mesh.facesTop)[0])
True
```

facesUp

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.)
>>> print numerix.allequal((18, 19, 20),
...                        numerix.nonzero(mesh.facesTop)[0])
True
>>> mesh = Grid2D(nx = 3, ny = 2, dx = 0.5, dy = 2.)
>>> print numerix.allequal((6, 7, 8),
...                        numerix.nonzero(mesh.facesTop)[0])
True
```

getCellCenters(**args, **kwds*)

Deprecated since version 3.0: use the `cellCenters` property instead

getCellVolumes(**args*, ***kwds*)
Deprecated since version 3.0: use the `cellVolumes` property instead

getDim(**args*, ***kwds*)
Deprecated since version 3.0: use the `dim` property instead

getExteriorFaces(**args*, ***kwds*)
Deprecated since version 3.0: use the `exteriorFaces` property instead Return only the faces that have one neighboring cell.

getFaceCellIDs(**args*, ***kwds*)
Deprecated since version 3.0: use the `faceCellIDs` property instead

getFaceCenters(**args*, ***kwds*)
Deprecated since version 3.0: use the `faceCenters` property instead

getFacesBack(**args*, ***kwds*)
Deprecated since version 3.0: use the `facesBack` property instead

getFacesBottom(**args*, ***kwds*)
Deprecated since version 3.0: use the `facesBottom` property instead

getFacesDown(**args*, ***kwds*)
Deprecated since version 3.0: use the `facesBottom` property instead

getFacesFront(**args*, ***kwds*)
Deprecated since version 3.0: use the `facesFront` property instead

getFacesLeft(**args*, ***kwds*)
Deprecated since version 3.0: use the `facesLeft` property instead

getFacesRight(**args*, ***kwds*)
Deprecated since version 3.0: use the `facesRight` property instead

getFacesTop(**args*, ***kwds*)
Deprecated since version 3.0: use the `facesTop` property instead

getFacesUp(**args*, ***kwds*)
Deprecated since version 3.0: use the `facesTop` property instead

getInteriorFaceCellIDs(**args*, ***kwds*)
Deprecated since version 3.0: use the `interiorFaceCellIDs` property instead

getInteriorFaceIDs(**args*, ***kwds*)
Deprecated since version 3.0: use the `interiorFaceIDs` property instead

getInteriorFaces(**args*, ***kwds*)
Deprecated since version 3.0: use the `interiorFaces` property instead Return only the faces that have two neighboring cells.

getNearestCell(*point*)

getNumberOfCells(**args*, ***kwds*)
Deprecated since version 3.0: use the `numberOfCells` property instead

getPhysicalShape(**args*, ***kwds*)
Deprecated since version 3.0: use the `physicalShape` property instead

getScale(**args*, ***kwds*)
Deprecated since version 3.0: use the `scale` property instead

getShape(**args*, ***kwds*)
Deprecated since version 3.0: use the `shape` property instead

getVertexCoords(**args*, ***kwds*)
Deprecated since version 3.0: use the `vertexCoords` property instead

interiorFaceCellIDs

interiorFaceIDs

interiorFaces

scale

scaledCellDistances

scaledCellToCellDistances

scaledCellVolumes

scaledFaceAreas

scaledFaceToCellDistances

setScale(**args*, ***kwds*)
Deprecated since version 3.0: use the `scale` property instead

x

Equivalent to using `cellCenters[0]`.

```
>>> from fipy import *
>>> print Grid1D(nx=2).x
[ 0.5  1.5]
```

y

Equivalent to using `cellCenters[1]`.

```
>>> from fipy import *
>>> print Grid2D(nx=2, ny=2).y
[ 0.5  0.5  1.5  1.5]
>>> print Grid1D(nx=2).y
Traceback (most recent call last):
  ...
AttributeError: 1D meshes do not have a "y" attribute.
```

z

Equivalent to using `cellCenters[2]`.

```
>>> from fipy import *
>>> print Grid3D(nx=2, ny=2, nz=2).z
[ 0.5  0.5  0.5  0.5  1.5  1.5  1.5  1.5]
>>> print Grid2D(nx=2, ny=2).z
Traceback (most recent call last):
  ...
AttributeError: 1D and 2D meshes do not have a "z" attribute.
```

21.3 `cylindricalGrid1D` Module

1D Mesh

class fipy.meshes.cylindricalGrid1D.**CylindricalGrid1D**(*dx=1.0*, *nx=None*, *origin=(0,)*, *overlap=2*, *communicator=ParallelCommWrapper()*, **args*, ***kwargs*)

Bases: fipy.meshes.grid1D.Grid1D

Creates a 1D cylindrical grid mesh.

```
>>> mesh = CylindricalGrid1D(nx = 3)
>>> print mesh.cellCenters
[[ 0.5  1.5  2.5]]

>>> mesh = CylindricalGrid1D(dx = (1, 2, 3))
>>> print mesh.cellCenters
[[ 0.5  2.   4.5]]

>>> print numerix.allclose(mesh.cellVolumes, (0.5, 4., 13.5))
True

>>> mesh = CylindricalGrid1D(nx = 2, dx = (1, 2, 3))
Traceback (most recent call last):
...
IndexError: nx != len(dx)

>>> mesh = CylindricalGrid1D(nx=2, dx=(1., 2.)) + ((1.,),)
>>> print mesh.cellCenters
[[ 1.5  3. ]]
>>> print numerix.allclose(mesh.cellVolumes, (1.5, 6))
True
```

21.4 cylindricalGrid2D Module

2D rectangular Mesh

class fipy.meshes.cylindricalGrid2D.**CylindricalGrid2D**(*dx=1.0*, *dy=1.0*, *nx=None*, *ny=None*, *origin=((0.0,), (0.0,))*, *overlap=2*, *communicator=ParallelCommWrapper()*, **args*, ***kwargs*)

Bases: fipy.meshes.grid2D.Grid2D

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

cellCenters

cellVolumes

faceCenters

21.5 cylindricalUniformGrid1D Module

1D Mesh

class `fipy.meshes.cylindricalUniformGrid1D.`**`CylindricalUniformGrid1D`**(*dx=1.0, nx=1, origin=(0,), overlap=2, communicator=ParallelCommWrapper(), *args, **kwargs*)

Bases: `fipy.meshes.uniformGrid1D.UniformGrid1D`

Creates a 1D cylindrical grid mesh.

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

`cellVolumes`

21.6 `cylindricalUniformGrid2D` Module

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

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

Bases: `fipy.meshes.uniformGrid2D.UniformGrid2D`

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

`cellVolumes`

21.7 `factoryMeshes` Module

`fipy.meshes.factoryMeshes.`**`Grid3D`**(*dx=1.0, dy=1.0, dz=1.0, nx=None, ny=None, nz=None, Lx=None, Ly=None, Lz=None, overlap=2, communicator=ParallelCommWrapper()*)

Factory function to select between UniformGrid3D and Grid3D. If *Lx* is specified the length of the domain is always *Lx* regardless of *dx*.

Parameters

- *dx*: grid spacing in the horizontal direction
- *dy*: grid spacing in the vertical direction
- *dz*: grid spacing in the z-direction
- *nx*: number of cells in the horizontal direction
- *ny*: number of cells in the vertical direction

- *nz*: number of cells in the z-direction
- *Lx*: the domain length in the horizontal direction
- *Ly*: the domain length in the vertical direction
- *Lz*: the domain length in the z-direction
- *overlap*: the number of overlapping cells for parallel simulations. Generally 2 is adequate. Higher order equations or discretizations require more.
- *communicator*: either *fipy.tools.parallel* or *fipy.tools.serial*. Select *fipy.tools.serial* to create a serial mesh when running in parallel. Mostly used for test purposes.

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

Factory function to select between UniformGrid2D and Grid2D. If *Lx* is specified the length of the domain is always *Lx* regardless of *dx*.

Parameters

- *dx*: grid spacing in the horizontal direction
- *dy*: grid spacing in the vertical direction
- *nx*: number of cells in the horizontal direction
- *ny*: number of cells in the vertical direction
- *Lx*: the domain length in the horizontal direction
- *Ly*: the domain length in the vertical direction
- *overlap*: the number of overlapping cells for parallel simulations. Generally 2 is adequate. Higher order equations or discretizations require more.
- *communicator*: either *fipy.tools.parallel* or *fipy.tools.serial*. Select *fipy.tools.serial* to create a serial mesh when running in parallel. Mostly used for test purposes.

```
>>> print Grid2D(Lx=3., nx=2).dx
1.5
```

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

Factory function to select between UniformGrid1D and Grid1D. If *Lx* is specified the length of the domain is always *Lx* regardless of *dx*.

Parameters

- *dx*: grid spacing in the horizonal direction
- *nx*: number of cells in the horizonal direction
- *Lx*: the domain length in the horizonal direction
- *overlap*: the number of overlapping cells for parallel simulations. Generally 2 is adequate. Higher order equations or discretizations require more.
- *communicator*: either *fipy.tools.parallel* or *fipy.tools.serial*. Select *fipy.tools.serial* to create a serial mesh when running in parallel. Mostly used for test purposes.

fipy.meshes.factoryMeshes.**CylindricalGrid2D**(*dr=None*, *dz=None*, *nr=None*, *nz=None*, *Lr=None*, *Lz=None*, *dx=1.0*, *dy=1.0*, *nx=None*, *ny=None*, *Lx=None*, *Ly=None*, *origin=((0,), (0,))*, *overlap=2*, *communicator=ParallelCommWrapper()*)

Factory function to select between CylindricalUniformGrid2D and CylindricalGrid2D. If *Lx* is specified the length of the domain is always *Lx* regardless of *dx*.

Parameters

- *dr* or *dx*: grid spacing in the radial direction
- *dz* or *dy*: grid spacing in the vertical direction
- *nr* or *nx*: number of cells in the radial direction
- *nz* or *ny*: number of cells in the vertical direction
- *Lr* or *Lx*: the domain length in the radial direction
- *Lz* or *Ly*: the domain length in the vertical direction
- *origin* : position of the mesh's origin in the form ((x,),(y,))
- *overlap*: the number of overlapping cells for parallel simulations. Generally 2 is adequate. Higher order equations or discretizations require more.
- *communicator*: either *fipy.tools.parallel* or *fipy.tools.serial*. Select *fipy.tools.serial* to create a serial mesh when running in parallel. Mostly used for test purposes.

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

Factory function to select between CylindricalUniformGrid1D and CylindricalGrid1D. If *Lx* is specified the length of the domain is always *Lx* regardless of *dx*.

Parameters

- *dr* or *dx*: grid spacing in the radial direction
- *nr* or *nx*: number of cells in the radial direction
- *Lr* or *Lx*: the domain length in the radial direction
- *origin* : position of the mesh's origin in the form (x,)
- *overlap*: the number of overlapping cells for parallel simulations. Generally 2 is adequate. Higher order equations or discretizations require more.
- *communicator*: either *fipy.tools.parallel* or *fipy.tools.serial*. Select *fipy.tools.serial* to create a serial mesh when running in parallel. Mostly used for test purposes.

21.8 `gmshMesh` Module

`fipy.meshes.gmshMesh.`**`openMSHFile`**`(`*name*, *dimensions=None*, *coordDimensions=None*, *communicator=ParallelCommWrapper()*, *order=1*, *mode='r'*, *background=None*`)`

Open a Gmsh MSH file

Parameters

- *filename*: a string indicating gmsh output file
- *dimensions*: an integer indicating dimension of mesh
- *coordDimensions*: an integer indicating dimension of shapes
- *order*: ???

- *mode*: a string beginning with 'r' for reading and 'w' for writing. The file will be created if it doesn't exist when opened for writing; it will be truncated when opened for writing. Add a 'b' to the mode for binary files.
- *background*: a *CellVariable* that specifies the desired characteristic lengths of the mesh cells

`fipy.meshes.gmshMesh.`**`openPOSFile`**(*name*, *communicator=ParallelCommWrapper()*, *mode='w'*)

Open a Gmsh POS post-processing file

class `fipy.meshes.gmshMesh.`**`Gmsh2D`**(*arg*, *coordDimensions=2*, *communicator=ParallelCommWrapper()*, *order=1*, *background=None*)

Bases: `fipy.meshes.mesh2D.Mesh2D`

Construct a 2D Mesh using Gmsh

```
>>> radius = 5.
>>> side = 4.
>>> squaredCircle = Gmsh2D('''
... // A mesh consisting of a square inside a circle inside a circle
...
... // define the basic dimensions of the mesh
...
... cellSize = 1;
... radius = %(radius)g;
... side = %(side)g;
...
... // define the compass points of the inner circle
...
... Point(1) = {0, 0, 0, cellSize};
... Point(2) = {-radius, 0, 0, cellSize};
... Point(3) = {0, radius, 0, cellSize};
... Point(4) = {radius, 0, 0, cellSize};
... Point(5) = {0, -radius, 0, cellSize};
...
... // define the compass points of the outer circle
...
... Point(6) = {-2*radius, 0, 0, cellSize};
... Point(7) = {0, 2*radius, 0, cellSize};
... Point(8) = {2*radius, 0, 0, cellSize};
... Point(9) = {0, -2*radius, 0, cellSize};
...
... // define the corners of the square
...
... Point(10) = {side/2, side/2, 0, cellSize/2};
... Point(11) = {-side/2, side/2, 0, cellSize/2};
... Point(12) = {-side/2, -side/2, 0, cellSize/2};
... Point(13) = {side/2, -side/2, 0, cellSize/2};
...
... // define the inner circle
...
... Circle(1) = {2, 1, 3};
... Circle(2) = {3, 1, 4};
... Circle(3) = {4, 1, 5};
... Circle(4) = {5, 1, 2};
...
... // define the outer circle
...
... Circle(5) = {6, 1, 7};
... Circle(6) = {7, 1, 8};
... Circle(7) = {8, 1, 9};
```

```
... Circle(8) = {9, 1, 6};
...
... // define the square
...
... Line(9) = {10, 13};
... Line(10) = {13, 12};
... Line(11) = {12, 11};
... Line(12) = {11, 10};
...
... // define the three boundaries
...
... Line Loop(1) = {1, 2, 3, 4};
... Line Loop(2) = {5, 6, 7, 8};
... Line Loop(3) = {9, 10, 11, 12};
...
... // define the three domains
...
... Plane Surface(1) = {2, 1};
... Plane Surface(2) = {1, 3};
... Plane Surface(3) = {3};
...
... // label the three domains
...
... // attention: if you use any "Physical" labels, you *must* label
... // all elements that correspond to FiPy Cells (Physical Surace in 2D
... // and Physical Volume in 3D) or Gmsh will not include them and FiPy
... // will not be able to include them in the Mesh.
...
... // note: if you do not use any labels, all Cells will be included.
...
... Physical Surface("Outer") = {1};
... Physical Surface("Middle") = {2};
... Physical Surface("Inner") = {3};
...
... // label the "north-west" part of the exterior boundary
...
... // note: you only need to label the Face elements
... // (Physical Line in 2D and Physical Surface in 3D) that correspond
... // to boundaries you are interested in. FiPy does not need them to
... // construct the Mesh.
...
... Physical Line("NW") = {5};
... ''' % locals())
```

It can be easier to specify certain domains and boundaries within Gmsh than it is to define the same domains and boundaries with FiPy expressions.

Here we compare obtaining the same Cells and Faces using FiPy's parametric descriptions and Gmsh's labels.

```
>>> x, y = squaredCircle.cellCenters

>>> middle = ((x**2 + y**2 <= radius**2)
...           & ~((x > -side/2) & (x < side/2)
...               & (y > -side/2) & (y < side/2)))

>>> print (middle == squaredCircle.physicalCells["Middle"]).all()
True
```

```
>>> X, Y = squaredCircle.faceCenters

>>> NW = ((X**2 + Y**2 > (1.99*radius)**2)
...       & (X**2 + Y**2 < (2.01*radius)**2)
...       & (X <= 0) & (Y >= 0))

>>> print (NW == squaredCircle.physicalFaces["NW"]).all()
True
```

It is possible to direct Gmsh to give the mesh different densities in different locations

```
>>> geo = '''
... // A mesh consisting of a square
...
... // define the corners of the square
...
... Point(1) = {1, 1, 0, 1};
... Point(2) = {0, 1, 0, 1};
... Point(3) = {0, 0, 0, 1};
... Point(4) = {1, 0, 0, 1};
...
... // define the square
...
... Line(1) = {1, 2};
... Line(2) = {2, 3};
... Line(3) = {3, 4};
... Line(4) = {4, 1};
...
... // define the boundary
...
... Line Loop(1) = {1, 2, 3, 4};
...
... // define the domain
...
... Plane Surface(1) = {1};
... '''

>>> from fipy import CellVariable, numerix

>>> std = []
>>> bkg = None
>>> for refine in range(4):
...     square = Gmsh2D(geo, background=bkg)
...     x, y = square.cellCenters
...     bkg = CellVariable(mesh=square, value=abs(x / 4) + 0.01)
...     std.append(numerix.std(numerix.sqrt(2 * square.cellVolumes) / bkg))
```

Check that the mesh is monotonically approaching the desired density

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

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

```
>>> print std[-1] < 0.2
True
```

The initial mesh doesn't have to be from Gmsh

```
>>> from fipy import Tri2D

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

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

>>> print std1 > std2
True
```

Parameters

- ***arg*: a string giving (i) the path to an MSH file, (ii) a path to a** Gmsh geometry (".geo") file, or (iii) a Gmsh geometry script
- *coordDimensions*: an integer indicating dimension of shapes
- *order*: ???
- *background*: a *CellVariable* that specifies the desired characteristic lengths of the mesh cells

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

Bases: `fipy.meshes.gmshMesh.Gmsh2D`

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

Parameters

- ***arg*: a string giving (i) the path to an MSH file, (ii) a path to a** Gmsh geometry (".geo") file, or (iii) a Gmsh geometry script
- *coordDimensions*: an integer indicating dimension of shapes
- *order*: ???
- *background*: a *CellVariable* that specifies the desired characteristic lengths of the mesh cells

class `fipy.meshes.gmshMesh.`**`Gmsh3D`**(*arg*, *communicator=ParallelCommWrapper()*, *order=1*, *background=None*)

Bases: `fipy.meshes.mesh.Mesh`

Create a 3D Mesh using Gmsh

Parameters

- ***arg*: a string giving (i) the path to an MSH file, (ii) a path to a** Gmsh geometry (".geo") file, or (iii) a Gmsh geometry script
- *order*: ???
- *background*: a *CellVariable* that specifies the desired characteristic lengths of the mesh cells

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

Bases: `fipy.meshes.gmshMesh.Gmsh2D`

Should serve as a drop-in replacement for Grid2D.

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

Bases: fipy.meshes.gmshMesh.Gmsh3D

Should serve as a drop-in replacement for Grid3D.

class fipy.meshes.gmshMesh.**GmshImporter2D**(*arg*, *coordDimensions=2*)

Bases: fipy.meshes.gmshMesh.Gmsh2D

class fipy.meshes.gmshMesh.**GmshImporter2DIn3DSpace**(*arg*)

Bases: fipy.meshes.gmshMesh.Gmsh2DIn3DSpace

class fipy.meshes.gmshMesh.**GmshImporter3D**(*arg*)

Bases: fipy.meshes.gmshMesh.Gmsh3D

21.9 grid1D Module

1D Mesh

class fipy.meshes.grid1D.**Grid1D**(*dx=1.0*, *nx=None*, *overlap=2*, *communicator=ParallelCommWrapper()*, *_BuilderClass=<class 'fipy.meshes.builders.grid1DBuilder._NonuniformGrid1DBuilder'>*, *_RepresentationClass=<class 'fipy.meshes.representations.gridRepresentation._Grid1DRe*, *_TopologyClass=<class 'fipy.meshes.topologies.gridTopology._Grid1DTopology'>*)

Bases: fipy.meshes.mesh1D.Mesh1D

Creates a 1D grid mesh.

```
>>> mesh = Grid1D(nx = 3)
>>> print mesh.cellCenters
[[ 0.5  1.5  2.5]]

>>> mesh = Grid1D(dx = (1, 2, 3))
>>> print mesh.cellCenters
[[ 0.5  2.   4.5]]

>>> mesh = Grid1D(nx = 2, dx = (1, 2, 3))
Traceback (most recent call last):
...
IndexError: nx != len(dx)
```

21.10 grid2D Module

2D rectangular Mesh

class fipy.meshes.grid2D.**Grid2D**(*dx=1.0*, *dy=1.0*, *nx=None*, *ny=None*, *overlap=2*, *communicator=ParallelCommWrapper()*, *_RepresentationClass=<class 'fipy.meshes.representations.gridRepresentation._Grid2DRepresentation'>*, *_TopologyClass=<class 'fipy.meshes.topologies.gridTopology._Grid2DTopology'>*)

Bases: fipy.meshes.mesh2D.Mesh2D

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

21.11 `grid3D` Module

class `fipy.meshes.grid3D.`**`Grid3D`**(*dx=1.0, dy=1.0, dz=1.0, nx=None, ny=None, nz=None, overlap=2, communicator=ParallelCommWrapper(), _RepresentationClass=<class 'fipy.meshes.representations.gridRepresentation._Grid3DRepresentation'>, _TopologyClass=<class 'fipy.meshes.topologies.gridTopology._Grid3DTopology'>*)

Bases: `fipy.meshes.mesh.Mesh`

3D rectangular-prism Mesh

X axis runs from left to right. Y axis runs from bottom to top. Z axis runs from front to back.

Numbering System:

Vertices: Numbered in the usual way. X coordinate changes most quickly, then Y, then Z.

Cells: Same numbering system as vertices.

Faces: XY faces numbered first, then XZ faces, then YZ faces. Within each subcategory, it is numbered in the usual way.

21.12 `mesh` Module

exception `fipy.meshes.mesh.`**`MeshAdditionError`**

Bases: `exceptions.Exception`

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

Bases: `fipy.meshes.abstractMesh.AbstractMesh`

Generic mesh class using numerix to do the calculations

Meshes contain cells, faces, and vertices.

This is built for a non-mixed element mesh.

21.13 `mesh1D` Module

Generic mesh class using numerix to do the calculations

Meshes contain cells, faces, and vertices.

This is built for a non-mixed element mesh.

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

Bases: `fipy.meshes.mesh.Mesh`

21.14 `mesh2D` Module

Generic mesh class using numerix to do the calculations

Meshes contain cells, faces, and vertices.

This is built for a non-mixed element mesh.

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

Bases: `fipy.meshes.mesh.Mesh`

`extrude`(*extrudeFunc=<function <lambda> at 0x107c3a6e0>*, *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).cellCenters
[[ 0.5  1.5  0.5  1.5  0.5  1.5  0.5  1.5]
 [ 0.5  0.5  1.5  1.5  0.5  0.5  1.5  1.5]
 [ 0.5  0.5  0.5  0.5  1.5  1.5  1.5  1.5]]
```

```
>>> from fipy.meshes.tri2D import Tri2D
>>> print Tri2D().extrude(layers=2).cellCenters
[[ 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       ]]
```

21.15 `periodicGrid1D` Module

Peridoic 1D Mesh

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

Bases: `fipy.meshes.grid1D.Grid1D`

Creates a Periodic grid mesh.

```
>>> mesh = PeriodicGrid1D(dx = (1, 2, 3))
```

```
>>> print numerix.allclose(numerix.nonzero(mesh.exteriorFaces)[0],
...                        [3])
True
```

```
>>> print numerix.allclose(mesh.faceCellIDs.filled(-999),
...                        [[2, 0, 1, 2],
...                         [0, 1, 2, -999]])
True
```

```
>>> print numerix.allclose(mesh._cellDistances,
...                        [ 2., 1.5, 2.5, 1.5])
True

>>> print numerix.allclose(mesh._cellToCellDistances,
...                        [[ 2.,   1.5,   2.5],
...                         [ 1.5,   2.5,   2. ]])
True

>>> print numerix.allclose(mesh._faceNormals,
...                        [[ 1.,   1.,   1.,   1.]])
True

>>> print numerix.allclose(mesh._cellVertexIDs,
...                        [[1, 2, 2],
...                        [0, 1, 0]])
True
```

cellCenters

Defined outside of a geometry class since we need the *CellVariable* version of *cellCenters*; that is, the *cellCenters* defined in fipy.meshes.mesh and not in any geometry (since a *CellVariable* requires a reference to a mesh).

21.16 `periodicGrid2D` Module

2D periodic rectangular Mesh

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

Bases: `fipy.meshes.periodicGrid2D._BasePeriodicGrid2D`

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

>>> mesh = PeriodicGrid2D(dx = 1., dy = 0.5, nx = 2, ny = 2)

>>> print numerix.allclose(numerix.nonzero(mesh.exteriorFaces)[0],
...                        [ 4,  5,  8, 11])
True

>>> print numerix.allclose(mesh.faceCellIDs.filled(-1),
...                        [[2, 3, 0, 1, 2, 3, 1, 0, 1, 3, 2, 3],
...                         [0, 1, 2, 3, -1, -1, 0, 1, -1, 2, 3, -1]])
True

>>> print numerix.allclose(mesh._cellDistances,
...                        [ 0.5, 0.5, 0.5, 0.5, 0.25, 0.25, 1., 1., 0.5, 1., 1., 0.5])
True

>>> print numerix.allclose(mesh.cellFaceIDs,
...                        [[0, 1, 2, 3],
...                         [7, 6, 10, 9],
...                         [2, 3, 0, 1],
```

```
...                            [6, 7, 9, 10]])
True

>>> print numerix.allclose(mesh._cellToCellDistances,
...                           [[ 0.5, 0.5, 0.5, 0.5],
...                            [ 1., 1., 1., 1. ],
...                            [ 0.5, 0.5, 0.5, 0.5],
...                            [ 1., 1., 1., 1. ]])
True

>>> normals = [[0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1],
...             [1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0]]

>>> print numerix.allclose(mesh._faceNormals, normals)
True

>>> print numerix.allclose(mesh._cellVertexIDs,
...                           [[4, 5, 7, 8],
...                            [3, 4, 6, 7],
...                            [1, 2, 4, 5],
...                            [0, 1, 3, 4]])
True
```

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

Bases: `fipy.meshes.periodicGrid2D._BasePeriodicGrid2D`

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

Bases: `fipy.meshes.periodicGrid2D._BasePeriodicGrid2D`

21.17 `skewedGrid2D` Module

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

Bases: `fipy.meshes.mesh2D.Mesh2D`

Creates a 2D grid mesh with horizontal faces numbered first and then vertical faces. The points are skewed by a random amount (between *rand* and *-rand*) in the X and Y directions.

`physicalShape`
Return physical dimensions of Grid2D.

`shape`

21.18 `test` Module

Test implementation of the mesh

21.19 `tri2D` Module

class `fipy.meshes.tri2D.`**`Tri2D`**(*dx=1.0, dy=1.0, nx=1, ny=1, _RepresentationClass=<class 'fipy.meshes.representations.gridRepresentation._Grid2DRepresentation'>, _TopologyClass=<class 'fipy.meshes.topologies.meshTopology._Mesh2DTopology'>*)

Bases: `fipy.meshes.mesh2D.Mesh2D`

This class creates a mesh made out of triangles. It does this by starting with a standard Cartesian mesh (*Grid2D*) and dividing each cell in that mesh (hereafter referred to as a 'box') into four equal parts with the dividing lines being the diagonals.

Creates a 2D triangular mesh with horizontal faces numbered first then vertical faces, then diagonal faces. Vertices are numbered starting with the vertices at the corners of boxes and then the vertices at the centers of boxes. Cells on the right of boxes are numbered first, then cells on the top of boxes, then cells on the left of boxes, then cells on the bottom of boxes. Within each of the 'sub-categories' in the above, the vertices, cells and faces are numbered in the usual way.

Parameters

- *dx, dy*: The X and Y dimensions of each 'box'. If *dx* <> *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*.

`physicalShape`

Return physical dimensions of Grid2D.

`shape`

21.20 `uniformGrid` Module

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

Bases: `fipy.meshes.abstractMesh.AbstractMesh`

Wrapped scaled geometry properties

21.21 `uniformGrid1D` Module

1D Mesh

class `fipy.meshes.uniformGrid1D.`**`UniformGrid1D`**(*dx=1.0, nx=1, origin=(0,), overlap=2, communicator=ParallelCommWrapper(), _RepresentationClass=<class 'fipy.meshes.representations.gridRepresentation._Grid1DRepresentat _TopologyClass=<class 'fipy.meshes.topologies.gridTopology._Grid1DTopology'>*)

Bases: `fipy.meshes.uniformGrid.UniformGrid`

Creates a 1D grid mesh.

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

exteriorFaces
Geometry set and calc

faceCellIDs

vertexCoords

21.22 uniformGrid2D Module

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

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

Bases: fipy.meshes.uniformGrid.UniformGrid

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

faceCellIDs

faceVertexIDs

vertexCoords

21.23 uniformGrid3D Module

class fipy.meshes.uniformGrid3D.**UniformGrid3D**(*dx=1.0, dy=1.0, dz=1.0, nx=1, ny=1, nz=1, origin=[[0], [0], [0]], overlap=2, communicator=ParallelCommWrapper(), _RepresentationClass=<class 'fipy.meshes.representations.gridRepresentation._Grid3DRepresentation'>, _TopologyClass=<class 'fipy.meshes.topologies.gridTopology._Grid3DTopology'>*)

Bases: fipy.meshes.uniformGrid.UniformGrid

3D rectangular-prism Mesh with uniform grid spacing in each dimension.

X axis runs from left to right. Y axis runs from bottom to top. Z axis runs from front to back.

Numbering System:

Vertices: Numbered in the usual way. X coordinate changes most quickly, then Y, then Z.

*** arrays are arranged Z, Y, X because in numerix, the final index is the one that changes the most quickly ***

Cells: Same numbering system as vertices.

Faces: XY faces numbered first, then XZ faces, then YZ faces. Within each subcategory, it is numbered in the usual way.

faceCellIDs

faceVertexIDs

vertexCoords

21.24 Subpackages

21.24.1 builders Package

`builders` Package

`abstractGridBuilder` Module

`grid1DBuilder` Module

`grid2DBuilder` Module

`grid3DBuilder` Module

`periodicGrid1DBuilder` Module

`utilityClasses` Module

21.24.2 numMesh Package

`cylindricalGrid1D` Module

`cylindricalGrid2D` Module

`cylindricalUniformGrid1D` Module

`cylindricalUniformGrid2D` Module

`deprecatedWarning` Module

`fipy.meshes.numMesh.deprecatedWarning.`**`numMeshDeprecated`**`()`

`gmshImport` Module

`grid1D` Module

`grid2D` Module

`grid3D` Module

`periodicGrid1D` Module

`periodicGrid2D` Module

`skewedGrid2D` Module

`tri2D` Module

`uniformGrid1D` Module

`uniformGrid2D` Module

`uniformGrid3D` Module

21.24.3 representations Package

`abstractRepresentation` Module

`gridRepresentation` Module

`meshRepresentation` Module

21.24.4 topologies Package

`abstractTopology` Module

`gridTopology` Module

`meshTopology` Module

Chapter 22

models Package

22.1 `models` Package

class `fipy.models.`**`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 import Grid1D
>>> from fipy.tools import serial
>>> mesh = Grid1D(dx = .5, nx = 8, communicator=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, -0.25, 0.25, 0.75, 1.25, 1.75)
>>> print var.allclose(answer)
1
```

A 1D test case with very small dimensions.

```
>>> dx = 1e-10
>>> mesh = Grid1D(dx = dx, nx = 8, communicator=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 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.

$$X_{\text{min}} = \frac{\left|\vec{s} \times \vec{t}\right|}{\left|\vec{s} - \vec{t}\right|}$$

and in 3D,

$$X_{\text{min}} = \frac{1}{3!} \left|\vec{s} \cdot \left(\vec{t} \times \vec{u}\right)\right|$$

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.

cellInterfaceAreas

Returns the length of the interface that crosses the cell

A simple 1D test:

```
>>> from fipy.meshes import Grid1D
>>> mesh = Grid1D(dx = 1., nx = 4)
>>> distanceVariable = DistanceVariable(mesh = mesh,
...                                     value = (-1.5, -0.5, 0.5, 1.5))
>>> answer = CellVariable(mesh=mesh, value=(0, 0., 1., 0))
>>> print numerix.allclose(distanceVariable.cellInterfaceAreas,
...                        answer)
True
```

A 2D test case:

```
>>> from fipy.meshes import Grid2D
>>> from fipy.variables.cellVariable import CellVariable
>>> mesh = Grid2D(dx = 1., dy = 1., nx = 3, ny = 3)
>>> distanceVariable = DistanceVariable(mesh = mesh,
...                                     value = (1.5, 0.5, 1.5,
...                                              0.5,-0.5, 0.5,
...                                              1.5, 0.5, 1.5))
>>> answer = CellVariable(mesh=mesh,
...                       value=(0, 1, 0, 1, 0, 1, 0, 1, 0))
>>> print numerix.allclose(distanceVariable.cellInterfaceAreas, answer)
True
```

Another 2D test case:

```
>>> mesh = Grid2D(dx = .5, dy = .5, nx = 2, ny = 2)
>>> from fipy.variables.cellVariable import CellVariable
>>> distanceVariable = DistanceVariable(mesh = mesh,
...                                     value = (-0.5, 0.5, 0.5, 1.5))
>>> answer = CellVariable(mesh=mesh,
...                       value=(0, numerix.sqrt(2) / 4,  numerix.sqrt(2) / 4, 0))
>>> print numerix.allclose(distanceVariable.cellInterfaceAreas,
...                        answer)
True
```

Test to check that the circumfrence of a circle is, in fact, $2\pi r$.

```
>>> mesh = Grid2D(dx = 0.05, dy = 0.05, nx = 20, ny = 20)
>>> r = 0.25
>>> x, y = mesh.cellCenters
>>> rad = numerix.sqrt((x - .5)**2 + (y - .5)**2) - r
>>> distanceVariable = DistanceVariable(mesh = mesh, value = rad)
>>> print numerix.allclose(distanceVariable.cellInterfaceAreas.sum(), 1.57984690073)
1
```

extendVariable(*extensionVariable*, *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(**args*, ***kwds*)

Deprecated since version 3.0: use the `cellInterfaceAreas` property instead

class fipy.models.**SurfactantVariable**(*value=0.0*, *distanceVar=None*, *name='surfactant variable'*, *hasOld=False*)

Bases: `fipy.variables.cellVariable.CellVariable`

The *SurfactantVariable* maintains a conserved volumetric concentration on cells adjacent to, but in front of, the interface. The *value* argument corresponds to the initial concentration of surfactant on the interface (moles divided by area). The value held by the *SurfactantVariable* is actually a volume density (moles divided by volume).

A simple 1D test:

```
>>> from fipy.meshes import Grid1D
>>> mesh = Grid1D(dx = 1., nx = 4)
>>> from fipy.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 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(**args*, ***kwds*)
Deprecated since version 3.0: use the `interfaceVar` property instead

interfaceVar
Returns the *SurfactantVariable* rendered as an *_InterfaceSurfactantVariable* which evaluates the surfactant concentration as an area concentration the interface rather than a volumetric concentration.

class fipy.models.**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=None*, *dt=None*)
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=None*, *boundaryConditions=()*, *dt=None*, *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

class fipy.models.**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,

$$\dot{\theta} = J v \theta + k c (1 - \theta - \theta_{\text{other}}) - \theta c_{\text{other}} k_{\text{other}} - k^{-} \theta$$

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}})‘ 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 import Grid2D
>>> dx = .5
>>> dy = 2.3
>>> dt = 0.25
>>> k = 0.56
>>> initialValue = 0.1
>>> c = 0.2
```

```
>>> from fipy.meshes 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.interfaceVar,
...                  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 import Grid2D
>>> dx = 0.5
>>> dy = 2.73
>>> dt = 0.001
>>> k0 = 1.
>>> k1 = 10.
>>> theta0 = 0.
>>> theta1 = 0.
>>> c0 = 1.
>>> c1 = 1.
>>> totalSteps = 10
>>> 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, theta1, 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.interfaceVar,
...                        numerix.array((0, 0, answer0, 0, 0)), rtol = 1e-2)
1
>>> print numerix.allclose(var1.interfaceVar,
...                        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.cellCenters
>>> check = var0.interfaceVar + var1.interfaceVar
>>> answer = CellVariable(mesh=mesh, value=check)
>>> answer[x==1.25] = 1.
>>> print check.allequal(answer)
True
```

The following test case is to fix a bug where setting the adosrbtion 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.interfaceVar)
>>> answer[x==1.25] = 0.

>>> print var0.interfaceVar.allclose(answer)
True
```

The following test case is to fix a bug that allows the accelerator to become negative.

```
>>> nx = 5
>>> ny = 5
>>> dx = 1.
>>> dy = 1.
>>> mesh = Grid2D(dx=dx, dy=dy, nx = nx, ny = ny)
>>> x, y = mesh.cellCenters

>>> 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.interfaceVar)

>>> from fipy.models.levelSet.advection.higherOrderAdvectionEquation \
...     import buildHigherOrderAdvectionEquation
>>> advEq = buildHigherOrderAdvectionEquation(advectionCoeff = extVar)

>>> dt = 0.1

>>> for i in range(50):
...     disVar.calcDistanceFunction()
...     extVar.value = (numerix.array(accVar.interfaceVar))
...     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=None*, *dt=None*)

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=None*, *boundaryConditions=()*, *dt=None*, *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

fipy.models.**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

$$D \hat{n} \cdot \nabla c = -kc(1 - \theta) \qquad \text{at } \phi = 0.$$

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.

class fipy.models.**MayaviSurfactantViewer**(*distanceVar*, *surfactantVar=None*, *levelSetValue=0.0*, *title=None*, *smooth=0*, *zoomFactor=1.0*, *animate=False*, *limits={}*, ***kwlimits*)

Bases: `fipy.viewers.viewer.AbstractViewer`

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.cellCenters

>>> var.setValue(1, where=(x - Lx / 2.)**2 + (y - Ly / 2.)**2 < (Lx / 4.)**2)
>>> var.calcDistanceFunction()
>>> viewer = MayaviSurfactantViewer(var, smooth = 2)
>>> viewer.plot()
>>> viewer._promptForOpinion()
>>> del viewer

>>> var = DistanceVariable(mesh = mesh, value = -1)

>>> var.setValue(1, where=(y > 2. * Ly / 3.) | ((x > Lx / 2.) & (y > Ly / 3.)) | ((y < Ly / 6.)
>>> 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*)

class fipy.models.**MatplotlibSurfactantViewer**(*distanceVar*, *surfactantVar=None*, *levelSetValue=0.0*, *title=None*, *smooth=0*, *zoomFactor=1.0*, *animate=False*, *limits={}*, ***kwlimits*)

Bases: fipy.viewers.matplotlibViewer.matplotlibViewer.AbstractMatplotlibViewer

The *MatplotlibSurfactantViewer* creates a viewer with the Matplotlib python plotting package that displays a *DistanceVariable*.

Create a *MatplotlibSurfactantViewer*.

```
>>> from fipy import *
>>> m = Grid2D(nx=100, ny=100)
>>> x, y = m.cellCenters
>>> v = CellVariable(mesh=m, value=x**2 + y**2 - 10**2)
>>> s = CellVariable(mesh=m, value=sin(x / 10) * cos(y / 30))
>>> viewer = MatplotlibSurfactantViewer(distanceVar=v, surfactantVar=s)
>>> for r in range(1,200):
...     v.setValue(x**2 + y**2 - r**2)
...     viewer.plot()
```

```
>>> from fipy import *
>>> dx = 1.
>>> dy = 1.
>>> nx = 11
>>> ny = 11
>>> Lx = ny * dy
>>> Ly = nx * dx
>>> mesh = Grid2D(dx = dx, dy = dy, nx = nx, ny = ny)
>>> # from fipy.models.levelSet.distanceFunction.distanceVariable import DistanceVariable
>>> var = DistanceVariable(mesh = mesh, value = -1)

>>> x, y = mesh.cellCenters

>>> var.setValue(1, where=(x - Lx / 2.)**2 + (y - Ly / 2.)**2 < (Lx / 4.)**2)
>>> var.calcDistanceFunction()
>>> viewer = MatplotlibSurfactantViewer(var, smooth = 2)
>>> viewer.plot()
>>> viewer._promptForOpinion()
>>> del viewer

>>> var = DistanceVariable(mesh = mesh, value = -1)

>>> var.setValue(1, where=(y > 2. * Ly / 3.) | ((x > Lx / 2.) & (y > Ly / 3.)) | ((y < Ly / 6.)
>>> 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)

`fipy.models.`**`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.

`fipy.models.`**`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*

`fipy.models.`**`buildMetalIonDiffusionEquation`**(*ionVar=None*, *distanceVar=None*, *depositionRate=1*, *transientCoeff=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,

$$D \hat{n} \cdot \nabla c = \frac{v(c)}{\Omega} \qquad \text{at } phi = 0$$

where

$$v(c) = cV_0$$

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 import Grid1D
>>> nx = 11
>>> dx = 1.
>>> from fipy.tools import serial
>>> mesh = Grid1D(nx = nx, dx = dx, communicator=serial)
>>> x, = mesh.cellCenters
>>> 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.

>>> eqn = buildMetalIonDiffusionEquation(ionVar = ionVar,
...                                      distanceVar = disVar,
...                                      depositionRate = v * ionVar,
...                                      diffusionCoeff = diffusion,
...                                      metalIonMolarVolume = omega)

>>> ionVar.constrain(cinf, mesh.facesRight)

>>> for i in range(10):
...     eqn.solve(ionVar, dt = 1000)

>>> 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.

class `fipy.models.`**`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 then 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.numberOfCells - len(numerix.nonzero(mesh.electrolyteMask)[0])
150

>>> from fipy.variables.cellVariable import CellVariable
>>> var = CellVariable(mesh = mesh, value = 0.)

>>> from fipy.terms.diffusionTerm import DiffusionTerm
>>> eq = DiffusionTerm()

>>> var.constrain(0., mesh.facesBottom)
>>> var.constrain(domainHeight, mesh.facesTop)

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

Evaluate the result:

```
>>> centers = mesh.cellCenters[1].copy()
```

Note: the copy makes the array contiguous for inlining

```
>>> localErrors = (centers - var)**2 / centers**2
>>> globalError = numerix.sqrt(numerix.sum(localErrors) / mesh.numberOfCells)
>>> 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.

electrolyteMask

getElectrolyteMask(**args*, ***kwds*)
Deprecated since version 3.0: use the `electrolyteMask` property instead

22.2 `test` Module

22.3 Subpackages

22.3.1 levelSet Package

`levelSet` Package

class fipy.models.levelSet.**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 import Grid1D
>>> from fipy.tools import serial
>>> mesh = Grid1D(dx = .5, nx = 8, communicator=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, -0.25, 0.25, 0.75, 1.25, 1.75)
>>> print var.allclose(answer)
1
```

A 1D test case with very small dimensions.

```
>>> dx = 1e-10
>>> mesh = Grid1D(dx = dx, nx = 8, communicator=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 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.

$$X_{\text{min}} = \frac{\left|\vec{s} \times \vec{t}\right|}{\left|\vec{s} - \vec{t}\right|}$$

and in 3D,

$$X_{\text{min}} = \frac{1}{3!}\left|\vec{s} \cdot \left(\vec{t} \times \vec{u}\right)\right|$$

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.

cellInterfaceAreas

Returns the length of the interface that crosses the cell

A simple 1D test:

```
>>> from fipy.meshes import Grid1D
>>> mesh = Grid1D(dx = 1., nx = 4)
>>> distanceVariable = DistanceVariable(mesh = mesh,
...                                      value = (-1.5, -0.5, 0.5, 1.5))
>>> answer = CellVariable(mesh=mesh, value=(0, 0., 1., 0))
>>> print numerix.allclose(distanceVariable.cellInterfaceAreas,
...                        answer)
True
```

A 2D test case:

```
>>> from fipy.meshes import Grid2D
>>> from fipy.variables.cellVariable import CellVariable
>>> mesh = Grid2D(dx = 1., dy = 1., nx = 3, ny = 3)
>>> distanceVariable = DistanceVariable(mesh = mesh,
...                                     value = (1.5, 0.5, 1.5,
...                                              0.5,-0.5, 0.5,
...                                              1.5, 0.5, 1.5))
>>> answer = CellVariable(mesh=mesh,
...                       value=(0, 1, 0, 1, 0, 1, 0, 1, 0))
>>> print numerix.allclose(distanceVariable.cellInterfaceAreas, answer)
True
```

Another 2D test case:

```
>>> mesh = Grid2D(dx = .5, dy = .5, nx = 2, ny = 2)
>>> from fipy.variables.cellVariable import CellVariable
>>> distanceVariable = DistanceVariable(mesh = mesh,
...                                     value = (-0.5, 0.5, 0.5, 1.5))
>>> answer = CellVariable(mesh=mesh,
...                       value=(0, numerix.sqrt(2) / 4,  numerix.sqrt(2) / 4, 0))
>>> print numerix.allclose(distanceVariable.cellInterfaceAreas,
...                        answer)
True
```

Test to check that the circumfrence of a circle is, in fact, $2\pi r$.

```
>>> mesh = Grid2D(dx = 0.05, dy = 0.05, nx = 20, ny = 20)
>>> r = 0.25
>>> x, y = mesh.cellCenters
>>> rad = numerix.sqrt((x - .5)**2 + (y - .5)**2) - r
>>> distanceVariable = DistanceVariable(mesh = mesh, value = rad)
>>> print numerix.allclose(distanceVariable.cellInterfaceAreas.sum(), 1.57984690073)
1
```

extendVariable(*extensionVariable*, *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(**args*, ***kwds*)

Deprecated since version 3.0: use the `cellInterfaceAreas` property instead

class fipy.models.levelSet.**SurfactantVariable**(*value=0.0*, *distanceVar=None*, *name='surfactant variable'*, *hasOld=False*)

Bases: `fipy.variables.cellVariable.CellVariable`

The *SurfactantVariable* maintains a conserved volumetric concentration on cells adjacent to, but in front of, the interface. The *value* argument corresponds to the initial concentration of surfactant on the interface (moles divided by area). The value held by the *SurfactantVariable* is actually a volume density (moles divided by volume).

A simple 1D test:

```
>>> from fipy.meshes import Grid1D
>>> mesh = Grid1D(dx = 1., nx = 4)
```

```
>>> from fipy.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 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(**args*, ***kwds*)
Deprecated since version 3.0: use the `interfaceVar` property instead

interfaceVar
Returns the *SurfactantVariable* rendered as an *_InterfaceSurfactantVariable* which evaluates the surfactant concentration as an area concentration the interface rather than a volumetric concentration.

class fipy.models.levelSet.**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=None*, *dt=None*)
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=None*, *boundaryConditions=()*, *dt=None*, *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

class fipy.models.levelSet.**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}})‘ 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 import Grid2D
>>> dx = .5
>>> dy = 2.3
>>> dt = 0.25
```

```
>>> k = 0.56
>>> initialValue = 0.1
>>> c = 0.2

>>> from fipy.meshes 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.interfaceVar,
...                   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 import Grid2D
>>> dx = 0.5
>>> dy = 2.73
>>> dt = 0.001
>>> k0 = 1.
>>> k1 = 10.
>>> theta0 = 0.
>>> theta1 = 0.
>>> c0 = 1.
>>> c1 = 1.
>>> totalSteps = 10
>>> 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, theta1, 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.interfaceVar,
...                     numerix.array((0, 0, answer0, 0, 0)), rtol = 1e-2)
1
>>> print numerix.allclose(var1.interfaceVar,
...                     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.cellCenters
>>> check = var0.interfaceVar + var1.interfaceVar
>>> answer = CellVariable(mesh=mesh, value=check)
>>> answer[x==1.25] = 1.
>>> print check.allequal(answer)
True
```

The following test case is to fix a bug where setting the adosrbtion 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.interfaceVar)
>>> answer[x==1.25] = 0.

>>> print var0.interfaceVar.allclose(answer)
True
```

The following test case is to fix a bug that allows the accelerator to become negative.

```
>>> nx = 5
>>> ny = 5
>>> dx = 1.
>>> dy = 1.
>>> mesh = Grid2D(dx=dx, dy=dy, nx = nx, ny = ny)
>>> x, y = mesh.cellCenters
```

```
>>> 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.interfaceVar)

>>> from fipy.models.levelSet.advection.higherOrderAdvectionEquation \
...     import buildHigherOrderAdvectionEquation
>>> advEq = buildHigherOrderAdvectionEquation(advectionCoeff = extVar)

>>> dt = 0.1

>>> for i in range(50):
...     disVar.calcDistanceFunction()
...     extVar.value = (numerix.array(accVar.interfaceVar))
...     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=None*, *dt=None*)

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=None*, *boundaryConditions=()*, *dt=None*, *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

fipy.models.levelSet.**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

$$D \hat{n} \cdot \nabla c = -kc(1 - \theta) \qquad \text{at } \phi = 0.$$

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.

class fipy.models.levelSet.**MayaviSurfactantViewer**(*distanceVar*, *surfactantVar=None*, *levelSetValue=0.0*, *title=None*, *smooth=0*, *zoomFactor=1.0*, *animate=False*, *limits={}*, ***kwlimits*)

Bases: fipy.viewers.viewer.AbstractViewer

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.cellCenters

>>> var.setValue(1, where=(x - Lx / 2.)**2 + (y - Ly / 2.)**2 < (Lx / 4.)**2)
>>> var.calcDistanceFunction()
>>> viewer = MayaviSurfactantViewer(var, smooth = 2)
>>> viewer.plot()
>>> viewer._promptForOpinion()
>>> del viewer

>>> var = DistanceVariable(mesh = mesh, value = -1)

>>> var.setValue(1, where=(y > 2. * Ly / 3.) | ((x > Lx / 2.) & (y > Ly / 3.)) | ((y < Ly / 6.)
>>> 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*)

class fipy.models.levelSet.**MatplotlibSurfactantViewer** (*distanceVar, surfactantVar=None, levelSetValue=0.0, title=None, smooth=0, zoomFactor=1.0, animate=False, limits={}, **kwlimits*)

Bases: fipy.viewers.matplotlibViewer.matplotlibViewer.AbstractMatplotlibViewer

The *MatplotlibSurfactantViewer* creates a viewer with the Matplotlib python plotting package that displays a *DistanceVariable*.

Create a *MatplotlibSurfactantViewer*.

```
>>> from fipy import *
>>> m = Grid2D(nx=100, ny=100)
>>> x, y = m.cellCenters
>>> v = CellVariable(mesh=m, value=x**2 + y**2 - 10**2)
>>> s = CellVariable(mesh=m, value=sin(x / 10) * cos(y / 30))
>>> viewer = MatplotlibSurfactantViewer(distanceVar=v, surfactantVar=s)
>>> for r in range(1,200):
...     v.setValue(x**2 + y**2 - r**2)
...     viewer.plot()
```

```
>>> from fipy import *
>>> dx = 1.
>>> dy = 1.
>>> nx = 11
>>> ny = 11
>>> Lx = ny * dy
>>> Ly = nx * dx
>>> mesh = Grid2D(dx = dx, dy = dy, nx = nx, ny = ny)
>>> # from fipy.models.levelSet.distanceFunction.distanceVariable import DistanceVariable
>>> var = DistanceVariable(mesh = mesh, value = -1)

>>> x, y = mesh.cellCenters

>>> var.setValue(1, where=(x - Lx / 2.)**2 + (y - Ly / 2.)**2 < (Lx / 4.)**2)
>>> var.calcDistanceFunction()
>>> viewer = MatplotlibSurfactantViewer(var, smooth = 2)
>>> viewer.plot()
>>> viewer._promptForOpinion()
>>> del viewer

>>> var = DistanceVariable(mesh = mesh, value = -1)

>>> var.setValue(1, where=(y > 2. * Ly / 3.) | ((x > Lx / 2.) & (y > Ly / 3.)) | ((y < Ly / 6.)
>>> 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)

`fipy.models.levelSet.`**`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.

`fipy.models.levelSet.`**`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*

`fipy.models.levelSet.`**`buildMetalIonDiffusionEquation`**(*ionVar=None*, *distanceVar=None*, *depositionRate=1*, *transientCoeff=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,

$$D \hat{n} \cdot \nabla c = \frac{v(c)}{\Omega} \qquad \text{at } phi = 0$$

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 import Grid1D
>>> nx = 11
>>> dx = 1.
>>> from fipy.tools import serial
>>> mesh = Grid1D(nx = nx, dx = dx, communicator=serial)
>>> x, = mesh.cellCenters
>>> 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.

>>> eqn = buildMetalIonDiffusionEquation(ionVar = ionVar,
...                                      distanceVar = disVar,
...                                      depositionRate = v * ionVar,
...                                      diffusionCoeff = diffusion,
...                                      metalIonMolarVolume = omega)

>>> ionVar.constrain(cinf, mesh.facesRight)

>>> for i in range(10):
...     eqn.solve(ionVar, dt = 1000)

>>> 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.

class `fipy.models.levelSet.`**`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 then 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.numberOfCells - len(numerix.nonzero(mesh.electrolyteMask)[0])
150

>>> from fipy.variables.cellVariable import CellVariable
>>> var = CellVariable(mesh = mesh, value = 0.)

>>> from fipy.terms.diffusionTerm import DiffusionTerm
>>> eq = DiffusionTerm()

>>> var.constrain(0., mesh.facesBottom)
>>> var.constrain(domainHeight, mesh.facesTop)

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

Evaluate the result:

```
>>> centers = mesh.cellCenters[1].copy()
```

Note: the copy makes the array contiguous for inlining

```
>>> localErrors = (centers - var)**2 / centers**2
>>> globalError = numerix.sqrt(numerix.sum(localErrors) / mesh.numberOfCells)
>>> 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.

electrolyteMask

getElectrolyteMask(**args*, ***kwds*)
Deprecated since version 3.0: use the `electrolyteMask` property instead

test Module

Subpackages

advection Package

advection Package

fipy.models.levelSet.advection.**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.

fipy.models.levelSet.advection.**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*

`advectionEquation` Module

`fipy.models.levelSet.advection.advectionEquation.`**`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.

`advectionTerm` Module

`higherOrderAdvectionEquation` Module

`fipy.models.levelSet.advection.higherOrderAdvectionEquation.`**`buildHigherOrderAdvectionEquati`**

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*

`higherOrderAdvectionTerm` Module

distanceFunction Package

`distanceFunction` Package

class `fipy.models.levelSet.distanceFunction.`**`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 import Grid1D
>>> from fipy.tools import serial
>>> mesh = Grid1D(dx = .5, nx = 8, communicator=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, -0.25, 0.25, 0.75, 1.25, 1.75)
>>> print var.allclose(answer)
1
```

A 1D test case with very small dimensions.

```
>>> dx = 1e-10
>>> mesh = Grid1D(dx = dx, nx = 8, communicator=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 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.

$$X_{\text{min}} = \frac{\left|\vec{s} \times \vec{t}\right|}{\left|\vec{s} - \vec{t}\right|}$$

and in 3D,

$$X_{\text{min}} = \frac{1}{3!} \left|\vec{s} \cdot \left(\vec{t} \times \vec{u}\right)\right|$$

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.

cellInterfaceAreas
Returns the length of the interface that crosses the cell

A simple 1D test:

```
>>> from fipy.meshes import Grid1D
>>> mesh = Grid1D(dx = 1., nx = 4)
>>> distanceVariable = DistanceVariable(mesh = mesh,
...                                     value = (-1.5, -0.5, 0.5, 1.5))
>>> answer = CellVariable(mesh=mesh, value=(0, 0., 1., 0))
>>> print numerix.allclose(distanceVariable.cellInterfaceAreas,
...                        answer)
True
```

A 2D test case:

```
>>> from fipy.meshes import Grid2D
>>> from fipy.variables.cellVariable import CellVariable
>>> mesh = Grid2D(dx = 1., dy = 1., nx = 3, ny = 3)
>>> distanceVariable = DistanceVariable(mesh = mesh,
...                                     value = (1.5, 0.5, 1.5,
...                                              0.5,-0.5, 0.5,
...                                              1.5, 0.5, 1.5))
>>> answer = CellVariable(mesh=mesh,
...                       value=(0, 1, 0, 1, 0, 1, 0, 1, 0))
>>> print numerix.allclose(distanceVariable.cellInterfaceAreas, answer)
True
```

Another 2D test case:

```
>>> mesh = Grid2D(dx = .5, dy = .5, nx = 2, ny = 2)
>>> from fipy.variables.cellVariable import CellVariable
>>> distanceVariable = DistanceVariable(mesh = mesh,
...                                     value = (-0.5, 0.5, 0.5, 1.5))
>>> answer = CellVariable(mesh=mesh,
...                       value=(0, numerix.sqrt(2) / 4,  numerix.sqrt(2) / 4, 0))
>>> print numerix.allclose(distanceVariable.cellInterfaceAreas,
...                        answer)
True
```

Test to check that the circumfrence of a circle is, in fact, $2\pi r$.

```
>>> mesh = Grid2D(dx = 0.05, dy = 0.05, nx = 20, ny = 20)
>>> r = 0.25
>>> x, y = mesh.cellCenters
>>> rad = numerix.sqrt((x - .5)**2 + (y - .5)**2) - r
```

```
>>> distanceVariable = DistanceVariable(mesh = mesh, value = rad)
>>> print numerix.allclose(distanceVariable.cellInterfaceAreas.sum(), 1.57984690073)
1
```

extendVariable(*extensionVariable*, *deleteIslands=False*)
Takes a *cellVariable* and extends the variable from the zero to the region encapuslated 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(**args*, ***kwds*)
Deprecated since version 3.0: use the `cellInterfaceAreas` property instead

distanceVariable Module

class fipy.models.levelSet.distanceFunction.distanceVariable.**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 import Grid1D
>>> from fipy.tools import serial
>>> mesh = Grid1D(dx = .5, nx = 8, communicator=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, -0.25, 0.25, 0.75, 1.25, 1.75)
>>> print var.allclose(answer)
1
```

A 1D test case with very small dimensions.

```
>>> dx = 1e-10
>>> mesh = Grid1D(dx = dx, nx = 8, communicator=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 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.

$$X_{\text{min}} = \frac{\left| \vec{s} \times \vec{t} \right|}{\left| \vec{s} - \vec{t} \right|}$$

and in 3D,

$$X_{\text{min}} = \frac{1}{3!} \left| \vec{s} \cdot \left(\vec{t} \times \vec{u} \right) \right|$$

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.

cellInterfaceAreas

Returns the length of the interface that crosses the cell

A simple 1D test:

```
>>> from fipy.meshes import Grid1D
>>> mesh = Grid1D(dx = 1., nx = 4)
>>> distanceVariable = DistanceVariable(mesh = mesh,
...                                     value = (-1.5, -0.5, 0.5, 1.5))
>>> answer = CellVariable(mesh=mesh, value=(0, 0., 1., 0))
>>> print numerix.allclose(distanceVariable.cellInterfaceAreas,
...                        answer)
True
```

A 2D test case:

```
>>> from fipy.meshes import Grid2D
>>> from fipy.variables.cellVariable import CellVariable
>>> mesh = Grid2D(dx = 1., dy = 1., nx = 3, ny = 3)
>>> distanceVariable = DistanceVariable(mesh = mesh,
...                                     value = (1.5, 0.5, 1.5,
...                                              0.5,-0.5, 0.5,
...                                              1.5, 0.5, 1.5))
>>> answer = CellVariable(mesh=mesh,
...                       value=(0, 1, 0, 1, 0, 1, 0, 1, 0))
>>> print numerix.allclose(distanceVariable.cellInterfaceAreas, answer)
True
```

Another 2D test case:

```
>>> mesh = Grid2D(dx = .5, dy = .5, nx = 2, ny = 2)
>>> from fipy.variables.cellVariable import CellVariable
>>> distanceVariable = DistanceVariable(mesh = mesh,
...                                     value = (-0.5, 0.5, 0.5, 1.5))
>>> answer = CellVariable(mesh=mesh,
...                       value=(0, numerix.sqrt(2) / 4,  numerix.sqrt(2) / 4, 0))
>>> print numerix.allclose(distanceVariable.cellInterfaceAreas,
...                        answer)
True
```

Test to check that the circumfrence of a circle is, in fact, $2\pi r$.

```
>>> mesh = Grid2D(dx = 0.05, dy = 0.05, nx = 20, ny = 20)
>>> r = 0.25
>>> x, y = mesh.cellCenters
>>> rad = numerix.sqrt((x - .5)**2 + (y - .5)**2) - r
>>> distanceVariable = DistanceVariable(mesh = mesh, value = rad)
>>> print numerix.allclose(distanceVariable.cellInterfaceAreas.sum(), 1.57984690073)
1
```

extendVariable (*extensionVariable*, *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 (**args*, ***kwds*)

Deprecated since version 3.0: use the `cellInterfaceAreas` property instead

`levelSetDiffusionEquation` Module

`levelSetDiffusionVariable` Module

electroChem Package

`electroChem` Package

`fipy.models.levelSet.electroChem.`**`buildMetalIonDiffusionEquation`**(*ionVar=None*, *distanceVar=None*, *depositionRate=1*, *transientCoeff=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,

$$D = \begin{cases} D_c & \text{when } \phi > 0 \\ 0 & \text{when } \phi \le 0 \end{cases}$$

The velocity of the interface generally has a linear dependence on ion concentration. The following boundary condition applies at the zero level set,

$$D \hat{n} \cdot \nabla c = \frac{v(c)}{\Omega} \qquad \text{at } phi = 0$$

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 import Grid1D
>>> nx = 11
>>> dx = 1.
>>> from fipy.tools import serial
>>> mesh = Grid1D(nx = nx, dx = dx, communicator=serial)
>>> x, = mesh.cellCenters
>>> 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.

>>> eqn = buildMetalIonDiffusionEquation(ionVar = ionVar,
...                                      distanceVar = disVar,
...                                      depositionRate = v * ionVar,
...                                      diffusionCoeff = diffusion,
...                                      metalIonMolarVolume = omega)

>>> ionVar.constrain(cinf, mesh.facesRight)

>>> for i in range(10):
...     eqn.solve(ionVar, dt = 1000)

>>> 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.

class `fipy.models.levelSet.electroChem.`**`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 then 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.numberOfCells - len(numerix.nonzero(mesh.electrolyteMask)[0])
150

>>> from fipy.variables.cellVariable import CellVariable
>>> var = CellVariable(mesh = mesh, value = 0.)

>>> from fipy.terms.diffusionTerm import DiffusionTerm
>>> eq = DiffusionTerm()

>>> var.constrain(0., mesh.facesBottom)
>>> var.constrain(domainHeight, mesh.facesTop)

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

Evaluate the result:

```
>>> centers = mesh.cellCenters[1].copy()
```

Note: the copy makes the array contiguous for inlining

```
>>> localErrors = (centers - var)**2 / centers**2
>>> globalError = numerix.sqrt(numerix.sum(localErrors) / mesh.numberOfCells)
>>> 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.

electrolyteMask

getElectrolyteMask(**args*, ***kwds*)
Deprecated since version 3.0: use the `electrolyteMask` property instead

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 *Gmsh2D* object that forms a transition mesh from a fine to a course region. The third mesh is another *Grid2D* object that forms the boundary layer. This region consists of very large elements and is only used for the diffusion in the boundary layer.

class `fipy.models.levelSet.electroChem.gapFillMesh.`**`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 then 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.numberOfCells - len(numerix.nonzero(mesh.electrolyteMask)[0])
150

>>> from fipy.variables.cellVariable import CellVariable
>>> var = CellVariable(mesh = mesh, value = 0.)

>>> from fipy.terms.diffusionTerm import DiffusionTerm
>>> eq = DiffusionTerm()

>>> var.constrain(0., mesh.facesBottom)
>>> var.constrain(domainHeight, mesh.facesTop)

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

Evaluate the result:

```
>>> centers = mesh.cellCenters[1].copy()
```

Note: the copy makes the array contiguous for inlining

```
>>> localErrors = (centers - var)**2 / centers**2
>>> globalError = numerix.sqrt(numerix.sum(localErrors) / mesh.numberOfCells)
>>> 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.

electrolyteMask

getElectrolyteMask(**args*, ***kwds*)
Deprecated since version 3.0: use the `electrolyteMask` property instead

metalIonDiffusionEquation Module

fipy.models.levelSet.electroChem.metalIonDiffusionEquation.**buildMetalIonDiffusionEquation**(

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,

$$D\hat{n} \cdot \nabla c = \frac{v(c)}{\Omega} \qquad \text{at } phi = 0$$

where

$$v(c) = cV_0$$

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 import Grid1D
>>> nx = 11
>>> dx = 1.
>>> from fipy.tools import serial
>>> mesh = Grid1D(nx = nx, dx = dx, communicator=serial)
>>> x, = mesh.cellCenters
>>> 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.

>>> eqn = buildMetalIonDiffusionEquation(ionVar = ionVar,
...                                      distanceVar = disVar,
...                                      depositionRate = v * ionVar,
...                                      diffusionCoeff = diffusion,
...                                      metalIonMolarVolume = omega)

>>> ionVar.constrain(cinf, mesh.facesRight)

>>> for i in range(10):
...     eqn.solve(ionVar, dt = 1000)

>>> 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.

`metalIonSourceVariable` Module

`test` Module

surfactant Package

`surfactant` Package

class `fipy.models.levelSet.surfactant.`**`SurfactantVariable`**(*value=0.0*, *distanceVar=None*, *name='surfactant variable'*, *hasOld=False*)

Bases: `fipy.variables.cellVariable.CellVariable`

The *SurfactantVariable* maintains a conserved volumetric concentration on cells adjacent to, but in front of, the interface. The *value* argument corresponds to the initial concentration of surfactant on the interface (moles divided by area). The value held by the *SurfactantVariable* is actually a volume density (moles divided by volume).

A simple 1D test:

```
>>> from fipy.meshes import Grid1D
>>> mesh = Grid1D(dx = 1., nx = 4)
>>> from fipy.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 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(**args*, ***kwds*)
Deprecated since version 3.0: use the `interfaceVar` property instead

interfaceVar
Returns the *SurfactantVariable* rendered as an *_InterfaceSurfactantVariable* which evaluates the surfactant concentration as an area concentration the interface rather than a volumetric concentration.

class fipy.models.levelSet.surfactant.**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=None*, *dt=None*)
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=None*, *boundaryConditions=()*, *dt=None*, *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

class fipy.models.levelSet.surfactant.**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}})‘ 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 import Grid2D
>>> dx = .5
>>> dy = 2.3
>>> dt = 0.25
>>> k = 0.56
>>> initialValue = 0.1
>>> c = 0.2

>>> from fipy.meshes 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.interfaceVar,
...                  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 import Grid2D
>>> dx = 0.5
>>> dy = 2.73
>>> dt = 0.001
>>> k0 = 1.
>>> k1 = 10.
>>> theta0 = 0.
>>> theta1 = 0.
>>> c0 = 1.
>>> c1 = 1.
>>> totalSteps = 10
>>> 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, theta1, 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.interfaceVar,
...                  numerix.array((0, 0, answer0, 0, 0)), rtol = 1e-2)
1
>>> print numerix.allclose(var1.interfaceVar,
```

```
...                     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.cellCenters
>>> check = var0.interfaceVar + var1.interfaceVar
>>> answer = CellVariable(mesh=mesh, value=check)
>>> answer[x==1.25] = 1.
>>> print check.allequal(answer)
True
```

The following test case is to fix a bug where setting the adosrbtion 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.interfaceVar)
>>> answer[x==1.25] = 0.

>>> print var0.interfaceVar.allclose(answer)
True
```

The following test case is to fix a bug that allows the accelerator to become negative.

```
>>> nx = 5
>>> ny = 5
>>> dx = 1.
>>> dy = 1.
>>> mesh = Grid2D(dx=dx, dy=dy, nx = nx, ny = ny)
>>> x, y = mesh.cellCenters

>>> 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.interfaceVar)

>>> from fipy.models.levelSet.advection.higherOrderAdvectionEquation \
...     import buildHigherOrderAdvectionEquation
>>> advEq = buildHigherOrderAdvectionEquation(advectionCoeff = extVar)

>>> dt = 0.1

>>> for i in range(50):
...     disVar.calcDistanceFunction()
...     extVar.value = (numerix.array(accVar.interfaceVar))
...     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=None*, *dt=None*)
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=None*, *boundaryConditions=()*, *dt=None*, *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

`fipy.models.levelSet.surfactant.`**`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

$$D \hat{n} \cdot \nabla c = -kc(1 - \theta) \qquad \text{at } \phi = 0.$$

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.

class fipy.models.levelSet.surfactant.**MayaviSurfactantViewer**(*distanceVar*, *surfactantVar=None*, *levelSetValue=0.0*, *title=None*, *smooth=0*, *zoomFactor=1.0*, *animate=False*, *limits={}*, ***kwlimits*)

Bases: fipy.viewers.viewer.AbstractViewer

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.cellCenters

>>> var.setValue(1, where=(x - Lx / 2.)**2 + (y - Ly / 2.)**2 < (Lx / 4.)**2)
>>> var.calcDistanceFunction()
>>> viewer = MayaviSurfactantViewer(var, smooth = 2)
>>> viewer.plot()
>>> viewer._promptForOpinion()
>>> del viewer

>>> var = DistanceVariable(mesh = mesh, value = -1)

>>> var.setValue(1, where=(y > 2. * Ly / 3.) | ((x > Lx / 2.) & (y > Ly / 3.)) | ((y < Ly / 6.)
>>> 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*)

class fipy.models.levelSet.surfactant.**MatplotlibSurfactantViewer** (*distanceVar*, *surfactantVar=None*, *levelSetValue=0.0*, *title=None*, *smooth=0*, *zoomFactor=1.0*, *animate=False*, *limits={}*, ***kwlimits*)

Bases: fipy.viewers.matplotlibViewer.matplotlibViewer.AbstractMatplotlibViewer

The *MatplotlibSurfactantViewer* creates a viewer with the Matplotlib python plotting package that displays a *DistanceVariable*.

Create a *MatplotlibSurfactantViewer*.

```
>>> from fipy import *
>>> m = Grid2D(nx=100, ny=100)
>>> x, y = m.cellCenters
>>> v = CellVariable(mesh=m, value=x**2 + y**2 - 10**2)
>>> s = CellVariable(mesh=m, value=sin(x / 10) * cos(y / 30))
>>> viewer = MatplotlibSurfactantViewer(distanceVar=v, surfactantVar=s)
>>> for r in range(1,200):
...     v.setValue(x**2 + y**2 - r**2)
...     viewer.plot()
```

```
>>> from fipy import *
>>> dx = 1.
>>> dy = 1.
>>> nx = 11
>>> ny = 11
>>> Lx = ny * dy
>>> Ly = nx * dx
>>> mesh = Grid2D(dx = dx, dy = dy, nx = nx, ny = ny)
>>> # from fipy.models.levelSet.distanceFunction.distanceVariable import DistanceVariable
>>> var = DistanceVariable(mesh = mesh, value = -1)

>>> x, y = mesh.cellCenters

>>> var.setValue(1, where=(x - Lx / 2.)**2 + (y - Ly / 2.)**2 < (Lx / 4.)**2)
>>> var.calcDistanceFunction()
>>> viewer = MatplotlibSurfactantViewer(var, smooth = 2)
>>> viewer.plot()
>>> viewer._promptForOpinion()
>>> del viewer
```

```
>>> var = DistanceVariable(mesh = mesh, value = -1)

>>> var.setValue(1, where=(y > 2. * Ly / 3.) | ((x > Lx / 2.) & (y > Ly / 3.)) | ((y < Ly / 6.)
>>> 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)

`adsorbingSurfactantEquation` Module

class `fipy.models.levelSet.surfactant.adsorbingSurfactantEquation.`**`AdsorbingSurfactantEquatio`**

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}})‘ 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 import Grid2D
>>> dx = .5
>>> dy = 2.3
>>> dt = 0.25
>>> k = 0.56
>>> initialValue = 0.1
>>> c = 0.2

>>> from fipy.meshes 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.interfaceVar,
...                  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 import Grid2D
>>> dx = 0.5
>>> dy = 2.73
>>> dt = 0.001
>>> k0 = 1.
```

```
>>> k1 = 10.
>>> theta0 = 0.
>>> theta1 = 0.
>>> c0 = 1.
>>> c1 = 1.
>>> totalSteps = 10
>>> 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, theta1, 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.interfaceVar,
...                        numerix.array((0, 0, answer0, 0, 0)), rtol = 1e-2)
1
>>> print numerix.allclose(var1.interfaceVar,
...                        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.cellCenters
>>> check = var0.interfaceVar + var1.interfaceVar
>>> answer = CellVariable(mesh=mesh, value=check)
>>> answer[x==1.25] = 1.
>>> print check.allequal(answer)
True
```

The following test case is to fix a bug where setting the adosrbtion 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.interfaceVar)
>>> answer[x==1.25] = 0.

>>> print var0.interfaceVar.allclose(answer)
True
```

The following test case is to fix a bug that allows the accelerator to become negative.

```
>>> nx = 5
>>> ny = 5
>>> dx = 1.
>>> dy = 1.
>>> mesh = Grid2D(dx=dx, dy=dy, nx = nx, ny = ny)
>>> x, y = mesh.cellCenters

>>> 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.interfaceVar)

>>> from fipy.models.levelSet.advection.higherOrderAdvectionEquation \
...     import buildHigherOrderAdvectionEquation
>>> advEq = buildHigherOrderAdvectionEquation(advectionCoeff = extVar)

>>> dt = 0.1

>>> for i in range(50):
...     disVar.calcDistanceFunction()
...     extVar.value = (numerix.array(accVar.interfaceVar))
...     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=None*, *dt=None*)

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=None*, *boundaryConditions=()*, *dt=None*, *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

`convectionCoeff` Module

`lines` Module

`matplotlibSurfactantViewer` Module

class `fipy.models.levelSet.surfactant.matplotlibSurfactantViewer.`**`MatplotlibSurfactantViewer`**(*a*

*s f t V l e V t t s z F t a i n l i * i*

Bases: `fipy.viewers.matplotlibViewer.matplotlibViewer.AbstractMatplotlibViewer`

The *MatplotlibSurfactantViewer* creates a viewer with the Matplotlib python plotting package that displays a *DistanceVariable*.

Create a *MatplotlibSurfactantViewer*.

```
>>> from fipy import *
>>> m = Grid2D(nx=100, ny=100)
>>> x, y = m.cellCenters
>>> v = CellVariable(mesh=m, value=x**2 + y**2 - 10**2)
>>> s = CellVariable(mesh=m, value=sin(x / 10) * cos(y / 30))
>>> viewer = MatplotlibSurfactantViewer(distanceVar=v, surfactantVar=s)
>>> for r in range(1,200):
...     v.setValue(x**2 + y**2 - r**2)
...     viewer.plot()
```

```
>>> from fipy import *
>>> dx = 1.
>>> dy = 1.
>>> nx = 11
>>> ny = 11
>>> Lx = ny * dy
>>> Ly = nx * dx
>>> mesh = Grid2D(dx = dx, dy = dy, nx = nx, ny = ny)
>>> # from fipy.models.levelSet.distanceFunction.distanceVariable import DistanceVariable
>>> var = DistanceVariable(mesh = mesh, value = -1)

>>> x, y = mesh.cellCenters

>>> var.setValue(1, where=(x - Lx / 2.)**2 + (y - Ly / 2.)**2 < (Lx / 4.)**2)
>>> var.calcDistanceFunction()
>>> viewer = MatplotlibSurfactantViewer(var, smooth = 2)
>>> viewer.plot()
>>> viewer._promptForOpinion()
>>> del viewer
```

```
>>> var = DistanceVariable(mesh = mesh, value = -1)

>>> var.setValue(1, where=(y > 2. * Ly / 3.) | ((x > Lx / 2.) & (y > Ly / 3.)) | ((y < Ly / 6.)
>>> 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)

`mayaviSurfactantViewer` Module

class `fipy.models.levelSet.surfactant.mayaviSurfactantViewer.`**`MayaviSurfactantViewer`**(*distanceVar, surfactantVar=None, levelSetValue=0.0, title=None, smooth=0, zoomFactor=1.0, animate=False, limits={}, **kwlimits*)

Bases: `fipy.viewers.viewer.AbstractViewer`

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.cellCenters

>>> var.setValue(1, where=(x - Lx / 2.)**2 + (y - Ly / 2.)**2 < (Lx / 4.)**2)
>>> var.calcDistanceFunction()
>>> viewer = MayaviSurfactantViewer(var, smooth = 2)
>>> viewer.plot()
>>> viewer._promptForOpinion()
>>> del viewer

>>> var = DistanceVariable(mesh = mesh, value = -1)

>>> var.setValue(1, where=(y > 2. * Ly / 3.) | ((x > Lx / 2.) & (y > Ly / 3.)) | ((y < Ly / 6.)
>>> 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*)

surfactantBulkDiffusionEquation Module

fipy.models.levelSet.surfactant.surfactantBulkDiffusionEquation.**buildSurfactantBulkDiffusi**

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

$$D \hat{n} \cdot \nabla c = -kc(1 - \theta) \qquad \text{at } \phi = 0.$$

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.

`surfactantEquation` Module

class `fipy.models.levelSet.surfactant.surfactantEquation.`**`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=None*, *dt=None*)

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=None*, *boundaryConditions=()*, *dt=None*, *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

`surfactantVariable` Module

class `fipy.models.levelSet.surfactant.surfactantVariable.`**`SurfactantVariable`**(*value=0.0*, *distanceVar=None*, *name='surfactant variable'*, *hasOld=False*)

Bases: `fipy.variables.cellVariable.CellVariable`

The *SurfactantVariable* maintains a conserved volumetric concentration on cells adjacent to, but in front of, the interface. The *value* argument corresponds to the initial concentration of surfactant on the interface (moles divided by area). The value held by the *SurfactantVariable* is actually a volume density (moles divided by volume).

A simple 1D test:

```
>>> from fipy.meshes import Grid1D
>>> mesh = Grid1D(dx = 1., nx = 4)
>>> from fipy.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 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(**args*, ***kwds*)

Deprecated since version 3.0: use the `interfaceVar` property instead

interfaceVar

Returns the *SurfactantVariable* rendered as an *_InterfaceSurfactantVariable* which evaluates the surfactant concentration as an area concentration the interface rather than a volumetric concentration.

Chapter 23

solvers Package

23.1 `solvers` Package

exception fipy.solvers.**SolverConvergenceWarning**(*solver*, *iter*, *relres*)
Bases: exceptions.Warning

exception fipy.solvers.**MaximumIterationWarning**(*solver*, *iter*, *relres*)
Bases: fipy.solvers.solver.SolverConvergenceWarning

exception fipy.solvers.**PreconditionerWarning**(*solver*, *iter*, *relres*)
Bases: fipy.solvers.solver.SolverConvergenceWarning

exception fipy.solvers.**IllConditionedPreconditionerWarning**(*solver*, *iter*, *relres*)
Bases: fipy.solvers.solver.PreconditionerWarning

exception fipy.solvers.**PreconditionerNotPositiveDefiniteWarning**(*solver*, *iter*, *relres*)
Bases: fipy.solvers.solver.PreconditionerWarning

exception fipy.solvers.**MatrixIllConditionedWarning**(*solver*, *iter*, *relres*)
Bases: fipy.solvers.solver.SolverConvergenceWarning

exception fipy.solvers.**StagnatedSolverWarning**(*solver*, *iter*, *relres*)
Bases: fipy.solvers.solver.SolverConvergenceWarning

exception fipy.solvers.**ScalarQuantityOutOfRangeWarning**(*solver*, *iter*, *relres*)
Bases: fipy.solvers.solver.SolverConvergenceWarning

class fipy.solvers.**Solver**(*tolerance=1e-10*, *iterations=1000*, *precon=None*)
Bases: object

The base *LinearXSolver* class.

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

Create a *Solver* object.

Parameters

- *tolerance*: The required error tolerance.
- *iterations*: The maximum number of iterative steps to perform.
- *precon*: Preconditioner to use. This parameter is only available for Trilinos solvers.

fipy.solvers.**DefaultSolver**
alias of LinearPCGSolver

fipy.solvers.**DummySolver**
alias of LinearPCGSolver

fipy.solvers.**DefaultAsymmetricSolver**
alias of LinearLUSolver

fipy.solvers.**GeneralSolver**
alias of LinearLUSolver

class fipy.solvers.**LinearCGSSolver**(*precon=None*, **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.

Parameters

- *precon*: Preconditioner to use

class fipy.solvers.**LinearPCGSolver**(*precon=<fipy.solvers.pysparse.preconditioners.ssorPreconditioner.SsorPreconditioner instance at 0x107c78fc8>*, **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 by default. Alternatively, Jacobi preconditioning can be specified through *precon*. The PCG method solves systems with a symmetric positive definite coefficient matrix.

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

Parameters

- *precon*: Preconditioner to use

class fipy.solvers.**LinearGMRESSolver**(*precon=<fipy.solvers.pysparse.preconditioners.jacobiPreconditioner.JacobiPrecon instance at 0x107c831b8>*, **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.

Parameters

- *precon*: Preconditioner to use

class fipy.solvers.**LinearLUSolver**(*tolerance=1e-10*, *iterations=10*, *maxIterations=10*, *precon=None*)
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. For large systems a number of iterations is generally required.
- *precon*: not used but maintains a common interface.

class `fipy.solvers.`**`LinearJORSolver`**(*tolerance=1e-10*, *iterations=1000*, *relaxation=1.0*)
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.
- *relaxation*: The relaxation.

class `fipy.solvers.`**`JacobiPreconditioner`**
Bases: `fipy.solvers.pysparse.preconditioners.preconditioner.Preconditioner`

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

Create a *Preconditioner* object.

class `fipy.solvers.`**`SsorPreconditioner`**
Bases: `fipy.solvers.pysparse.preconditioners.preconditioner.Preconditioner`

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

Create a *Preconditioner* object.

23.2 `pysparseMatrixSolver` Module

23.3 `solver` Module

The iterative solvers may output warnings if the solution is considered unsatisfactory. If you are not interested in these warnings, you can invoke python with a warning filter such as:

```
$ python -Wignore::fipy.SolverConvergenceWarning myscript.py
```

If you are extremely concerned about your preconditioner for some reason, you can abort whenever it has problems with:

```
$ python -Werror::fipy.PreconditionerWarning myscript.py
```

exception `fipy.solvers.solver.`**`SolverConvergenceWarning`**(*solver*, *iter*, *relres*)
Bases: `exceptions.Warning`

exception `fipy.solvers.solver.`**`MaximumIterationWarning`**(*solver*, *iter*, *relres*)
Bases: `fipy.solvers.solver.SolverConvergenceWarning`

exception `fipy.solvers.solver.`**`PreconditionerWarning`**(*solver*, *iter*, *relres*)
Bases: `fipy.solvers.solver.SolverConvergenceWarning`

exception `fipy.solvers.solver.`**`IllConditionedPreconditionerWarning`**(*solver*, *iter*, *relres*)
Bases: `fipy.solvers.solver.PreconditionerWarning`

exception `fipy.solvers.solver.`**`PreconditionerNotPositiveDefiniteWarning`**(*solver*, *iter*, *relres*)
Bases: `fipy.solvers.solver.PreconditionerWarning`

exception fipy.solvers.solver.**MatrixIllConditionedWarning**(*solver*, *iter*, *relres*)
Bases: fipy.solvers.solver.SolverConvergenceWarning

exception fipy.solvers.solver.**StagnatedSolverWarning**(*solver*, *iter*, *relres*)
Bases: fipy.solvers.solver.SolverConvergenceWarning

exception fipy.solvers.solver.**ScalarQuantityOutOfRangeWarning**(*solver*, *iter*, *relres*)
Bases: fipy.solvers.solver.SolverConvergenceWarning

class fipy.solvers.solver.**Solver**(*tolerance=1e-10*, *iterations=1000*, *precon=None*)
Bases: object

The base *LinearXSolver* class.

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

Create a *Solver* object.

Parameters

- *tolerance*: The required error tolerance.
- *iterations*: The maximum number of iterative steps to perform.
- *precon*: Preconditioner to use. This parameter is only available for Trilinos solvers.

23.4 test Module

23.5 Subpackages

23.5.1 pyAMG Package

pyAMG Package

fipy.solvers.pyAMG.**DefaultSolver**
alias of LinearGMRESSolver

fipy.solvers.pyAMG.**DummySolver**
alias of LinearGMRESSolver

fipy.solvers.pyAMG.**DefaultAsymmetricSolver**
alias of LinearLUSolver

fipy.solvers.pyAMG.**GeneralSolver**
alias of LinearGeneralSolver

class fipy.solvers.pyAMG.**LinearGMRESSolver**(*tolerance=1e-15*, *iterations=2000*, *precon=<fipy.solvers.pyAMG.preconditioners.smoothedAggregationPrecondi instance at 0x110c66cb0>*)
Bases: fipy.solvers.scipy.linearGMRESSolver.LinearGMRESSolver

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

Parameters

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

- *precon*: Preconditioner to use.

class `fipy.solvers.pyAMG.`**`LinearCGSSolver`**(*tolerance=1e-15*, *iterations=2000*, *precon=<fipy.solvers.pyAMG.preconditioners.smoothedAggregationPreconditio instance at 0x110c34ef0>*)

Bases: `fipy.solvers.scipy.linearCGSSolver.LinearCGSSolver`

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

Parameters

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

class `fipy.solvers.pyAMG.`**`LinearPCGSolver`**(*tolerance=1e-15*, *iterations=2000*, *precon=<fipy.solvers.pyAMG.preconditioners.smoothedAggregationPreconditio instance at 0x110c344d0>*)

Bases: `fipy.solvers.scipy.linearPCGSolver.LinearPCGSolver`

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

Parameters

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

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

Bases: `fipy.solvers.scipy.scipySolver._ScipySolver`

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

Create a *Solver* object.

Parameters

- *tolerance*: The required error tolerance.
- *iterations*: The maximum number of iterative steps to perform.
- *precon*: Preconditioner to use. This parameter is only available for Trilinos solvers.

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

Bases: `fipy.solvers.scipy.scipySolver._ScipySolver`

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

Create a *Solver* object.

Parameters

- *tolerance*: The required error tolerance.
- *iterations*: The maximum number of iterative steps to perform.
- *precon*: Preconditioner to use. This parameter is only available for Trilinos solvers.

linearCGSSolver Module

class fipy.solvers.pyAMG.linearCGSSolver.**LinearCGSSolver**(*tolerance=1e-15*, *iterations=2000*, *precon=<fipy.solvers.pyAMG.preconditioners.smoothed... instance at 0x110c34ef0>*)

Bases: fipy.solvers.scipy.linearCGSSolver.LinearCGSSolver

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

Parameters

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

linearGMRESSolver Module

class fipy.solvers.pyAMG.linearGMRESSolver.**LinearGMRESSolver**(*tolerance=1e-15*, *iterations=2000*, *precon=<fipy.solvers.pyAMG.preconditioners.smo... instance at 0x110c66cb0>*)

Bases: fipy.solvers.scipy.linearGMRESSolver.LinearGMRESSolver

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

Parameters

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

linearGeneralSolver Module

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

Bases: fipy.solvers.scipy.scipySolver._ScipySolver

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

Create a *Solver* object.

Parameters

- *tolerance*: The required error tolerance.
- *iterations*: The maximum number of iterative steps to perform.
- *precon*: Preconditioner to use. This parameter is only available for Trilinos solvers.

`linearLUSolver` Module

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

Bases: `fipy.solvers.scipy.scipySolver._ScipySolver`

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

Create a *Solver* object.

Parameters

- *tolerance*: The required error tolerance.
- *iterations*: The maximum number of iterative steps to perform.
- *precon*: Preconditioner to use. This parameter is only available for Trilinos solvers.

`linearPCGSolver` Module

class `fipy.solvers.pyAMG.linearPCGSolver.`**`LinearPCGSolver`**(*tolerance=1e-15*, *iterations=2000*, *precon=<fipy.solvers.pyAMG.preconditioners.smoothedA instance at 0x110c344d0>*)

Bases: `fipy.solvers.scipy.linearPCGSolver.LinearPCGSolver`

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

Parameters

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

`smoothedAggregationSolver` Module

Subpackages

preconditioners Package

`preconditioners` **Package**

`smoothedAggregationPreconditioner` **Module**

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

23.5.2 pysparse Package

`pysparse` Package

`fipy.solvers.pysparse.`**`DefaultSolver`**

alias of `LinearPCGSolver`

fipy.solvers.pysparse.**DummySolver**
alias of LinearPCGSolver

fipy.solvers.pysparse.**DefaultAsymmetricSolver**
alias of LinearLUSolver

fipy.solvers.pysparse.**GeneralSolver**
alias of LinearLUSolver

class fipy.solvers.pysparse.**LinearCGSSolver**(*precon=None*, **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.

Parameters

- *precon*: Preconditioner to use

class fipy.solvers.pysparse.**LinearPCGSolver**(*precon=<fipy.solvers.pysparse.preconditioners.ssorPreconditioner.SsorP instance at 0x107c78fc8>*, **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 by default. Alternatively, Jacobi preconditioning can be specified through *precon*. The PCG method solves systems with a symmetric positive definite coefficient matrix.

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

Parameters

- *precon*: Preconditioner to use

class fipy.solvers.pysparse.**LinearGMRESSolver**(*precon=<fipy.solvers.pysparse.preconditioners.jacobiPreconditioner.J instance at 0x107c831b8>*, **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.

Parameters

- *precon*: Preconditioner to use

class fipy.solvers.pysparse.**LinearLUSolver**(*tolerance=1e-10*, *iterations=10*, *maxIterations=10*, *precon=None*)
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. For large systems a number of iterations is generally required.
- *precon*: not used but maintains a common interface.

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

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.
- *relaxation*: The relaxation.

class `fipy.solvers.pysparse.`**`JacobiPreconditioner`**

Bases: `fipy.solvers.pysparse.preconditioners.preconditioner.Preconditioner`

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

Create a *Preconditioner* object.

class `fipy.solvers.pysparse.`**`SsorPreconditioner`**

Bases: `fipy.solvers.pysparse.preconditioners.preconditioner.Preconditioner`

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

Create a *Preconditioner* object.

`linearCGSSolver` Module

class `fipy.solvers.pysparse.linearCGSSolver.`**`LinearCGSSolver`**(*precon=None*, **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.

Parameters

- *precon*: Preconditioner to use

`linearGMRESSolver` Module

class `fipy.solvers.pysparse.linearGMRESSolver.`**`LinearGMRESSolver`**(*precon=<fipy.solvers.pysparse.preconditio instance at 0x107c831b8>*, **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.

Parameters

- *precon*: Preconditioner to use

`linearJORSolver` Module

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

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.
- *relaxation*: The relaxation.

`linearLUSolver` Module

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

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. For large systems a number of iterations is generally required.
- *precon*: not used but maintains a common interface.

`linearPCGSolver` Module

class `fipy.solvers.pysparse.linearPCGSolver.`**`LinearPCGSolver`**(*precon=<fipy.solvers.pysparse.preconditioners.s instance at 0x107c78fc8>*, **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 by default. Alternatively, Jacobi preconditioning can be specified through *precon*. The PCG method solves systems with a symmetric positive definite coefficient matrix.

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

Parameters

- *precon*: Preconditioner to use

`pysparseSolver` Module

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

Bases: `fipy.solvers.pysparseMatrixSolver._PysparseMatrixSolver`

The base *pysparseSolver* class.

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

Subpackages

preconditioners Package

`preconditioners` Package

class `fipy.solvers.pysparse.preconditioners.`**`JacobiPreconditioner`**

Bases: `fipy.solvers.pysparse.preconditioners.preconditioner.Preconditioner`

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

Create a *Preconditioner* object.

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

Bases: `fipy.solvers.pysparse.preconditioners.preconditioner.Preconditioner`

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

Create a *Preconditioner* object.

`jacobiPreconditioner` Module

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

Bases: `fipy.solvers.pysparse.preconditioners.preconditioner.Preconditioner`

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

Create a *Preconditioner* object.

`preconditioner` Module

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

Base preconditioner class

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

Create a *Preconditioner* object.

ssorPreconditioner Module

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

Bases: fipy.solvers.pysparse.preconditioners.preconditioner.Preconditioner

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

Create a *Preconditioner* object.

23.5.3 scipy Package

scipy Package

fipy.solvers.scipy.**DefaultSolver**

alias of LinearLUSolver

fipy.solvers.scipy.**DummySolver**

alias of LinearGMRESSolver

fipy.solvers.scipy.**DefaultAsymmetricSolver**

alias of LinearLUSolver

fipy.solvers.scipy.**GeneralSolver**

alias of LinearLUSolver

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

Bases: fipy.solvers.scipy.scipyKrylovSolver._ScipyKrylovSolver

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

Parameters

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

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

Bases: fipy.solvers.scipy.scipyKrylovSolver._ScipyKrylovSolver

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

Parameters

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

class fipy.solvers.scipy.**LinearBicgstabSolver**

Bases: fipy.solvers.scipy.scipyKrylovSolver._ScipyKrylovSolver

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

Parameters

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

class `fipy.solvers.scipy.`**`LinearLUSolver`**(*tolerance=1e-10*, *iterations=1000*, *precon=None*)
Bases: `fipy.solvers.scipy.scipySolver._ScipySolver`

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

Create a *Solver* object.

Parameters

- *tolerance*: The required error tolerance.
- *iterations*: The maximum number of iterative steps to perform.
- *precon*: Preconditioner to use. This parameter is only available for Trilinos solvers.

class `fipy.solvers.scipy.`**`LinearPCGSolver`**(*tolerance=1e-15*, *iterations=2000*, *precon=None*)
Bases: `fipy.solvers.scipy.scipyKrylovSolver._ScipyKrylovSolver`

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

Parameters

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

`linearBicgstabSolver` Module

class `fipy.solvers.scipy.linearBicgstabSolver.`**`LinearBicgstabSolver`**
Bases: `fipy.solvers.scipy.scipyKrylovSolver._ScipyKrylovSolver`

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

Parameters

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

`linearCGSSolver` Module

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

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

Parameters

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

linearGMRESSolver Module

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

Bases: fipy.solvers.scipy.scipyKrylovSolver._ScipyKrylovSolver

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

Parameters

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

linearLUSolver Module

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

Bases: fipy.solvers.scipy.scipySolver._ScipySolver

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

Create a *Solver* object.

Parameters

- *tolerance*: The required error tolerance.
- *iterations*: The maximum number of iterative steps to perform.
- *precon*: Preconditioner to use. This parameter is only available for Trilinos solvers.

linearPCGSolver Module

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

Bases: fipy.solvers.scipy.scipyKrylovSolver._ScipyKrylovSolver

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

Parameters

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

`scipyKrylovSolver` Module

`scipySolver` Module

23.5.4 trilinos Package

`trilinos` Package

`fipy.solvers.trilinos.`**`DefaultSolver`**
alias of `LinearGMRESSolver`

`fipy.solvers.trilinos.`**`DummySolver`**
alias of `LinearGMRESSolver`

`fipy.solvers.trilinos.`**`DefaultAsymmetricSolver`**
alias of `LinearGMRESSolver`

`fipy.solvers.trilinos.`**`GeneralSolver`**
alias of `LinearGMRESSolver`

class `fipy.solvers.trilinos.`**`LinearCGSSolver`**(*tolerance=1e-10, iterations=1000, precon=<fipy.solvers.trilinos.preconditioners.multilevelDDPreconditioner.M instance at 0x110c36cf8>*)

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.
- *precon*: Preconditioner to use.

class `fipy.solvers.trilinos.`**`LinearPCGSolver`**(*tolerance=1e-10, iterations=1000, precon=<fipy.solvers.trilinos.preconditioners.multilevelDDPreconditioner.M instance at 0x110c1def0>*)

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.
- *precon*: Preconditioner to use.

class `fipy.solvers.trilinos.`**`LinearGMRESSolver`**(*tolerance=1e-10, iterations=1000, precon=<fipy.solvers.trilinos.preconditioners.multilevelDDPreconditione instance at 0x110c365a8>*)

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.
- *precon*: Preconditioner to use.

class fipy.solvers.trilinos.**LinearLUSolver**(*tolerance=1e-10*, *iterations=10*, *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.

class fipy.solvers.trilinos.**LinearBicgstabSolver**(*tolerance=1e-10*, *iterations=1000*, *precon=<fipy.solvers.trilinos.preconditioners.jacobiPreconditioner.J instance at 0x1109a3440>*)

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.
- *precon*: Preconditioner to use.

class fipy.solvers.trilinos.**TrilinosMLTest**(*tolerance=1e-10*, *iterations=5*, *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.
- *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.

class fipy.solvers.trilinos.**MultilevelDDPreconditioner**

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.

class fipy.solvers.trilinos.**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.

class `fipy.solvers.trilinos.`**`MultilevelDDMLPreconditioner`**
Bases: `fipy.solvers.trilinos.preconditioners.preconditioner.Preconditioner`

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

Create a *Preconditioner* object.

class `fipy.solvers.trilinos.`**`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.

class `fipy.solvers.trilinos.`**`JacobiPreconditioner`**
Bases: `fipy.solvers.trilinos.preconditioners.preconditioner.Preconditioner`

Jacobi Preconditioner for Trilinos solvers.

Create a *Preconditioner* object.

class `fipy.solvers.trilinos.`**`ICPreconditioner`**
Bases: `fipy.solvers.trilinos.preconditioners.preconditioner.Preconditioner`

Incomplete Cholesky Preconditioner from IFPACK for Trilinos Solvers.

Create a *Preconditioner* object.

class `fipy.solvers.trilinos.`**`DomDecompPreconditioner`**
Bases: `fipy.solvers.trilinos.preconditioners.preconditioner.Preconditioner`

Domain Decomposition preconditioner for Trilinos solvers.

Create a *Preconditioner* object.

class `fipy.solvers.trilinos.`**`MultilevelSGSPreconditioner`**(*levels=10*)
Bases: `fipy.solvers.trilinos.preconditioners.preconditioner.Preconditioner`

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

Initialize the multilevel preconditioner

- *levels*: Maximum number of levels

class `fipy.solvers.trilinos.`**`MultilevelSolverSmootherPreconditioner`**(*levels=10*)
Bases: `fipy.solvers.trilinos.preconditioners.preconditioner.Preconditioner`

Multilevel preconditioner for Trilinos solvers using Aztec solvers as smoothers.

Initialize the multilevel preconditioner

- *levels*: Maximum number of levels

`linearBicgstabSolver` Module

class `fipy.solvers.trilinos.linearBicgstabSolver.`**`LinearBicgstabSolver`**(*tolerance=1e-10*, *iterations=1000*, *precon=<fipy.solvers.trilinos.precond* *instance at 0x1109a3440>*)

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.
- *precon*: Preconditioner to use.

`linearCGSSolver` Module

class `fipy.solvers.trilinos.linearCGSSolver.`**`LinearCGSSolver`**(*tolerance=1e-10*, *iterations=1000*, *precon=<fipy.solvers.trilinos.preconditioners.multil* *instance at 0x110c36cf8>*)

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.
- *precon*: Preconditioner to use.

`linearGMRESSolver` Module

class `fipy.solvers.trilinos.linearGMRESSolver.`**`LinearGMRESSolver`**(*tolerance=1e-10*, *iterations=1000*, *precon=<fipy.solvers.trilinos.preconditioners.* *instance at 0x110c365a8>*)

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.
- *precon*: Preconditioner to use.

`linearLUSolver` Module

class `fipy.solvers.trilinos.linearLUSolver.`**`LinearLUSolver`**(*tolerance=1e-10*, *iterations=10*, *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.

`linearPCGSolver` Module

class `fipy.solvers.trilinos.linearPCGSolver.`**`LinearPCGSolver`**(*tolerance=1e-10*, *iterations=1000*, *precon=<fipy.solvers.trilinos.preconditioners.multil instance at 0x110c1def0>*)

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.
- *precon*: Preconditioner to use.

`trilinosAztecOOSolver` Module

class `fipy.solvers.trilinos.trilinosAztecOOSolver.`**`TrilinosAztecOOSolver`**(*tolerance=1e-10*, *iterations=1000*, *precon=<fipy.solvers.trilinos.prec instance at 0x110c36e60>*)

Bases: `fipy.solvers.trilinos.trilinosSolver.TrilinosSolver`

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

Parameters

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

`trilinosMLTest` Module

class `fipy.solvers.trilinos.trilinosMLTest.`**`TrilinosMLTest`**(*tolerance=1e-10*, *iterations=5*, *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.
- *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.

`trilinosNonlinearSolver` Module

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

Bases: `fipy.solvers.trilinos.trilinosSolver.TrilinosSolver`

`solve`(*dt=None*)

`trilinosSolver` Module

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

Bases: `fipy.solvers.solver.Solver`

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

Subpackages

preconditioners Package

`preconditioners` Package

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

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.

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

class `fipy.solvers.trilinos.preconditioners.`**`MultilevelDDMLPreconditioner`**
Bases: `fipy.solvers.trilinos.preconditioners.preconditioner.Preconditioner`

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

Create a *Preconditioner* object.

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

class `fipy.solvers.trilinos.preconditioners.`**`JacobiPreconditioner`**
Bases: `fipy.solvers.trilinos.preconditioners.preconditioner.Preconditioner`

Jacobi Preconditioner for Trilinos solvers.

Create a *Preconditioner* object.

class `fipy.solvers.trilinos.preconditioners.`**`ICPreconditioner`**
Bases: `fipy.solvers.trilinos.preconditioners.preconditioner.Preconditioner`

Incomplete Cholesky Preconditioner from IFPACK for Trilinos Solvers.

Create a *Preconditioner* object.

class `fipy.solvers.trilinos.preconditioners.`**`DomDecompPreconditioner`**
Bases: `fipy.solvers.trilinos.preconditioners.preconditioner.Preconditioner`

Domain Decomposition preconditioner for Trilinos solvers.

Create a *Preconditioner* object.

class `fipy.solvers.trilinos.preconditioners.`**`MultilevelSGSPreconditioner`**(*levels=10*)
Bases: `fipy.solvers.trilinos.preconditioners.preconditioner.Preconditioner`

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

Initialize the multilevel preconditioner

- *levels*: Maximum number of levels

class `fipy.solvers.trilinos.preconditioners.`**`MultilevelSolverSmootherPreconditioner`**(*levels=10*)
Bases: `fipy.solvers.trilinos.preconditioners.preconditioner.Preconditioner`

Multilevel preconditioner for Trilinos solvers using Aztec solvers as smoothers.

Initialize the multilevel preconditioner

- *levels*: Maximum number of levels

`domDecompPreconditioner` Module

class `fipy.solvers.trilinos.preconditioners.domDecompPreconditioner.`**`DomDecompPreconditioner`**

Bases: `fipy.solvers.trilinos.preconditioners.preconditioner.Preconditioner`

Domain Decomposition preconditioner for Trilinos solvers.

Create a *Preconditioner* object.

`icPreconditioner` Module

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

Bases: `fipy.solvers.trilinos.preconditioners.preconditioner.Preconditioner`

Incomplete Cholesky Preconditioner from IFPACK for Trilinos Solvers.

Create a *Preconditioner* object.

`jacobiPreconditioner` Module

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

Bases: `fipy.solvers.trilinos.preconditioners.preconditioner.Preconditioner`

Jacobi Preconditioner for Trilinos solvers.

Create a *Preconditioner* object.

`multilevelDDMLPreconditioner` Module

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

Bases: `fipy.solvers.trilinos.preconditioners.preconditioner.Preconditioner`

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

Create a *Preconditioner* object.

`multilevelDDPreconditioner` Module

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

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.

`multilevelNSSAPreconditioner` Module

class `fipy.solvers.trilinos.preconditioners.multilevelNSSAPreconditioner.`**`MultilevelNSSAPrecor`**

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.

`multilevelSAPreconditioner` Module

class `fipy.solvers.trilinos.preconditioners.multilevelSAPreconditioner.`**`MultilevelSAPreconditi`**

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.

`multilevelSGSPreconditioner` Module

class `fipy.solvers.trilinos.preconditioners.multilevelSGSPreconditioner.`**`MultilevelSGSPrecondi`**

Bases: `fipy.solvers.trilinos.preconditioners.preconditioner.Preconditioner`

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

Initialize the multilevel preconditioner

- *levels*: Maximum number of levels

`multilevelSolverSmootherPreconditioner` Module

class `fipy.solvers.trilinos.preconditioners.multilevelSolverSmootherPreconditioner.`**`Multilevel`**

Bases: `fipy.solvers.trilinos.preconditioners.preconditioner.Preconditioner`

Multilevel preconditioner for Trilinos solvers using Aztec solvers as smoothers.

Initialize the multilevel preconditioner

- *levels*: Maximum number of levels

`preconditioner` Module

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

The base Preconditioner class.

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

Create a *Preconditioner* object.

Chapter 24

steppers Package

24.1 `steppers` Package

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

Parameters

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

Returns

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

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

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

Parameters

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

Returns

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

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

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

Parameters

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

Returns

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

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

`fipy.steppers.`**`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

24.2 `pidStepper` Module

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

Bases: `fipy.steppers.stepper.Stepper`

Adaptive stepper using a PID controller, based on:

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

24.3 `pseudoRKQSStepper` Module

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

Bases: `fipy.steppers.stepper.Stepper`

Adaptive stepper based on the `rkqs` (Runge-Kutta "quality-controlled" stepper) algorithm of 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.

24.4 `stepper` Module

class `fipy.steppers.stepper.`**`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 25

terms Package

25.1 `terms` Package

exception `fipy.terms.`**`ExplicitVariableError`**(*s='Terms with explicit Variables cannot mix with Terms with implicit Variables.'*)
Bases: `exceptions.Exception`

exception `fipy.terms.`**`TermMultiplyError`**(*s='Must multiply terms by int or float.'*)
Bases: `exceptions.Exception`

exception `fipy.terms.`**`AbstractBaseClassError`**(*s="can't instantiate abstract base class"*)
Bases: `exceptions.NotImplementedError`

exception `fipy.terms.`**`VectorCoeffError`**(*s='The coefficient must be a vector value.'*)
Bases: `exceptions.TypeError`

exception `fipy.terms.`**`SolutionVariableNumberError`**(*s='Different number of solution variables and equations.'*)
Bases: `exceptions.Exception`

exception `fipy.terms.`**`SolutionVariableRequiredError`**(*s='The solution variable needs to be specified.'*)
Bases: `exceptions.Exception`

exception `fipy.terms.`**`IncorrectSolutionVariable`**(*s='The solution variable is incorrect.'*)
Bases: `exceptions.Exception`

`fipy.terms.`**`ConvectionTerm`**
alias of `PowerLawConvectionTerm`

class `fipy.terms.`**`TransientTerm`**(*coeff=1.0*, *var=None*)
Bases: `fipy.terms.cellTerm.CellTerm`

The *TransientTerm* represents

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

where ρ is the *coeff* value.

The following test case verifies that variable coefficients and old coefficient values work correctly. We will solve the following equation

$$\frac{\partial \phi^2}{\partial t} = k.$$

The analytic solution is given by

$$\phi = \sqrt{\phi_0^2 + kt},$$

where ϕ_0 is the initial value.

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

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

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

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

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

```
>>> 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
```

class `fipy.terms.`**`DiffusionTerm`**(*coeff=(1.0,), var=None*)

Bases: `fipy.terms.diffusionTermNoCorrection.DiffusionTermNoCorrection`

This term represents a higher order diffusion term. The order of the term is determined by the number of *coeffs*, such that:

```
DiffusionTerm(D1)
```

represents a typical 2nd-order diffusion term of the form

$$\nabla \cdot (D_1 \nabla \phi)$$

and:

```
DiffusionTerm((D1,D2))
```

represents a 4th-order Cahn-Hilliard term of the form

$$\nabla \cdot \{D_1 \nabla [\nabla \cdot (D_2 \nabla \phi)]\}$$

and so on.

class `fipy.terms.`**`DiffusionTermCorrection`**(*coeff=(1.0,), var=None*)

Bases: `fipy.terms.abstractDiffusionTerm._AbstractDiffusionTerm`

class `fipy.terms.`**`DiffusionTermNoCorrection`**(*coeff=(1.0,), var=None*)

Bases: `fipy.terms.abstractDiffusionTerm._AbstractDiffusionTerm`

class `fipy.terms.DiffusionTermCorrection` (*coeff=(1.0,), var=None*)
Bases: `fipy.terms.abstractDiffusionTerm._AbstractDiffusionTerm`

class `fipy.terms.DiffusionTermNoCorrection` (*coeff=(1.0,), var=None*)
Bases: `fipy.terms.abstractDiffusionTerm._AbstractDiffusionTerm`

class `fipy.terms.ExplicitDiffusionTerm` (*coeff=(1.0,), var=None*)
Bases: `fipy.terms.abstractDiffusionTerm._AbstractDiffusionTerm`

The discretization for the *ExplicitDiffusionTerm* is given by

$$\int_V \nabla \cdot (\Gamma \nabla \phi) dV \simeq \sum_f \Gamma_f \frac{\phi_A^\text{old} - \phi_P^\text{old}}{d_{AP}} A_f$$

where ϕ_A^old and ϕ_P^old are the old values of the variable. The term is added to the RHS vector and makes no contribution to the solution matrix.

`fipy.terms.ImplicitDiffusionTerm`
alias of `DiffusionTerm`

class `fipy.terms.ImplicitSourceTerm` (*coeff=0.0, var=None*)
Bases: `fipy.terms.sourceTerm.SourceTerm`

The *ImplicitSourceTerm* represents

$$\int_V \phi S \, dV \simeq \phi_P S_P V_P$$

where S is the *coeff* value.

class `fipy.terms.ResidualTerm` (*equation, underRelaxation=1.0*)
Bases: `fipy.terms.explicitSourceTerm._ExplicitSourceTerm`

The *ResidualTerm* is a special form of explicit *SourceTerm* that adds the residual of one equation to another equation. Useful for Newton's method.

class `fipy.terms.CentralDifferenceConvectionTerm` (*coeff=1.0, var=None*)
Bases: `fipy.terms.abstractConvectionTerm._AbstractConvectionTerm`

This `Term` represents

$$\int_V \nabla \cdot (\vec{u} \phi) \, dV \simeq \sum_f (\vec{n} \cdot \vec{u})_f \phi_f A_f$$

where $\phi_f = \alpha_f \phi_P + (1 - \alpha_f) \phi_A$ and α_f is calculated using the central differencing scheme. For further details see *Numerical Schemes*.

Create a *_AbstractConvectionTerm* object.

```
>>> from fipy import *
>>> m = Grid1D(nx = 2)
>>> cv = CellVariable(mesh = m)
>>> fv = FaceVariable(mesh = m)
>>> vcv = CellVariable(mesh=m, rank=1)
>>> vfv = FaceVariable(mesh=m, rank=1)
>>> __ConvectionTerm(coeff = cv)
Traceback (most recent call last):
    ...
VectorCoeffError: The coefficient must be a vector value.
>>> __ConvectionTerm(coeff = fv)
Traceback (most recent call last):
```

```
    ...
VectorCoeffError: The coefficient must be a vector value.
>>> __ConvectionTerm(coeff = vcv)
__ConvectionTerm(coeff=_ArithmeticCellToFaceVariable(value=array([[ 0.,  0.,  0.]]), mesh=Unifor
>>> __ConvectionTerm(coeff = vfv)
__ConvectionTerm(coeff=FaceVariable(value=array([[ 0.,  0.,  0.]]), mesh=UniformGrid1D(dx=1.0, n
>>> __ConvectionTerm(coeff = (1,))
__ConvectionTerm(coeff=(1,))
>>> ExplicitUpwindConvectionTerm(coeff = (0,)).solve(var=cv, solver=DummySolver())
Traceback (most recent call last):
    ...
TransientTermError: The equation requires a TransientTerm with explicit convection.
>>> (TransientTerm(0.) - ExplicitUpwindConvectionTerm(coeff = (0,))).solve(var=cv, solver=DummyS

>>> (TransientTerm() - ExplicitUpwindConvectionTerm(coeff = 1)).solve(var=cv, solver=DummySolver
Traceback (most recent call last):
    ...
VectorCoeffError: The coefficient must be a vector value.
>>> m2 = Grid2D(nx=2, ny=1)
>>> cv2 = CellVariable(mesh=m2)
>>> vcv2 = CellVariable(mesh=m2, rank=1)
>>> vfv2 = FaceVariable(mesh=m2, rank=1)
>>> __ConvectionTerm(coeff=vcv2)
__ConvectionTerm(coeff=_ArithmeticCellToFaceVariable(value=array([[ 0.,  0.,  0.,  0.,  0.,  0.,
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.]]), mesh=UniformGrid2D(dx=1.0, nx=2, dy=1.0, ny=1)))
>>> __ConvectionTerm(coeff=vfv2)
__ConvectionTerm(coeff=FaceVariable(value=array([[ 0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.]]), mesh=UniformGrid2D(dx=1.0, nx=2, dy=1.0, ny=1)))
>>> (TransientTerm() - ExplicitUpwindConvectionTerm(coeff = ((0,),(0,)))).solve(var=cv2, solver=
>>> (TransientTerm() - ExplicitUpwindConvectionTerm(coeff = (0,0))).solve(var=cv2, solver=DummyS
```

Parameters

- *coeff* : The *Term*'s coefficient value.

class `fipy.terms.`**`ExplicitUpwindConvectionTerm`**(*coeff=1.0*, *var=None*)

Bases: `fipy.terms.abstractUpwindConvectionTerm._AbstractUpwindConvectionTerm`

The discretization for this `Term` is given by

$$\int_V \nabla \cdot (\vec{u}\phi)\,dV \simeq \sum_f (\vec{n} \cdot \vec{u})_f \phi_f A_f$$

where $\phi_f = \alpha_f \phi_P^\text{old} + (1 - \alpha_f)\phi_A^\text{old}$ and α_f is calculated using the upwind scheme. For further details see *Numerical Schemes*.

Create a *_AbstractConvectionTerm* object.

```
>>> from fipy import *
>>> m = Grid1D(nx = 2)
>>> cv = CellVariable(mesh = m)
>>> fv = FaceVariable(mesh = m)
>>> vcv = CellVariable(mesh=m, rank=1)
>>> vfv = FaceVariable(mesh=m, rank=1)
>>> __ConvectionTerm(coeff = cv)
Traceback (most recent call last):
    ...
VectorCoeffError: The coefficient must be a vector value.
>>> __ConvectionTerm(coeff = fv)
```

```
Traceback (most recent call last):
    ...
VectorCoeffError: The coefficient must be a vector value.
>>> __ConvectionTerm(coeff = vcv)
__ConvectionTerm(coeff=_ArithmeticCellToFaceVariable(value=array([[ 0.,  0.,  0.]]), mesh=Unifor
>>> __ConvectionTerm(coeff = vfv)
__ConvectionTerm(coeff=FaceVariable(value=array([[ 0.,  0.,  0.]]), mesh=UniformGrid1D(dx=1.0, n
>>> __ConvectionTerm(coeff = (1,))
__ConvectionTerm(coeff=(1,))
>>> ExplicitUpwindConvectionTerm(coeff = (0,)).solve(var=cv, solver=DummySolver())
Traceback (most recent call last):
    ...
TransientTermError: The equation requires a TransientTerm with explicit convection.
>>> (TransientTerm(0.) - ExplicitUpwindConvectionTerm(coeff = (0,))).solve(var=cv, solver=DummyS

>>> (TransientTerm() - ExplicitUpwindConvectionTerm(coeff = 1)).solve(var=cv, solver=DummySolver
Traceback (most recent call last):
    ...
VectorCoeffError: The coefficient must be a vector value.
>>> m2 = Grid2D(nx=2, ny=1)
>>> cv2 = CellVariable(mesh=m2)
>>> vcv2 = CellVariable(mesh=m2, rank=1)
>>> vfv2 = FaceVariable(mesh=m2, rank=1)
>>> __ConvectionTerm(coeff=vcv2)
__ConvectionTerm(coeff=_ArithmeticCellToFaceVariable(value=array([[ 0.,  0.,  0.,  0.,  0.,  0.,
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.]]), mesh=UniformGrid2D(dx=1.0, nx=2, dy=1.0, ny=1)))
>>> __ConvectionTerm(coeff=vfv2)
__ConvectionTerm(coeff=FaceVariable(value=array([[ 0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.]]), mesh=UniformGrid2D(dx=1.0, nx=2, dy=1.0, ny=1)))
>>> (TransientTerm() - ExplicitUpwindConvectionTerm(coeff = ((0,),(0,)))).solve(var=cv2, solver=
>>> (TransientTerm() - ExplicitUpwindConvectionTerm(coeff = (0,0))).solve(var=cv2, solver=DummyS
```

Parameters

- *coeff* : The *Term*'s coefficient value.

class `fipy.terms.`**`ExponentialConvectionTerm`**(*coeff=1.0*, *var=None*)

Bases: `fipy.terms.asymmetricConvectionTerm._AsymmetricConvectionTerm`

The discretization for this `Term` is given by

$$\int_V \nabla \cdot (\vec{u}\phi)\,dV \simeq \sum_f (\vec{n} \cdot \vec{u})_f \phi_f A_f$$

where $\phi_f = \alpha_f \phi_P + (1 - \alpha_f)\phi_A$ and α_f is calculated using the exponential scheme. For further details see *Numerical Schemes*.

Create a *_AbstractConvectionTerm* object.

```
>>> from fipy import *
>>> m = Grid1D(nx = 2)
>>> cv = CellVariable(mesh = m)
>>> fv = FaceVariable(mesh = m)
>>> vcv = CellVariable(mesh=m, rank=1)
>>> vfv = FaceVariable(mesh=m, rank=1)
>>> __ConvectionTerm(coeff = cv)
Traceback (most recent call last):
    ...
```

```
VectorCoeffError: The coefficient must be a vector value.
>>> __ConvectionTerm(coeff = fv)
Traceback (most recent call last):
    ...
VectorCoeffError: The coefficient must be a vector value.
>>> __ConvectionTerm(coeff = vcv)
__ConvectionTerm(coeff=_ArithmeticCellToFaceVariable(value=array([[ 0.,  0.,  0.]]), mesh=Unifor
>>> __ConvectionTerm(coeff = vfv)
__ConvectionTerm(coeff=FaceVariable(value=array([[ 0.,  0.,  0.]]), mesh=UniformGrid1D(dx=1.0, n
>>> __ConvectionTerm(coeff = (1,))
__ConvectionTerm(coeff=(1,))
>>> ExplicitUpwindConvectionTerm(coeff = (0,)).solve(var=cv, solver=DummySolver())
Traceback (most recent call last):
    ...
TransientTermError: The equation requires a TransientTerm with explicit convection.
>>> (TransientTerm(0.) - ExplicitUpwindConvectionTerm(coeff = (0,))).solve(var=cv, solver=DummyS

>>> (TransientTerm() - ExplicitUpwindConvectionTerm(coeff = 1)).solve(var=cv, solver=DummySolver
Traceback (most recent call last):
    ...
VectorCoeffError: The coefficient must be a vector value.
>>> m2 = Grid2D(nx=2, ny=1)
>>> cv2 = CellVariable(mesh=m2)
>>> vcv2 = CellVariable(mesh=m2, rank=1)
>>> vfv2 = FaceVariable(mesh=m2, rank=1)
>>> __ConvectionTerm(coeff=vcv2)
__ConvectionTerm(coeff=_ArithmeticCellToFaceVariable(value=array([[ 0.,  0.,  0.,  0.,  0.,  0.,
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.]]), mesh=UniformGrid2D(dx=1.0, nx=2, dy=1.0, ny=1)))
>>> __ConvectionTerm(coeff=vfv2)
__ConvectionTerm(coeff=FaceVariable(value=array([[ 0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.]]), mesh=UniformGrid2D(dx=1.0, nx=2, dy=1.0, ny=1)))
>>> (TransientTerm() - ExplicitUpwindConvectionTerm(coeff = ((0,),(0,)))).solve(var=cv2, solver=
>>> (TransientTerm() - ExplicitUpwindConvectionTerm(coeff = (0,0))).solve(var=cv2, solver=DummyS
```

Parameters

- *coeff* : The *Term*'s coefficient value.

class `fipy.terms.`**`HybridConvectionTerm`**(*coeff=1.0*, *var=None*)

Bases: `fipy.terms.asymmetricConvectionTerm._AsymmetricConvectionTerm`

The discretization for this `Term` is given by

$$\int_V \nabla \cdot (\vec{u}\phi)\,dV \simeq \sum_f (\vec{n} \cdot \vec{u})_f \phi_f A_f$$

where $\phi_f = \alpha_f \phi_P + (1 - \alpha_f)\phi_A$ and α_f is calculated using the hybrid scheme. For further details see *Numerical Schemes*.

Create a *_AbstractConvectionTerm* object.

```
>>> from fipy import *
>>> m = Grid1D(nx = 2)
>>> cv = CellVariable(mesh = m)
>>> fv = FaceVariable(mesh = m)
>>> vcv = CellVariable(mesh=m, rank=1)
>>> vfv = FaceVariable(mesh=m, rank=1)
>>> __ConvectionTerm(coeff = cv)
```

```
Traceback (most recent call last):
    ...
VectorCoeffError: The coefficient must be a vector value.
>>> __ConvectionTerm(coeff = fv)
Traceback (most recent call last):
    ...
VectorCoeffError: The coefficient must be a vector value.
>>> __ConvectionTerm(coeff = vcv)
__ConvectionTerm(coeff=_ArithmeticCellToFaceVariable(value=array([[ 0.,  0.,  0.]]), mesh=Unifor
>>> __ConvectionTerm(coeff = vfv)
__ConvectionTerm(coeff=FaceVariable(value=array([[ 0.,  0.,  0.]]), mesh=UniformGrid1D(dx=1.0, n
>>> __ConvectionTerm(coeff = (1,))
__ConvectionTerm(coeff=(1,))
>>> ExplicitUpwindConvectionTerm(coeff = (0,)).solve(var=cv, solver=DummySolver())
Traceback (most recent call last):
    ...
TransientTermError: The equation requires a TransientTerm with explicit convection.
>>> (TransientTerm(0.) - ExplicitUpwindConvectionTerm(coeff = (0,))).solve(var=cv, solver=DummyS

>>> (TransientTerm() - ExplicitUpwindConvectionTerm(coeff = 1)).solve(var=cv, solver=DummySolver
Traceback (most recent call last):
    ...
VectorCoeffError: The coefficient must be a vector value.
>>> m2 = Grid2D(nx=2, ny=1)
>>> cv2 = CellVariable(mesh=m2)
>>> vcv2 = CellVariable(mesh=m2, rank=1)
>>> vfv2 = FaceVariable(mesh=m2, rank=1)
>>> __ConvectionTerm(coeff=vcv2)
__ConvectionTerm(coeff=_ArithmeticCellToFaceVariable(value=array([[ 0.,  0.,  0.,  0.,  0.,  0.,
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.]]), mesh=UniformGrid2D(dx=1.0, nx=2, dy=1.0, ny=1)))
>>> __ConvectionTerm(coeff=vfv2)
__ConvectionTerm(coeff=FaceVariable(value=array([[ 0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.]]), mesh=UniformGrid2D(dx=1.0, nx=2, dy=1.0, ny=1)))
>>> (TransientTerm() - ExplicitUpwindConvectionTerm(coeff = ((0,),(0,)))).solve(var=cv2, solver=
>>> (TransientTerm() - ExplicitUpwindConvectionTerm(coeff = (0,0))).solve(var=cv2, solver=DummyS
```

Parameters

- *coeff* : The *Term*‘s coefficient value.

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

Bases: fipy.terms.asymmetricConvectionTerm._AsymmetricConvectionTerm

The discretization for this Term is given by

$$\int_V \nabla \cdot (\vec{u}\phi)\, dV \simeq \sum_f (\vec{n} \cdot \vec{u})_f \phi_f A_f$$

where $\phi_f = \alpha_f \phi_P + (1 - \alpha_f)\phi_A$ and α_f is calculated using the power law scheme. For further details see *Numerical Schemes*.

Create a *_AbstractConvectionTerm* object.

```
>>> from fipy import *
>>> m = Grid1D(nx = 2)
>>> cv = CellVariable(mesh = m)
>>> fv = FaceVariable(mesh = m)
>>> vcv = CellVariable(mesh=m, rank=1)
```

```
>>> vfv = FaceVariable(mesh=m, rank=1)
>>> __ConvectionTerm(coeff = cv)
Traceback (most recent call last):
    ...
VectorCoeffError: The coefficient must be a vector value.
>>> __ConvectionTerm(coeff = fv)
Traceback (most recent call last):
    ...
VectorCoeffError: The coefficient must be a vector value.
>>> __ConvectionTerm(coeff = vcv)
__ConvectionTerm(coeff=_ArithmeticCellToFaceVariable(value=array([[ 0.,  0.,  0.]]), mesh=Unifor
>>> __ConvectionTerm(coeff = vfv)
__ConvectionTerm(coeff=FaceVariable(value=array([[ 0.,  0.,  0.]]), mesh=UniformGrid1D(dx=1.0, n
>>> __ConvectionTerm(coeff = (1,))
__ConvectionTerm(coeff=(1,))
>>> ExplicitUpwindConvectionTerm(coeff = (0,)).solve(var=cv, solver=DummySolver())
Traceback (most recent call last):
    ...
TransientTermError: The equation requires a TransientTerm with explicit convection.
>>> (TransientTerm(0.) - ExplicitUpwindConvectionTerm(coeff = (0,))).solve(var=cv, solver=DummyS

>>> (TransientTerm() - ExplicitUpwindConvectionTerm(coeff = 1)).solve(var=cv, solver=DummySolver
Traceback (most recent call last):
    ...
VectorCoeffError: The coefficient must be a vector value.
>>> m2 = Grid2D(nx=2, ny=1)
>>> cv2 = CellVariable(mesh=m2)
>>> vcv2 = CellVariable(mesh=m2, rank=1)
>>> vfv2 = FaceVariable(mesh=m2, rank=1)
>>> __ConvectionTerm(coeff=vcv2)
__ConvectionTerm(coeff=_ArithmeticCellToFaceVariable(value=array([[ 0.,  0.,  0.,  0.,  0.,  0.,
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.]]), mesh=UniformGrid2D(dx=1.0, nx=2, dy=1.0, ny=1)))
>>> __ConvectionTerm(coeff=vfv2)
__ConvectionTerm(coeff=FaceVariable(value=array([[ 0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.]]), mesh=UniformGrid2D(dx=1.0, nx=2, dy=1.0, ny=1)))
>>> (TransientTerm() - ExplicitUpwindConvectionTerm(coeff = ((0,),(0,)))).solve(var=cv2, solver=
>>> (TransientTerm() - ExplicitUpwindConvectionTerm(coeff = (0,0))).solve(var=cv2, solver=DummyS
```

Parameters

- *coeff* : The *Term*'s coefficient value.

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

Bases: fipy.terms.abstractUpwindConvectionTerm._AbstractUpwindConvectionTerm

The discretization for this Term is given by

$$\int_V \nabla \cdot (\vec{u}\phi)\,dV \simeq \sum_f (\vec{n} \cdot \vec{u})_f \phi_f A_f$$

where $\phi_f = \alpha_f \phi_P + (1 - \alpha_f)\phi_A$ and α_f is calculated using the upwind convection scheme. For further details see *Numerical Schemes*.

Create a *_AbstractConvectionTerm* object.

```
>>> from fipy import *
>>> m = Grid1D(nx = 2)
>>> cv = CellVariable(mesh = m)
```

```
>>> fv = FaceVariable(mesh = m)
>>> vcv = CellVariable(mesh=m, rank=1)
>>> vfv = FaceVariable(mesh=m, rank=1)
>>> __ConvectionTerm(coeff = cv)
Traceback (most recent call last):
    ...
VectorCoeffError: The coefficient must be a vector value.
>>> __ConvectionTerm(coeff = fv)
Traceback (most recent call last):
    ...
VectorCoeffError: The coefficient must be a vector value.
>>> __ConvectionTerm(coeff = vcv)
__ConvectionTerm(coeff=_ArithmeticCellToFaceVariable(value=array([[ 0.,  0.,  0.]]), mesh=Unifor
>>> __ConvectionTerm(coeff = vfv)
__ConvectionTerm(coeff=FaceVariable(value=array([[ 0.,  0.,  0.]]), mesh=UniformGrid1D(dx=1.0, n
>>> __ConvectionTerm(coeff = (1,))
__ConvectionTerm(coeff=(1,))
>>> ExplicitUpwindConvectionTerm(coeff = (0,)).solve(var=cv, solver=DummySolver())
Traceback (most recent call last):
    ...
TransientTermError: The equation requires a TransientTerm with explicit convection.
>>> (TransientTerm(0.) - ExplicitUpwindConvectionTerm(coeff = (0,))).solve(var=cv, solver=DummyS

>>> (TransientTerm() - ExplicitUpwindConvectionTerm(coeff = 1)).solve(var=cv, solver=DummySolver
Traceback (most recent call last):
    ...
VectorCoeffError: The coefficient must be a vector value.
>>> m2 = Grid2D(nx=2, ny=1)
>>> cv2 = CellVariable(mesh=m2)
>>> vcv2 = CellVariable(mesh=m2, rank=1)
>>> vfv2 = FaceVariable(mesh=m2, rank=1)
>>> __ConvectionTerm(coeff=vcv2)
__ConvectionTerm(coeff=_ArithmeticCellToFaceVariable(value=array([[ 0.,  0.,  0.,  0.,  0.,  0.,
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.]]), mesh=UniformGrid2D(dx=1.0, nx=2, dy=1.0, ny=1)))
>>> __ConvectionTerm(coeff=vfv2)
__ConvectionTerm(coeff=FaceVariable(value=array([[ 0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.]]), mesh=UniformGrid2D(dx=1.0, nx=2, dy=1.0, ny=1)))
>>> (TransientTerm() - ExplicitUpwindConvectionTerm(coeff = ((0,),(0,)))).solve(var=cv2, solver=
>>> (TransientTerm() - ExplicitUpwindConvectionTerm(coeff = (0,0))).solve(var=cv2, solver=DummyS
```

Parameters

- *coeff* : The *Term*'s coefficient value.

class `fipy.terms.`**`VanLeerConvectionTerm`**(*coeff=1.0*, *var=None*)

Bases: `fipy.terms.explicitUpwindConvectionTerm.ExplicitUpwindConvectionTerm`

Create a *_AbstractConvectionTerm* object.

```
>>> from fipy import *
>>> m = Grid1D(nx = 2)
>>> cv = CellVariable(mesh = m)
>>> fv = FaceVariable(mesh = m)
>>> vcv = CellVariable(mesh=m, rank=1)
>>> vfv = FaceVariable(mesh=m, rank=1)
>>> __ConvectionTerm(coeff = cv)
Traceback (most recent call last):
    ...
```

```
VectorCoeffError: The coefficient must be a vector value.
>>> __ConvectionTerm(coeff = fv)
Traceback (most recent call last):
    ...
VectorCoeffError: The coefficient must be a vector value.
>>> __ConvectionTerm(coeff = vcv)
__ConvectionTerm(coeff=_ArithmeticCellToFaceVariable(value=array([[ 0.,  0.,  0.]]), mesh=Unifor
>>> __ConvectionTerm(coeff = vfv)
__ConvectionTerm(coeff=FaceVariable(value=array([[ 0.,  0.,  0.]]), mesh=UniformGrid1D(dx=1.0, n
>>> __ConvectionTerm(coeff = (1,))
__ConvectionTerm(coeff=(1,))
>>> ExplicitUpwindConvectionTerm(coeff = (0,)).solve(var=cv, solver=DummySolver())
Traceback (most recent call last):
    ...
TransientTermError: The equation requires a TransientTerm with explicit convection.
>>> (TransientTerm(0.) - ExplicitUpwindConvectionTerm(coeff = (0,))).solve(var=cv, solver=DummyS

>>> (TransientTerm() - ExplicitUpwindConvectionTerm(coeff = 1)).solve(var=cv, solver=DummySolver
Traceback (most recent call last):
    ...
VectorCoeffError: The coefficient must be a vector value.
>>> m2 = Grid2D(nx=2, ny=1)
>>> cv2 = CellVariable(mesh=m2)
>>> vcv2 = CellVariable(mesh=m2, rank=1)
>>> vfv2 = FaceVariable(mesh=m2, rank=1)
>>> __ConvectionTerm(coeff=vcv2)
__ConvectionTerm(coeff=_ArithmeticCellToFaceVariable(value=array([[ 0.,  0.,  0.,  0.,  0.,  0.,
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.]]), mesh=UniformGrid2D(dx=1.0, nx=2, dy=1.0, ny=1)))
>>> __ConvectionTerm(coeff=vfv2)
__ConvectionTerm(coeff=FaceVariable(value=array([[ 0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.]]), mesh=UniformGrid2D(dx=1.0, nx=2, dy=1.0, ny=1)))
>>> (TransientTerm() - ExplicitUpwindConvectionTerm(coeff = ((0,),(0,)))).solve(var=cv2, solver=
>>> (TransientTerm() - ExplicitUpwindConvectionTerm(coeff = (0,0))).solve(var=cv2, solver=DummyS
```

Parameters

- *coeff* : The *Term*'s coefficient value.

25.2 `abstractBinaryTerm` Module

25.3 `abstractConvectionTerm` Module

25.4 `abstractDiffusionTerm` Module

25.5 `abstractUpwindConvectionTerm` Module

25.6 `asymmetricConvectionTerm` Module

25.7 `binaryTerm` Module

25.8 `cellTerm` Module

class `fipy.terms.cellTerm.`**`CellTerm`**(*coeff=1.0*, *var=None*)

Bases: `fipy.terms.nonDiffusionTerm._NonDiffusionTerm`

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

25.9 `centralDiffConvectionTerm` Module

class `fipy.terms.centralDiffConvectionTerm.`**`CentralDifferenceConvectionTerm`**(*coeff=1.0*, *var=None*)

Bases: `fipy.terms.abstractConvectionTerm._AbstractConvectionTerm`

This `Term` represents

$$\int_V \nabla \cdot (\vec{u} \phi)\, dV \simeq \sum_f (\vec{n} \cdot \vec{u})_f \phi_f A_f$$

where $\phi_f = \alpha_f \phi_P + (1 - \alpha_f) \phi_A$ and α_f is calculated using the central differencing scheme. For further details see *Numerical Schemes*.

Create a *_AbstractConvectionTerm* object.

```
>>> from fipy import *
>>> m = Grid1D(nx = 2)
>>> cv = CellVariable(mesh = m)
>>> fv = FaceVariable(mesh = m)
>>> vcv = CellVariable(mesh=m, rank=1)
>>> vfv = FaceVariable(mesh=m, rank=1)
>>> __ConvectionTerm(coeff = cv)
Traceback (most recent call last):
    ...
VectorCoeffError: The coefficient must be a vector value.
>>> __ConvectionTerm(coeff = fv)
Traceback (most recent call last):
    ...
```

```
VectorCoeffError: The coefficient must be a vector value.
>>> __ConvectionTerm(coeff = vcv)
__ConvectionTerm(coeff=_ArithmeticCellToFaceVariable(value=array([[ 0.,  0.,  0.]]), mesh=Unifor
>>> __ConvectionTerm(coeff = vfv)
__ConvectionTerm(coeff=FaceVariable(value=array([[ 0.,  0.,  0.]]), mesh=UniformGrid1D(dx=1.0, n
>>> __ConvectionTerm(coeff = (1,))
__ConvectionTerm(coeff=(1,))
>>> ExplicitUpwindConvectionTerm(coeff = (0,)).solve(var=cv, solver=DummySolver())
Traceback (most recent call last):
    ...
TransientTermError: The equation requires a TransientTerm with explicit convection.
>>> (TransientTerm(0.) - ExplicitUpwindConvectionTerm(coeff = (0,))).solve(var=cv, solver=DummyS

>>> (TransientTerm() - ExplicitUpwindConvectionTerm(coeff = 1)).solve(var=cv, solver=DummySolver
Traceback (most recent call last):
    ...
VectorCoeffError: The coefficient must be a vector value.
>>> m2 = Grid2D(nx=2, ny=1)
>>> cv2 = CellVariable(mesh=m2)
>>> vcv2 = CellVariable(mesh=m2, rank=1)
>>> vfv2 = FaceVariable(mesh=m2, rank=1)
>>> __ConvectionTerm(coeff=vcv2)
__ConvectionTerm(coeff=_ArithmeticCellToFaceVariable(value=array([[ 0.,  0.,  0.,  0.,  0.,  0.,
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.]]), mesh=UniformGrid2D(dx=1.0, nx=2, dy=1.0, ny=1)))
>>> __ConvectionTerm(coeff=vfv2)
__ConvectionTerm(coeff=FaceVariable(value=array([[ 0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.]]), mesh=UniformGrid2D(dx=1.0, nx=2, dy=1.0, ny=1)))
>>> (TransientTerm() - ExplicitUpwindConvectionTerm(coeff = ((0,),(0,)))).solve(var=cv2, solver=
>>> (TransientTerm() - ExplicitUpwindConvectionTerm(coeff = (0,0))).solve(var=cv2, solver=DummyS
```

Parameters

- *coeff* : The *Term*'s coefficient value.

25.10 `coupledBinaryTerm` Module

25.11 `diffusionTerm` Module

class `fipy.terms.diffusionTerm.`**`DiffusionTerm`**(*coeff=(1.0,), var=None*)

Bases: `fipy.terms.diffusionTermNoCorrection.DiffusionTermNoCorrection`

This term represents a higher order diffusion term. The order of the term is determined by the number of *coeffs*, such that:

```
DiffusionTerm(D1)
```

represents a typical 2nd-order diffusion term of the form

$$\nabla \cdot (D_1 \nabla \phi)$$

and:

```
DiffusionTerm((D1,D2))
```

represents a 4th-order Cahn-Hilliard term of the form

$$\nabla \cdot \{ D_1 \nabla [\nabla \cdot (D_2 \nabla \phi)] \}$$

and so on.

class fipy.terms.diffusionTerm.**DiffusionTermCorrection**(*coeff=(1.0,), var=None*)
Bases: fipy.terms.abstractDiffusionTerm._AbstractDiffusionTerm

class fipy.terms.diffusionTerm.**DiffusionTermNoCorrection**(*coeff=(1.0,), var=None*)
Bases: fipy.terms.abstractDiffusionTerm._AbstractDiffusionTerm

25.12 diffusionTermCorrection Module

class fipy.terms.diffusionTermCorrection.**DiffusionTermCorrection**(*coeff=(1.0,), var=None*)
Bases: fipy.terms.abstractDiffusionTerm._AbstractDiffusionTerm

25.13 diffusionTermNoCorrection Module

class fipy.terms.diffusionTermNoCorrection.**DiffusionTermNoCorrection**(*coeff=(1.0,), var=None*)
Bases: fipy.terms.abstractDiffusionTerm._AbstractDiffusionTerm

25.14 explicitDiffusionTerm Module

class fipy.terms.explicitDiffusionTerm.**ExplicitDiffusionTerm**(*coeff=(1.0,), var=None*)
Bases: fipy.terms.abstractDiffusionTerm._AbstractDiffusionTerm

The discretization for the *ExplicitDiffusionTerm* is given by

$$\int_V \nabla \cdot (\Gamma \nabla \phi) dV \simeq \sum_f \Gamma_f \frac{\phi_A^\text{old} - \phi_P^\text{old}}{d_{AP}} A_f$$

where ϕ_A^old and ϕ_P^old are the old values of the variable. The term is added to the RHS vector and makes no contribution to the solution matrix.

25.15 explicitSourceTerm Module

25.16 explicitUpwindConvectionTerm Module

class fipy.terms.explicitUpwindConvectionTerm.**ExplicitUpwindConvectionTerm**(*coeff=1.0, var=None*)
Bases: fipy.terms.abstractUpwindConvectionTerm._AbstractUpwindConvectionTerm

The discretization for this Term is given by

$$\int_V \nabla \cdot (\vec{u} \phi)\, dV \simeq \sum_f (\vec{n} \cdot \vec{u})_f \phi_f A_f$$

where $\phi_f = \alpha_f \phi_P^\text{old} + (1 - \alpha_f) \phi_A^\text{old}$ and α_f is calculated using the upwind scheme. For further details see *Numerical Schemes*.

Create a *_AbstractConvectionTerm* object.

```
>>> from fipy import *
>>> m = Grid1D(nx = 2)
>>> cv = CellVariable(mesh = m)
>>> fv = FaceVariable(mesh = m)
>>> vcv = CellVariable(mesh=m, rank=1)
>>> vfv = FaceVariable(mesh=m, rank=1)
>>> __ConvectionTerm(coeff = cv)
Traceback (most recent call last):
    ...
VectorCoeffError: The coefficient must be a vector value.
>>> __ConvectionTerm(coeff = fv)
Traceback (most recent call last):
    ...
VectorCoeffError: The coefficient must be a vector value.
>>> __ConvectionTerm(coeff = vcv)
__ConvectionTerm(coeff=_ArithmeticCellToFaceVariable(value=array([[ 0.,  0.,  0.]]), mesh=Unifor
>>> __ConvectionTerm(coeff = vfv)
__ConvectionTerm(coeff=FaceVariable(value=array([[ 0.,  0.,  0.]]), mesh=UniformGrid1D(dx=1.0, n
>>> __ConvectionTerm(coeff = (1,))
__ConvectionTerm(coeff=(1,))
>>> ExplicitUpwindConvectionTerm(coeff = (0,)).solve(var=cv, solver=DummySolver())
Traceback (most recent call last):
    ...
TransientTermError: The equation requires a TransientTerm with explicit convection.
>>> (TransientTerm(0.) - ExplicitUpwindConvectionTerm(coeff = (0,))).solve(var=cv, solver=DummyS

>>> (TransientTerm() - ExplicitUpwindConvectionTerm(coeff = 1)).solve(var=cv, solver=DummySolver
Traceback (most recent call last):
    ...
VectorCoeffError: The coefficient must be a vector value.
>>> m2 = Grid2D(nx=2, ny=1)
>>> cv2 = CellVariable(mesh=m2)
>>> vcv2 = CellVariable(mesh=m2, rank=1)
>>> vfv2 = FaceVariable(mesh=m2, rank=1)
>>> __ConvectionTerm(coeff=vcv2)
__ConvectionTerm(coeff=_ArithmeticCellToFaceVariable(value=array([[ 0.,  0.,  0.,  0.,  0.,  0.,
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.]]), mesh=UniformGrid2D(dx=1.0, nx=2, dy=1.0, ny=1)))
>>> __ConvectionTerm(coeff=vfv2)
__ConvectionTerm(coeff=FaceVariable(value=array([[ 0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.]]), mesh=UniformGrid2D(dx=1.0, nx=2, dy=1.0, ny=1)))
>>> (TransientTerm() - ExplicitUpwindConvectionTerm(coeff = ((0,),(0,)))).solve(var=cv2, solver=
>>> (TransientTerm() - ExplicitUpwindConvectionTerm(coeff = (0,0))).solve(var=cv2, solver=DummyS
```

Parameters

- *coeff* : The *Term*'s coefficient value.

25.17 `exponentialConvectionTerm` Module

class `fipy.terms.exponentialConvectionTerm.`**`ExponentialConvectionTerm`**(*coeff=1.0*, *var=None*)

Bases: `fipy.terms.asymmetricConvectionTerm._AsymmetricConvectionTerm`

The discretization for this Term is given by

$$\int_V \nabla \cdot (\vec{u}\phi)\,dV \simeq \sum_f (\vec{n} \cdot \vec{u})_f \phi_f A_f$$

where $\phi_f = \alpha_f \phi_P + (1 - \alpha_f)\phi_A$ and α_f is calculated using the exponential scheme. For further details see *Numerical Schemes*.

Create a *_AbstractConvectionTerm* object.

```
>>> from fipy import *
>>> m = Grid1D(nx = 2)
>>> cv = CellVariable(mesh = m)
>>> fv = FaceVariable(mesh = m)
>>> vcv = CellVariable(mesh=m, rank=1)
>>> vfv = FaceVariable(mesh=m, rank=1)
>>> __ConvectionTerm(coeff = cv)
Traceback (most recent call last):
    ...
VectorCoeffError: The coefficient must be a vector value.
>>> __ConvectionTerm(coeff = fv)
Traceback (most recent call last):
    ...
VectorCoeffError: The coefficient must be a vector value.
>>> __ConvectionTerm(coeff = vcv)
__ConvectionTerm(coeff=_ArithmeticCellToFaceVariable(value=array([[ 0.,  0.,  0.]]), mesh=Unifor
>>> __ConvectionTerm(coeff = vfv)
__ConvectionTerm(coeff=FaceVariable(value=array([[ 0.,  0.,  0.]]), mesh=UniformGrid1D(dx=1.0, n
>>> __ConvectionTerm(coeff = (1,))
__ConvectionTerm(coeff=(1,))
>>> ExplicitUpwindConvectionTerm(coeff = (0,)).solve(var=cv, solver=DummySolver())
Traceback (most recent call last):
    ...
TransientTermError: The equation requires a TransientTerm with explicit convection.
>>> (TransientTerm(0.) - ExplicitUpwindConvectionTerm(coeff = (0,))).solve(var=cv, solver=DummyS

>>> (TransientTerm() - ExplicitUpwindConvectionTerm(coeff = 1)).solve(var=cv, solver=DummySolver
Traceback (most recent call last):
    ...
VectorCoeffError: The coefficient must be a vector value.
>>> m2 = Grid2D(nx=2, ny=1)
>>> cv2 = CellVariable(mesh=m2)
>>> vcv2 = CellVariable(mesh=m2, rank=1)
>>> vfv2 = FaceVariable(mesh=m2, rank=1)
>>> __ConvectionTerm(coeff=vcv2)
__ConvectionTerm(coeff=_ArithmeticCellToFaceVariable(value=array([[ 0.,  0.,  0.,  0.,  0.,  0.,
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.]]), mesh=UniformGrid2D(dx=1.0, nx=2, dy=1.0, ny=1)))
>>> __ConvectionTerm(coeff=vfv2)
__ConvectionTerm(coeff=FaceVariable(value=array([[ 0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.]]), mesh=UniformGrid2D(dx=1.0, nx=2, dy=1.0, ny=1)))
>>> (TransientTerm() - ExplicitUpwindConvectionTerm(coeff = ((0,),(0,)))).solve(var=cv2, solver=
>>> (TransientTerm() - ExplicitUpwindConvectionTerm(coeff = (0,0))).solve(var=cv2, solver=DummyS
```

Parameters

- *coeff* : The *Term*‘s coefficient value.

25.18 faceTerm Module

class fipy.terms.faceTerm.**FaceTerm**(*coeff=1.0*, *var=None*)

Bases: fipy.terms.nonDiffusionTerm._NonDiffusionTerm

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

25.19 hybridConvectionTerm Module

class fipy.terms.hybridConvectionTerm.**HybridConvectionTerm**(*coeff=1.0*, *var=None*)

Bases: fipy.terms.asymmetricConvectionTerm._AsymmetricConvectionTerm

The discretization for this Term is given by

$$\int_V \nabla \cdot (\vec{u}\phi)\, dV \simeq \sum_f (\vec{n} \cdot \vec{u})_f \phi_f A_f$$

where $\phi_f = \alpha_f \phi_P + (1 - \alpha_f)\phi_A$ and α_f is calculated using the hybrid scheme. For further details see *Numerical Schemes*.

Create a *_AbstractConvectionTerm* object.

```
>>> from fipy import *
>>> m = Grid1D(nx = 2)
>>> cv = CellVariable(mesh = m)
>>> fv = FaceVariable(mesh = m)
>>> vcv = CellVariable(mesh=m, rank=1)
>>> vfv = FaceVariable(mesh=m, rank=1)
>>> __ConvectionTerm(coeff = cv)
Traceback (most recent call last):
    ...
VectorCoeffError: The coefficient must be a vector value.
>>> __ConvectionTerm(coeff = fv)
Traceback (most recent call last):
    ...
VectorCoeffError: The coefficient must be a vector value.
>>> __ConvectionTerm(coeff = vcv)
__ConvectionTerm(coeff=_ArithmeticCellToFaceVariable(value=array([[ 0.,  0.,  0.]]), mesh=Unifor
>>> __ConvectionTerm(coeff = vfv)
__ConvectionTerm(coeff=FaceVariable(value=array([[ 0.,  0.,  0.]]), mesh=UniformGrid1D(dx=1.0, n
>>> __ConvectionTerm(coeff = (1,))
__ConvectionTerm(coeff=(1,))
>>> ExplicitUpwindConvectionTerm(coeff = (0,)).solve(var=cv, solver=DummySolver())
Traceback (most recent call last):
    ...
TransientTermError: The equation requires a TransientTerm with explicit convection.
>>> (TransientTerm(0.) - ExplicitUpwindConvectionTerm(coeff = (0,))).solve(var=cv, solver=DummyS

>>> (TransientTerm() - ExplicitUpwindConvectionTerm(coeff = 1)).solve(var=cv, solver=DummySolver
Traceback (most recent call last):
    ...
VectorCoeffError: The coefficient must be a vector value.
>>> m2 = Grid2D(nx=2, ny=1)
>>> cv2 = CellVariable(mesh=m2)
>>> vcv2 = CellVariable(mesh=m2, rank=1)
```

```
>>> vfv2 = FaceVariable(mesh=m2, rank=1)
>>> __ConvectionTerm(coeff=vcv2)
__ConvectionTerm(coeff=_ArithmeticCellToFaceVariable(value=array([[ 0.,  0.,  0.,  0.,  0.,  0.,
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.]]), mesh=UniformGrid2D(dx=1.0, nx=2, dy=1.0, ny=1)))
>>> __ConvectionTerm(coeff=vfv2)
__ConvectionTerm(coeff=FaceVariable(value=array([[ 0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.]]), mesh=UniformGrid2D(dx=1.0, nx=2, dy=1.0, ny=1)))
>>> (TransientTerm() - ExplicitUpwindConvectionTerm(coeff = ((0,),(0,)))).solve(var=cv2, solver=
>>> (TransientTerm() - ExplicitUpwindConvectionTerm(coeff = (0,0))).solve(var=cv2, solver=DummyS
```

Parameters

- *coeff* : The *Term*'s coefficient value.

25.20 `implicitDiffusionTerm` Module

`fipy.terms.implicitDiffusionTerm.`**`ImplicitDiffusionTerm`**
alias of `DiffusionTerm`

25.21 `implicitSourceTerm` Module

class `fipy.terms.implicitSourceTerm.`**`ImplicitSourceTerm`**(*coeff=0.0*, *var=None*)
Bases: `fipy.terms.sourceTerm.SourceTerm`

The *ImplicitSourceTerm* represents

$$\int_V \phi S \, dV \simeq \phi_P S_P V_P$$

where S is the *coeff* value.

25.22 `nonDiffusionTerm` Module

25.23 `powerLawConvectionTerm` Module

class `fipy.terms.powerLawConvectionTerm.`**`PowerLawConvectionTerm`**(*coeff=1.0*, *var=None*)
Bases: `fipy.terms.asymmetricConvectionTerm._AsymmetricConvectionTerm`

The discretization for this `Term` is given by

$$\int_V \nabla \cdot (\vec{u}\phi) \, dV \simeq \sum_f (\vec{n} \cdot \vec{u})_f \phi_f A_f$$

where $\phi_f = \alpha_f \phi_P + (1 - \alpha_f)\phi_A$ and α_f is calculated using the power law scheme. For further details see *Numerical Schemes*.

Create a *_AbstractConvectionTerm* object.

```
>>> from fipy import *
>>> m = Grid1D(nx = 2)
>>> cv = CellVariable(mesh = m)
>>> fv = FaceVariable(mesh = m)
>>> vcv = CellVariable(mesh=m, rank=1)
>>> vfv = FaceVariable(mesh=m, rank=1)
>>> __ConvectionTerm(coeff = cv)
Traceback (most recent call last):
    ...
VectorCoeffError: The coefficient must be a vector value.
>>> __ConvectionTerm(coeff = fv)
Traceback (most recent call last):
    ...
VectorCoeffError: The coefficient must be a vector value.
>>> __ConvectionTerm(coeff = vcv)
__ConvectionTerm(coeff=_ArithmeticCellToFaceVariable(value=array([[ 0.,  0.,  0.]]), mesh=Unifor
>>> __ConvectionTerm(coeff = vfv)
__ConvectionTerm(coeff=FaceVariable(value=array([[ 0.,  0.,  0.]]), mesh=UniformGrid1D(dx=1.0, n
>>> __ConvectionTerm(coeff = (1,))
__ConvectionTerm(coeff=(1,))
>>> ExplicitUpwindConvectionTerm(coeff = (0,)).solve(var=cv, solver=DummySolver())
Traceback (most recent call last):
    ...
TransientTermError: The equation requires a TransientTerm with explicit convection.
>>> (TransientTerm(0.) - ExplicitUpwindConvectionTerm(coeff = (0,))).solve(var=cv, solver=DummyS

>>> (TransientTerm() - ExplicitUpwindConvectionTerm(coeff = 1)).solve(var=cv, solver=DummySolver
Traceback (most recent call last):
    ...
VectorCoeffError: The coefficient must be a vector value.
>>> m2 = Grid2D(nx=2, ny=1)
>>> cv2 = CellVariable(mesh=m2)
>>> vcv2 = CellVariable(mesh=m2, rank=1)
>>> vfv2 = FaceVariable(mesh=m2, rank=1)
>>> __ConvectionTerm(coeff=vcv2)
__ConvectionTerm(coeff=_ArithmeticCellToFaceVariable(value=array([[ 0.,  0.,  0.,  0.,  0.,  0.,
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.]]), mesh=UniformGrid2D(dx=1.0, nx=2, dy=1.0, ny=1)))
>>> __ConvectionTerm(coeff=vfv2)
__ConvectionTerm(coeff=FaceVariable(value=array([[ 0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.]]), mesh=UniformGrid2D(dx=1.0, nx=2, dy=1.0, ny=1)))
>>> (TransientTerm() - ExplicitUpwindConvectionTerm(coeff = ((0,),(0,)))).solve(var=cv2, solver=
>>> (TransientTerm() - ExplicitUpwindConvectionTerm(coeff = (0,0))).solve(var=cv2, solver=DummyS
```

Parameters

- *coeff* : The *Term*'s coefficient value.

25.24 `residualTerm` Module

class `fipy.terms.residualTerm.`**`ResidualTerm`**(*equation*, *underRelaxation=1.0*)

Bases: `fipy.terms.explicitSourceTerm._ExplicitSourceTerm`

The *ResidualTerm* is a special form of explicit *SourceTerm* that adds the residual of one equation to another equation. Useful for Newton's method.

25.25 `sourceTerm` Module

class `fipy.terms.sourceTerm.`**`SourceTerm`**(*coeff=0.0*, *var=None*)

Bases: `fipy.terms.cellTerm.CellTerm`

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

25.26 `term` Module

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

Bases: `object`

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

Create a *Term*.

Parameters

- *coeff*: The coefficient for the term. A *CellVariable* or number. *FaceVariable* objects are also acceptable for diffusion or convection terms.

`RHSvector`

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

`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`(*var=None*, *solver=None*, **args*, ***kwargs*)

`getMatrix`(**args*, ***kwds*)

Deprecated since version 3.0: use the `matrix` property instead

`getRHSvector`(**args*, ***kwds*)

Deprecated since version 3.0: use the `rHSvector` property instead

`justErrorVector`(*var=None*, *solver=None*, *boundaryConditions=()*, *dt=1.0*, *underRelaxation=None*, *residualFn=None*)

Builds the *Term*'s linear system once. This method also recalculates and returns the error as well as applying under-relaxation.

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.

justErrorVector returns the overlapping local value in parallel (not the non-overlapping value).

```
>>> from fipy import *
>>> m = Grid1D(nx=10)
>>> v = CellVariable(mesh=m)
>>> len(DiffusionTerm().justErrorVector(v, solver=DummySolver())) == m.numberOfCells
True
```

justResidualVector(*var=None*, *solver=None*, *boundaryConditions=()*, *dt=None*, *underRelaxation=None*, *residualFn=None*)

Builds the *Term*'s linear system once. This method also recalculates and returns the residual as well as applying under-relaxation.

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.

justResidualVector returns the overlapping local value in parallel (not the non-overlapping value).

```
>>> from fipy import *
>>> m = Grid1D(nx=10)
>>> v = CellVariable(mesh=m)
>>> len(DiffusionTerm().justResidualVector(v)) == m.numberOfCells
True
```

matrix

Return the matrix caculated in *solve()* or *sweep()*. The cacheMatrix() method should be called before *solve()* or *sweep()* to cache the matrix.

residualVectorAndNorm(*var=None*, *solver=None*, *boundaryConditions=()*, *dt=None*, *underRelaxation=None*, *residualFn=None*)

Builds the *Term*'s linear system once. This method also recalculates and returns the residual as well as applying under-relaxation.

Parameters

- *var*: 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=None*, *solver=None*, *boundaryConditions=()*, *dt=None*)
Builds and solves the *Term*'s linear system once. This method does not return the residual. It should be used when the residual is not required.

Parameters

- *var*: 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=None*, *solver=None*, *boundaryConditions=()*, *dt=None*, *underRelaxation=None*, *residualFn=None*, *cacheResidual=False*, *cacheError=False*)
Builds and solves the *Term*'s linear system once. This method also recalculates and returns the residual as well as applying under-relaxation.

Parameters

- *var*: 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.
- ***cacheResidual*: If *True*, calculate and store the residual vector** $\vec{r} = \mathsf{L}\vec{x} - \vec{b}$ in the *residualVector* member of *Term*
- ***cacheError*: If *True*, use the residual vector** $\vec{r}$ to solve $\mathsf{L}\vec{e} = \vec{r}$ for the error vector $\vec{e}$ and store it in the *errorVector* member of *Term*

25.27 `test` Module

25.28 `transientTerm` Module

class fipy.terms.transientTerm.**TransientTerm**(*coeff=1.0*, *var=None*)
Bases: `fipy.terms.cellTerm.CellTerm`

The *TransientTerm* represents

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

where ρ is the *coeff* value.

The following test case verifies that variable coefficients and old coefficient values work correctly. We will solve the following equation

$$\frac{\partial \phi^2}{\partial t} = k.$$

The analytic solution is given by

$$\phi = \sqrt{\phi_0^2 + kt},$$

where ϕ_0 is the initial value.

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

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

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

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

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

```
>>> 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
```

25.29 `unaryTerm` Module

25.30 `upwindConvectionTerm` Module

class `fipy.terms.upwindConvectionTerm.`**`UpwindConvectionTerm`**(*coeff=1.0*, *var=None*)
Bases: `fipy.terms.abstractUpwindConvectionTerm._AbstractUpwindConvectionTerm`

The discretization for this Term is given by

$$\int_V \nabla \cdot (\vec{u}\phi)\,dV \simeq \sum_f (\vec{n} \cdot \vec{u})_f \phi_f A_f$$

where $\phi_f = \alpha_f \phi_P + (1 - \alpha_f)\phi_A$ and α_f is calculated using the upwind convection scheme. For further details see *Numerical Schemes*.

Create a *_AbstractConvectionTerm* object.

```
>>> from fipy import *
>>> m = Grid1D(nx = 2)
>>> cv = CellVariable(mesh = m)
>>> fv = FaceVariable(mesh = m)
>>> vcv = CellVariable(mesh=m, rank=1)
>>> vfv = FaceVariable(mesh=m, rank=1)
>>> __ConvectionTerm(coeff = cv)
Traceback (most recent call last):
    ...
VectorCoeffError: The coefficient must be a vector value.
>>> __ConvectionTerm(coeff = fv)
Traceback (most recent call last):
    ...
VectorCoeffError: The coefficient must be a vector value.
>>> __ConvectionTerm(coeff = vcv)
__ConvectionTerm(coeff=_ArithmeticCellToFaceVariable(value=array([[ 0.,  0.,  0.]]), mesh=Unifor
>>> __ConvectionTerm(coeff = vfv)
__ConvectionTerm(coeff=FaceVariable(value=array([[ 0.,  0.,  0.]]), mesh=UniformGrid1D(dx=1.0, n
>>> __ConvectionTerm(coeff = (1,))
__ConvectionTerm(coeff=(1,))
>>> ExplicitUpwindConvectionTerm(coeff = (0,)).solve(var=cv, solver=DummySolver())
Traceback (most recent call last):
    ...
TransientTermError: The equation requires a TransientTerm with explicit convection.
>>> (TransientTerm(0.) - ExplicitUpwindConvectionTerm(coeff = (0,))).solve(var=cv, solver=DummyS

>>> (TransientTerm() - ExplicitUpwindConvectionTerm(coeff = 1)).solve(var=cv, solver=DummySolver
Traceback (most recent call last):
    ...
VectorCoeffError: The coefficient must be a vector value.
>>> m2 = Grid2D(nx=2, ny=1)
>>> cv2 = CellVariable(mesh=m2)
>>> vcv2 = CellVariable(mesh=m2, rank=1)
>>> vfv2 = FaceVariable(mesh=m2, rank=1)
>>> __ConvectionTerm(coeff=vcv2)
__ConvectionTerm(coeff=_ArithmeticCellToFaceVariable(value=array([[ 0.,  0.,  0.,  0.,  0.,  0.,
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.]]), mesh=UniformGrid2D(dx=1.0, nx=2, dy=1.0, ny=1)))
>>> __ConvectionTerm(coeff=vfv2)
__ConvectionTerm(coeff=FaceVariable(value=array([[ 0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.]]), mesh=UniformGrid2D(dx=1.0, nx=2, dy=1.0, ny=1)))
>>> (TransientTerm() - ExplicitUpwindConvectionTerm(coeff = ((0,),(0,)))).solve(var=cv2, solver=
>>> (TransientTerm() - ExplicitUpwindConvectionTerm(coeff = (0,0))).solve(var=cv2, solver=DummyS
```

Parameters

- *coeff* : The *Term*'s coefficient value.

25.31 `vanLeerConvectionTerm` Module

class `fipy.terms.vanLeerConvectionTerm.`**`VanLeerConvectionTerm`**(*coeff=1.0*, *var=None*)

Bases: `fipy.terms.explicitUpwindConvectionTerm.ExplicitUpwindConvectionTerm`

Create a *_AbstractConvectionTerm* object.

```
>>> from fipy import *
>>> m = Grid1D(nx = 2)
>>> cv = CellVariable(mesh = m)
>>> fv = FaceVariable(mesh = m)
>>> vcv = CellVariable(mesh=m, rank=1)
>>> vfv = FaceVariable(mesh=m, rank=1)
>>> __ConvectionTerm(coeff = cv)
Traceback (most recent call last):
    ...
VectorCoeffError: The coefficient must be a vector value.
>>> __ConvectionTerm(coeff = fv)
Traceback (most recent call last):
    ...
VectorCoeffError: The coefficient must be a vector value.
>>> __ConvectionTerm(coeff = vcv)
__ConvectionTerm(coeff=_ArithmeticCellToFaceVariable(value=array([[ 0.,  0.,  0.]]), mesh=Unifor
>>> __ConvectionTerm(coeff = vfv)
__ConvectionTerm(coeff=FaceVariable(value=array([[ 0.,  0.,  0.]]), mesh=UniformGrid1D(dx=1.0, n
>>> __ConvectionTerm(coeff = (1,))
__ConvectionTerm(coeff=(1,))
>>> ExplicitUpwindConvectionTerm(coeff = (0,)).solve(var=cv, solver=DummySolver())
Traceback (most recent call last):
    ...
TransientTermError: The equation requires a TransientTerm with explicit convection.
>>> (TransientTerm(0.) - ExplicitUpwindConvectionTerm(coeff = (0,))).solve(var=cv, solver=DummyS

>>> (TransientTerm() - ExplicitUpwindConvectionTerm(coeff = 1)).solve(var=cv, solver=DummySolver
Traceback (most recent call last):
    ...
VectorCoeffError: The coefficient must be a vector value.
>>> m2 = Grid2D(nx=2, ny=1)
>>> cv2 = CellVariable(mesh=m2)
>>> vcv2 = CellVariable(mesh=m2, rank=1)
>>> vfv2 = FaceVariable(mesh=m2, rank=1)
>>> __ConvectionTerm(coeff=vcv2)
__ConvectionTerm(coeff=_ArithmeticCellToFaceVariable(value=array([[ 0.,  0.,  0.,  0.,  0.,  0.,
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.]]), mesh=UniformGrid2D(dx=1.0, nx=2, dy=1.0, ny=1)))
>>> __ConvectionTerm(coeff=vfv2)
__ConvectionTerm(coeff=FaceVariable(value=array([[ 0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.]]), mesh=UniformGrid2D(dx=1.0, nx=2, dy=1.0, ny=1)))
>>> (TransientTerm() - ExplicitUpwindConvectionTerm(coeff = ((0,),(0,)))).solve(var=cv2, solver=
>>> (TransientTerm() - ExplicitUpwindConvectionTerm(coeff = (0,0))).solve(var=cv2, solver=DummyS
```

Parameters

- *coeff* : The *Term*'s coefficient value.

Chapter 26

tests Package

26.1 `tests` Package

unit testing scripts no chapter heading

26.2 `doctestPlus` Module

`fipy.tests.doctestPlus.`**`execButNoTest`**(*name='__main__'*)

`fipy.tests.doctestPlus.`**`register_skipper`**(*flag*, *test*, *why*, *skipWarning=True*)

Create a new doctest option flag for skipping tests

Parameters **flag** : str

Name of the option flag

test : function

A function which should return *True* if the test should be run

why : str

Explanation for why the test was skipped (to be used in a string "`Skipped %%(count)d doctest examples because %%(why)s`")

skipWarning : bool

Whether or not to report on tests skipped by this flag (default *True*)

`fipy.tests.doctestPlus.`**`report_skips`**()

Print out how many doctest examples were skipped due to flags

`fipy.tests.doctestPlus.`**`testmod`**(*m=None*, *name=None*, *globs=None*, *verbose=None*, *report=True*, *optionflags=0*, *extraglobs=None*, *raise_on_error=False*, *exclude_empty=False*)

Test examples in the given module. Return (#failures, #tests).

Largely duplicated from `doctest.testmod()`, but using `_SelectiveDocTestParser`.

Test examples in docstrings in functions and classes reachable from module m (or the current module if m is not supplied), starting with m.__doc__.

Also test examples reachable from dict m.__test__ if it exists and is not None. m.__test__ maps names to functions, classes and strings; function and class docstrings are tested even if the name is private; strings are tested directly, as if they were docstrings.

Return (#failures, #tests).

See help(doctest) for an overview.

Optional keyword arg "name" gives the name of the module; by default use m.__name__.

Optional keyword arg "globs" gives a dict to be used as the globals when executing examples; by default, use m.__dict__. A copy of this dict is actually used for each docstring, so that each docstring's examples start with a clean slate.

Optional keyword arg "extraglobs" gives a dictionary that should be merged into the globals that are used to execute examples. By default, no extra globals are used. This is new in 2.4.

Optional keyword arg "verbose" prints lots of stuff if true, prints only failures if false; by default, it's true iff "-v" is in sys.argv.

Optional keyword arg "report" prints a summary at the end when true, else prints nothing at the end. In verbose mode, the summary is detailed, else very brief (in fact, empty if all tests passed).

Optional keyword arg "optionflags" or's together module constants, and defaults to 0. This is new in 2.3. Possible values (see the docs for details):

> DONT_ACCEPT_TRUE_FOR_1 DONT_ACCEPT_BLANKLINE NORMALIZE_WHITESPACE ELLIPSIS SKIP IGNORE_EXCEPTION_DETAIL REPORT_UDIFF REPORT_CDIFF REPORT_NDIFF REPORT_ONLY_FIRST_FAILURE

as well as FiPy's flags

> GMSH SCIPY TVTK SERIAL PARALLEL PROCESSOR_0 PROCESSOR_0_OF_2 PROCESSOR_1_OF_2 PROCESSOR_0_OF_3 PROCESSOR_1_OF_3 PROCESSOR_2_OF_3

Optional keyword arg "raise_on_error" raises an exception on the first unexpected exception or failure. This allows failures to be post-mortem debugged.

26.3 `lateImportTest` Module

26.4 `testBase` Module

26.5 `testClass` Module

26.6 `testProgram` Module

Chapter 27

tools Package

27.1 `tools` Package

class `fipy.tools.`**`PhysicalField`**(*value*, *unit=None*, *array=None*)

Bases: `object`

Physical field or quantity with units

Physical Fields can be constructed in one of two ways:

- *PhysicalField(*value*, *unit*)*, where **value** is a number of arbitrary type and **unit** is a string containing the unit name

```
>>> print PhysicalField(value = 10., unit = 'm')
10.0 m
```

- *PhysicalField(*string*)*, where **string** contains both the value and the unit. This form is provided to make interactive use more convenient

```
>>> print PhysicalField(value = "10. m")
10.0 m
```

Dimensionless quantities, with a *unit* of 1, can be specified in several ways

```
>>> print PhysicalField(value = "1")
1.0 1
>>> print PhysicalField(value = 2., unit = " ")
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().allclose("1.57079632679 rad")
1
```

The input *PhysicalField* must be dimensionless

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

arccosh()

Return the inverse hyperbolic cosine of the *PhysicalField*

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

The input *PhysicalField* must be dimensionless

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

arcsin()

Return the inverse sine of the *PhysicalField* in radians

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

The input *PhysicalField* must be dimensionless

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

arctan()
Return the arctangent of the *PhysicalField* in radians

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

The input *PhysicalField* must be dimensionless

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

arctan2(*other*)
Return the arctangent of *self* divided by *other* in radians

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

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

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

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

arctanh()
Return the inverse hyperbolic tangent of the *PhysicalField*

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

The input *PhysicalField* must be dimensionless

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

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

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

conjugate()
Return the complex conjugate of the *PhysicalField*.

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

convertToUnit (*unit*)

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

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

copy ()

Make a duplicate.

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

The duplicate will not reflect changes made to the original

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

Likewise for arrays

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

cos ()

Return the cosine of the *PhysicalField*

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

The units of the *PhysicalField* must be an angle

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

cosh ()

Return the hyperbolic cosine of the *PhysicalField*

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

The units of the *PhysicalField* must be dimensionless

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

divide (*other*)

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

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

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

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

dot (*other*)

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

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

floor ()

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

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

getNumericValue (**args*, ***kwds*)

Deprecated since version 3.0: use the `numericValue` property instead

getShape (**args*, ***kwds*)

Deprecated since version 3.0: use the `shape` property instead

getUnit (**args*, ***kwds*)

Deprecated since version 3.0: use the `unit` property instead

getsctype (*default=None*)

Returns the Numpy sctype of the underlying array.

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

inBaseUnits ()

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

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

inDimensionless ()

Returns the numerical value of a dimensionless quantity.

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

It's an error to convert a quantity with units

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

inRadians()

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

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

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

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

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

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

inSIUnits()

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

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

inUnitsOf(**units*)

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

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

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

```
>>> t = PhysicalField(314159., 's')
>>> print numerix.allclose([e.allclose(v) for (e, v) in zip(t.inUnitsOf('d','h','min','s'),
...                                                         ['3.0 d', '15.0 h', '15.0 min',
...                             True)
1
```

isCompatible (*unit*)

itemset (*value*)

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

```
>>> a = PhysicalField(4.,"m")
>>> a.itemset(PhysicalField("6 ft"))
>>> print a.allclose("1.8288 m")
1
>>> a = PhysicalField(((3.,4.),(5.,6.)),"m")
>>> 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
```

itemsize

log ()

Return the natural logarithm of the *PhysicalField*

```
>>> print numerix.round_(PhysicalField(10).log(), 6)
2.302585
```

The input *PhysicalField* must be dimensionless

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

log10 ()

Return the base-10 logarithm of the *PhysicalField*

```
>>> print numerix.round_(PhysicalField(10.).log10(), 6)
1.0
```

The input *PhysicalField* must be dimensionless

```
>>> print numerix.round_(PhysicalField("1. m").log10(), 6)
Traceback (most recent call last):
    ...
TypeError: Incompatible units
```

multiply (*other*)

Multiply two physical quantities. The unit of the result is the product of the units of the operands.

```
>>> print PhysicalField(10., 'N') * PhysicalField(10., 'm')
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 *1*. This facilitates passing physical quantities to packages such as Numeric that cannot use units, while ensuring the quantities have the desired units.

```
>>> print (PhysicalField(10., 's') * PhysicalField(2., 'Hz'))
20.0
```

numericValue

Return the *PhysicalField* without units, after conversion to base SI units.

```
>>> print numerix.round_(PhysicalField("1 inch").numericValue, 6)
0.0254
```

put (*indices*, *values*)

put is the opposite of *take*. The values of *self* at the locations specified in *indices* are set to the corresponding value of *values*.

The *indices* can be any integer sequence object with values suitable for indexing into the flat form of *self*. The *values* must be any sequence of values that can be converted to the typecode of *self*.

```
>>> f = PhysicalField((1.,2.,3.),"m")
>>> f.put((2,0), PhysicalField((2.,3.),"inch"))
>>> print f
[ 0.0762  2.      0.0508] m
```

The units of *values* must be compatible with *self*.

```
>>> f.put(1, PhysicalField(3,"kg"))
Traceback (most recent call last):
    ...
TypeError: Incompatible units
```

ravel ()

reshape (*shape*)

Changes the shape of *self* to that specified in *shape*

```
>>> print PhysicalField((1.,2.,3.,4.),"m").reshape((2,2))
[[ 1.  2.]
 [ 3.  4.]] m
```

The new shape must have the same size as the existing one.

```
>>> print PhysicalField((1.,2.,3.,4.),"m").reshape((2,3))
Traceback (most recent call last):
    ...
ValueError: total size of new array must be unchanged
```

setUnit (**args*, ***kwds*)

Deprecated since version 3.0: use the `unit` property instead

shape

Tuple of array dimensions.

sign ()

Return the sign of the quantity. The *unit* is unchanged.

```
>>> from fipy.tools.numerix import sign
>>> print sign(PhysicalField(((3.,-2.),(-1.,4.)), 'm'))
[[ 1. -1.]
 [-1.  1.]]
```

sin ()

Return the sine of the *PhysicalField*

```
>>> print PhysicalField(numerix.pi/6,"rad").sin()
0.5
>>> print PhysicalField(30.,"deg").sin()
0.5
```

The units of the PhysicalField must be an angle

```
>>> PhysicalField(30.,"m").sin()
Traceback (most recent call last):
    ...
TypeError: Incompatible units
```

sinh()

Return the hyperbolic sine of the *PhysicalField*

```
>>> PhysicalField(0.).sinh()
0.0
```

The units of the *PhysicalField* must be dimensionless

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

sqrt()

Return the square root of the *PhysicalField*

```
>>> print PhysicalField("100. m**2").sqrt()
10.0 m
```

The resulting unit must be integral

```
>>> print PhysicalField("100. m").sqrt()
Traceback (most recent call last):
    ...
TypeError: Illegal exponent
```

subtract(*other*)

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

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

sum(*index=0*)

Returns the sum of all of the elements in *self* along the specified axis (first axis by default).

```
>>> print PhysicalField(((1.,2.),(3.,4.)), "m").sum()
[ 4.  6.] m
>>> print PhysicalField(((1.,2.),(3.,4.)), "m").sum(1)
[ 3.  7.] m
```

take(*indices*, *axis=0*)

Return the elements of *self* specified by the elements of *indices*. The resulting *PhysicalField* array has the same units as the original.

```
>>> print PhysicalField((1.,2.,3.),"m").take((2,0))
[ 3.  1.] m
```

The optional third argument specifies the axis along which the selection occurs, and the default value (as in the example above) is 0, the first axis.

```
>>> print PhysicalField(((1.,2.,3.),(4.,5.,6.)),"m").take((2,0), axis = 1)
[[ 3.  1.]
 [ 6.  4.]] m
```

tan()

Return the tangent of the *PhysicalField*

```
>>> numerix.round_(PhysicalField(numerix.pi/4,"rad").tan(), 6)
1.0
>>> numerix.round_(PhysicalField(45,"deg").tan(), 6)
1.0
```

The units of the *PhysicalField* must be an angle

```
>>> PhysicalField(45.,"m").tan()
Traceback (most recent call last):
    ...
TypeError: Incompatible units
```

tanh()

Return the hyperbolic tangent of the *PhysicalField*

```
>>> print numerix.allclose(PhysicalField(1.).tanh(), 0.761594155956)
True
```

The units of the *PhysicalField* must be dimensionless

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

tostring(*max_line_width=75*, *precision=8*, *suppress_small=False*, *separator=' '*)

Return human-readable form of a physical quantity

```
>>> p = PhysicalField(value = (3., 3.14159), unit = "eV")
>>> print p.tostring(precision = 3, separator = '|')
[ 3.    | 3.142] eV
```

unit

Return the unit object of *self*.

```
>>> PhysicalField("1 m").unit
<PhysicalUnit m>
```

class fipy.tools.**Vitals**

Bases: xml.dom.minidom.Document

Returns XML formatted information about current FiPy environment

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*)

27.2 copy_script Module

class fipy.tools.copy_script.**Copy_script**(*dist*)
Bases: distutils.cmd.Command

Create and initialize a new Command object. Most importantly, invokes the 'initialize_options()' method, which is the real initializer and depends on the actual command being instantiated.

description = 'copy an example script into a new editable file'

finalize_options()

initialize_options()

run()

user_options = [('From=', None, 'path and file name containing script to copy'), ('To=', None, 'path and file name to s

27.3 debug Module

fipy.tools.debug.**PRINT**(*label*, *arg=''*, *stall=True*)

27.4 decorators Module

fipy.tools.decorators.**getsetDeprecated**(**args*, ***kwargs*)
Issues a *DeprecationWarning* to use the appropriate property, rather than the get/set method of the same name

This function may also be used as a decorator.

Parameters **func** : function

The function to be deprecated.

old_name : str, optional

The name of the function to be deprecated. Default is None, in which case the name of *func* is used.

new_name : str, optional

The new name for the function. Default is None, in which case the deprecation message is that *old_name* is deprecated. If given, the deprecation message is that *old_name* is deprecated and *new_name* should be used instead.

message : str, optional

Additional explanation of the deprecation. Displayed in the docstring after the warning.

Returns **old_func** : function

The deprecated function.

fipy.tools.decorators.**mathMethodDeprecated**(**args*, ***kwargs*)
Issues a *DeprecationWarning* to use the appropriate ufunc from *numerix*, rather than the method of the same name

This function may also be used as a decorator.

Parameters **func** : function

The function to be deprecated.

old_name : str, optional

The name of the function to be deprecated. Default is None, in which case the name of *func* is used.

new_name : str, optional

The new name for the function. Default is None, in which case the deprecation message is that *old_name* is deprecated. If given, the deprecation message is that *old_name* is deprecated and *new_name* should be used instead.

message : str, optional

Additional explanation of the deprecation. Displayed in the docstring after the warning.

Returns **old_func** : function

The deprecated function.

27.5 `dump` Module

`fipy.tools.dump.`**`write`**(*data*, *filename=None*, *extension=''*, *communicator=ParallelCommWrapper()*)

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.
- *communicator*: Object with *procID* and *Nproc* attributes.

Test to check pickling and unpickling.

```
>>> from fipy.meshes import Grid1D
>>> old = Grid1D(nx = 2)
>>> f, tempfile = write(old)
>>> new = read(tempfile, f)
>>> print old.numberOfCells == new.numberOfCells
True
```

`fipy.tools.dump.`**`read`**(*filename*, *fileobject=None*, *communicator=ParallelCommWrapper()*, *mesh_unmangle=False*)

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
- *communicator*: Object with *procID* and *Nproc* attributes.
- *mesh_unmangle*: Correct improper pickling of non-uniform meshes (ticket:243)

27.6 `inline` Module

27.7 `numerix` Module

Replacement module for NumPy

> **Attention:** This module should be the only place in the code where `numpy` is explicitly imported and you should always import this module and not `numpy` in your own code. The documentation for `numpy` remains canonical for all functions and classes not explicitly documented here.

The functions provided in ths module replace and augment the *NumPy* module. The functions work with *Variables*, arrays or numbers. For example, create a *Variable*.

```
>>> from fipy.variables.variable import Variable
>>> var = Variable(value=0)
```

Take the tangent of such a variable. The returned value is itself a *Variable*.

```
>>> v = tan(var)
>>> v
tan(Variable(value=array(0)))
>>> print float(v)
0.0
```

Take the tangent of a int.

```
>>> tan(0)
0.0
```

Take the tangent of an array.

```
>>> print tan(array((0,0,0)))
[ 0.  0.  0.]
```

`fipy.tools.numerix.`**`dot`**(*a1*, *a2*, *axis=0*)

return array of vector dot-products of v1 and v2 for arrays a1 and a2 of vectors v1 and v2

We can't use `numpy.dot()` on an array of vectors

Test that Variables are returned as Variables.

```
>>> from fipy.meshes import Grid2D
>>> mesh = Grid2D(nx=2, ny=1)
>>> from fipy.variables.cellVariable import CellVariable
>>> v1 = CellVariable(mesh=mesh, value=((0,1),(2,3)), rank=1)
>>> v2 = CellVariable(mesh=mesh, value=((0,1),(2,3)), rank=1)
>>> dot(v1, v2)._variableClass
<class 'fipy.variables.cellVariable.CellVariable'>
>>> dot(v2, v1)._variableClass
<class 'fipy.variables.cellVariable.CellVariable'>
>>> print rank(dot(v2, v1))
0
>>> print dot(v1, v2)
[ 4 10]
>>> dot(v1, v1)._variableClass
<class 'fipy.variables.cellVariable.CellVariable'>
>>> print dot(v1, v1)
[ 4 10]
```

```
>>> v3 = array(((0,1),(2,3)))
>>> print type(dot(v3, v3)) is type(array(1))
1
>>> print dot(v3, v3)
[ 4 10]
```

fipy.tools.numerix.**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
```

fipy.tools.numerix.**allclose**(*first*, *second*, *rtol=1e-05*, *atol=1e-08*)
Tests whether or not *first* and *second* are equal, subect to the given relative and absolute tolerances, such that:

```
|first - second| < atol + rtol * |second|
```

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

fipy.tools.numerix.**take**(*a*, *indices*, *axis=0*, *fill_value=None*)
Selects the elements of *a* corresponding to *indices*.

fipy.tools.numerix.**reshape**(*arr*, *shape*)
Change the shape of *arr* to *shape*, as long as the product of all the lenghts of all the axes is constant (the total number of elements does not change).

fipy.tools.numerix.**put**(*arr*, *ids*, *values*)
The opposite of *take*. The values of *arr* at the locations specified by *ids* are set to the corresponding value of *values*.

The following is to test improvments to puts with masked arrays. Places in the code were assuming incorrect put behavior.

```
>>> maskValue = 999999

>>> arr = zeros(3, 'l')
>>> ids = MA.masked_values((2, maskValue), maskValue)
>>> values = MA.masked_values((4, maskValue), maskValue)
>>> put(arr, ids, values) ## this should work
>>> print arr
[0 0 4]

>>> arr = MA.masked_values((maskValue, 5, 10), maskValue)
>>> ids = MA.masked_values((2, maskValue), maskValue)
>>> values = MA.masked_values((4, maskValue), maskValue)
>>> put(arr, ids, values)
>>> print arr ## works as expected
[-- 5 4]

>>> arr = MA.masked_values((maskValue, 5, 10), maskValue)
>>> ids = MA.masked_values((maskValue, 2), maskValue)
>>> values = MA.masked_values((4, maskValue), maskValue)
```

```
>>> put(arr, ids, values)
>>> print arr ## should be [-- 5 --] maybe??
[-- 5 999999]
```

`fipy.tools.numerix.sum(arr, axis=0)`
The sum of all the elements of *arr* along the specified axis.

`fipy.tools.numerix.all(a, axis=None, out=None)`
Test whether all array elements along a given axis evaluate to True.

Parameters **a** : array_like

Input array or object that can be converted to an array.

axis : int, optional

Axis along which an logical AND is performed. The default (*axis* = *None*) is to perform a logical AND over a flattened input array. *axis* may be negative, in which case it counts from the last to the first axis.

out : ndarray, optional

Alternative output array in which to place the result. It must have the same shape as the expected output and the type is preserved.

`fipy.tools.numerix.rank(a)`
Get the rank of sequence a (the number of dimensions, not a matrix rank) The rank of a scalar is zero.

Note: The rank of a *MeshVariable* is for any single element. E.g., A *CellVariable* containing scalars at each cell, and defined on a 9 element *Grid1D*, has rank 0. If it is defined on a 3x3 *Grid2D*, it is still rank 0.

`fipy.tools.numerix.take(a, indices, axis=0, fill_value=None)`
Selects the elements of *a* corresponding to *indices*.

`fipy.tools.numerix.reshape(arr, shape)`
Change the shape of *arr* to *shape*, as long as the product of all the lenghts of all the axes is constant (the total number of elements does not change).

`fipy.tools.numerix.put(arr, ids, values)`
The opposite of *take*. The values of *arr* at the locations specified by *ids* are set to the corresponding value of *values*.

The following is to test improvments to puts with masked arrays. Places in the code were assuming incorrect put behavior.

```
>>> maskValue = 999999
```

```
>>> arr = zeros(3, 'l')
>>> ids = MA.masked_values((2, maskValue), maskValue)
>>> values = MA.masked_values((4, maskValue), maskValue)
>>> put(arr, ids, values) ## this should work
>>> print arr
[0 0 4]
```

```
>>> arr = MA.masked_values((maskValue, 5, 10), maskValue)
>>> ids = MA.masked_values((2, maskValue), maskValue)
>>> values = MA.masked_values((4, maskValue), maskValue)
>>> put(arr, ids, values)
>>> print arr ## works as expected
[-- 5 4]
```

```
>>> arr = MA.masked_values((maskValue, 5, 10), maskValue)
>>> ids = MA.masked_values((maskValue, 2), maskValue)
>>> values = MA.masked_values((4, maskValue), maskValue)
>>> put(arr, ids, values)
>>> print arr ## should be [-- 5 --] maybe??
[-- 5 999999]
```

`fipy.tools.numerix.sum`(*arr*, *axis=0*)
The sum of all the elements of *arr* along the specified axis.

`fipy.tools.numerix.all`(*a*, *axis=None*, *out=None*)
Test whether all array elements along a given axis evaluate to True.

Parameters **a** : array_like

Input array or object that can be converted to an array.

axis : int, optional

Axis along which an logical AND is performed. The default (*axis* = *None*) is to perform a logical AND over a flattened input array. *axis* may be negative, in which case it counts from the last to the first axis.

out : ndarray, optional

Alternative output array in which to place the result. It must have the same shape as the expected output and the type is preserved.

`fipy.tools.numerix.rank`(*a*)
Get the rank of sequence a (the number of dimensions, not a matrix rank) The rank of a scalar is zero.

Note: The rank of a *MeshVariable* is for any single element. E.g., A *CellVariable* containing scalars at each cell, and defined on a 9 element *Grid1D*, has rank 0. If it is defined on a 3x3 *Grid2D*, it is still rank 0.

`fipy.tools.numerix.all`(*a*, *axis=None*, *out=None*)
Test whether all array elements along a given axis evaluate to True.

Parameters **a** : array_like

Input array or object that can be converted to an array.

axis : int, optional

Axis along which an logical AND is performed. The default (*axis* = *None*) is to perform a logical AND over a flattened input array. *axis* may be negative, in which case it counts from the last to the first axis.

out : ndarray, optional

Alternative output array in which to place the result. It must have the same shape as the expected output and the type is preserved.

`fipy.tools.numerix.allclose`(*first*, *second*, *rtol=1e-05*, *atol=1e-08*)
Tests whether or not *first* and *second* are equal, subect to the given relative and absolute tolerances, such that:

```
|first - second| < atol + rtol * |second|
```

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

`fipy.tools.numerix.allequal`(*first*, *second*)
Returns *true* if every element of *first* is equal to the corresponding element of *second*.

fipy.tools.numerix.**dot**(*a1*, *a2*, *axis=0*)
return array of vector dot-products of v1 and v2 for arrays a1 and a2 of vectors v1 and v2

We can't use numpy.dot() on an array of vectors

Test that Variables are returned as Variables.

```
>>> from fipy.meshes import Grid2D
>>> mesh = Grid2D(nx=2, ny=1)
>>> from fipy.variables.cellVariable import CellVariable
>>> v1 = CellVariable(mesh=mesh, value=((0,1),(2,3)), rank=1)
>>> v2 = CellVariable(mesh=mesh, value=((0,1),(2,3)), rank=1)
>>> dot(v1, v2)._variableClass
<class 'fipy.variables.cellVariable.CellVariable'>
>>> dot(v2, v1)._variableClass
<class 'fipy.variables.cellVariable.CellVariable'>
>>> print rank(dot(v2, v1))
0
>>> print dot(v1, v2)
[ 4 10]
>>> dot(v1, v1)._variableClass
<class 'fipy.variables.cellVariable.CellVariable'>
>>> print dot(v1, v1)
[ 4 10]
>>> v3 = array(((0,1),(2,3)))
>>> print type(dot(v3, v3)) is type(array(1))
1
>>> print dot(v3, v3)
[ 4 10]
```

fipy.tools.numerix.**getShape**(*arr*)
Return the shape of *arr*

```
>>> getShape(1)
()
>>> getShape(1.)
()
>>> from fipy.variables.variable import Variable
>>> getShape(Variable(1))
()
>>> getShape(Variable(1.))
()
>>> getShape(Variable(1., unit="m"))
()
>>> getShape(Variable("1 m"))
()
```

fipy.tools.numerix.**getUnit**(*arr*)

fipy.tools.numerix.**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
```

`fipy.tools.numerix.`**`isclose`**(*first*, *second*, *rtol=1e-05*, *atol=1e-08*)

Returns which elements of *first* and *second* are equal, subect to the given relative and absolute tolerances, such that:

```
|first - second| < atol + rtol * |second|
```

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

`fipy.tools.numerix.`**`isFloat`**(*arr*)

`fipy.tools.numerix.`**`isInt`**(*arr*)

`fipy.tools.numerix.`**`L1norm`**(*arr*)

Parameters

- *arr*: The *array* to evaluate.

Returns $\|\texttt{arr}\|_1 = \sum_{j=1}^{n} |\texttt{arr}_j|$ is the L^1-norm of `arr`.

`fipy.tools.numerix.`**`L2norm`**(*arr*)

Parameters

- *arr*: The *array* to evaluate.

Returns $\|\texttt{arr}\|_2 = \sqrt{\sum_{j=1}^{n} |\texttt{arr}_j|^2}$ is the L^2-norm of `arr`.

`fipy.tools.numerix.`**`LINFnorm`**(*arr*)

Parameters

- *arr*: The *array* to evaluate.

Returns $\frac{\|\texttt{arr}\|_\infty = [\sum_{j=1}^{n} |\texttt{arr}_j|^\infty]^\infty =}{\max j |\texttt{arr}_j|}$ is the L^∞-norm of `arr`.

`fipy.tools.numerix.`**`nearest`**(*data*, *points*, *max_mem=100000000.0*)

find the indices of *data* that are closest to *points*

```
>>> from fipy import *
>>> m0 = Grid2D(dx=(.1, 1., 10.), dy=(.1, 1., 10.))
>>> m1 = Grid2D(nx=2, ny=2, dx=5., dy=5.)
>>> print nearest(m0.cellCenters.globalValue, m1.cellCenters.globalValue)
[4 5 7 8]
>>> print nearest(m0.cellCenters.globalValue, m1.cellCenters.globalValue, max_mem=100)
[4 5 7 8]
>>> print nearest(m0.cellCenters.globalValue, m1.cellCenters.globalValue, max_mem=10000)
[4 5 7 8]
```

`fipy.tools.numerix.`**`put`**(*arr*, *ids*, *values*)

The opposite of *take*. The values of *arr* at the locations specified by *ids* are set to the corresponding value of *values*.

The following is to test improvments to puts with masked arrays. Places in the code were assuming incorrect put behavior.

```
>>> maskValue = 999999
```

```
>>> arr = zeros(3, 'l')
>>> ids = MA.masked_values((2, maskValue), maskValue)
>>> values = MA.masked_values((4, maskValue), maskValue)
>>> put(arr, ids, values) ## this should work
>>> print arr
[0 0 4]

>>> arr = MA.masked_values((maskValue, 5, 10), maskValue)
>>> ids = MA.masked_values((2, maskValue), maskValue)
>>> values = MA.masked_values((4, maskValue), maskValue)
>>> put(arr, ids, values)
>>> print arr ## works as expected
[-- 5 4]

>>> arr = MA.masked_values((maskValue, 5, 10), maskValue)
>>> ids = MA.masked_values((maskValue, 2), maskValue)
>>> values = MA.masked_values((4, maskValue), maskValue)
>>> put(arr, ids, values)
>>> print arr ## should be [-- 5 --] maybe??
[-- 5 999999]
```

fipy.tools.numerix.**rank**(*a*)

Get the rank of sequence a (the number of dimensions, not a matrix rank) The rank of a scalar is zero.

Note: The rank of a *MeshVariable* is for any single element. E.g., A *CellVariable* containing scalars at each cell, and defined on a 9 element *Grid1D*, has rank 0. If it is defined on a 3x3 *Grid2D*, it is still rank 0.

fipy.tools.numerix.**reshape**(*arr*, *shape*)

Change the shape of *arr* to *shape*, as long as the product of all the lenghts of all the axes is constant (the total number of elements does not change).

fipy.tools.numerix.**sqrtDot**(*a1*, *a2*)

Return array of square roots of vector dot-products for arrays a1 and a2 of vectors v1 and v2

Usually used with v1==v2 to return magnitude of v1.

fipy.tools.numerix.**sum**(*arr*, *axis=0*)

The sum of all the elements of *arr* along the specified axis.

fipy.tools.numerix.**take**(*a*, *indices*, *axis=0*, *fill_value=None*)

Selects the elements of *a* corresponding to *indices*.

fipy.tools.numerix.**tostring**(*arr*, *max_line_width=75*, *precision=8*, *suppress_small=False*, *separator=' '*, *array_output=0*)

Returns a textual representation of a number or field of numbers. Each dimension is indicated by a pair of matching square brackets (*[]*), within which each subset of the field is output. The orientation of the dimensions is as follows: the last (rightmost) dimension is always horizontal, so that the frequent rank-1 fields use a minimum of screen real-estate. The next-to-last dimesnion 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
```

27.8 parser Module

`fipy.tools.parser.parse`(*larg*, *action=None*, *type=None*, *default=None*)

This is a wrapper function for the python *optparse* module. Unfortunately *optparse* does not allow command line arguments to be ignored. See the documentation for *optparse* for more details. Returns the argument value.

Parameters

- *larg*: 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.

27.9 test Module

27.10 vector Module

Vector utility functions that are inexplicably absent from Numeric

`fipy.tools.vector.putAdd`(*vector*, *ids*, *additionVector*)

This is a temporary replacement for Numeric.put as it was not doing what we thought it was doing.

`fipy.tools.vector.prune`(*array*, *shift*, *start=0*, *axis=0*)

removes elements with indices i = start + shift * n where n = 0, 1, 2, ...

```
>>> 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])
```

27.11 `vitals` Module

class `fipy.tools.vitals.`**`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*)

27.12 Subpackages

27.12.1 comms Package

`commWrapper` Module

class `fipy.tools.comms.commWrapper.`**`CommWrapper`**(*Epetra=None*)
Bases: `object`

MPI Communicator wrapper

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

`Barrier`()

`MaxAll`(*vec*)

`MinAll`(*vec*)

`Norm2`(*vec*)

`Nproc`

`all`(*a*, *axis=None*)

`allclose`(*a*, *b*, *rtol=1e-05*, *atol=1e-08*)

`allequal`(*a*, *b*)

allgather (*sendobj=None*, *recvobj=None*)

any (*a*, *axis=None*)

bcast (*obj=None*, *root=0*)

procID

sum (*a*, *axis=None*)

class fipy.tools.comms.commWrapper.**ParallelCommWrapper** (*Epetra=None*)
Bases: fipy.tools.comms.commWrapper.CommWrapper

MPI Communicator wrapper for parallel processes

dummyComm Module

class fipy.tools.comms.dummyComm.**DummyComm**
Bases: fipy.tools.comms.serialCommWrapper.SerialCommWrapper

Barrier ()

MaxAll (*vec*)

MinAll (*vec*)

sum (*a*, *axis=None*)

mpi4pyCommWrapper Module

class fipy.tools.comms.mpi4pyCommWrapper.**Mpi4pyCommWrapper** (*Epetra*, *MPI*)
Bases: fipy.tools.comms.commWrapper.CommWrapper

MPI Communicator wrapper

Encapsulates capabilities needed for both Epetra and mpi4py.

all (*a*, *axis=None*)

allclose (*a*, *b*, *rtol=1e-05*, *atol=1e-08*)

allequal (*a*, *b*)

allgather (*sendobj=None*, *recvobj=None*)

any (*a*, *axis=None*)

bcast (*obj=None*, *root=0*)

serialCommWrapper Module

class fipy.tools.comms.serialCommWrapper.**SerialCommWrapper** (*Epetra=None*)
Bases: fipy.tools.comms.commWrapper.CommWrapper

Norm2 (*vec*)

Nproc

procID

27.12.2 dimensions Package

`DictWithDefault` Module

`NumberDict` Module

`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 = 0.555555555556 K
   1 degR = 0.555555555556 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 kcali = 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 `fipy.tools.dimensions.physicalField.`**`PhysicalField`**(*value*, *unit=None*, *array=None*)

Bases: `object`

Physical field or quantity with units

Physical Fields can be constructed in one of two ways:

- *PhysicalField(*value*, *unit*)*, where **value** is a number of arbitrary type and **unit** is a string containing the unit name

```
>>> print PhysicalField(value = 10., unit = 'm')
10.0 m
```

- *PhysicalField(*string*)*, where **string** contains both the value and the unit. This form is provided to make interactive use more convenient

```
>>> print PhysicalField(value = "10. m")
10.0 m
```

Dimensionless quantities, with a *unit* of 1, can be specified in several ways

```
>>> print PhysicalField(value = "1")
1.0 1
>>> print PhysicalField(value = 2., unit = " ")
2.0 1
>>> print PhysicalField(value = 2.)
2.0 1
```

Physical arrays are also possible (and are the reason this code was adapted from Konrad 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().allclose("1.57079632679 rad")
1
```

The input *PhysicalField* must be dimensionless

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

arccosh()
Return the inverse hyperbolic cosine of the *PhysicalField*

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

The input *PhysicalField* must be dimensionless

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

arcsin()

Return the inverse sine of the *PhysicalField* in radians

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

The input *PhysicalField* must be dimensionless

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

arctan()

Return the arctangent of the *PhysicalField* in radians

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

The input *PhysicalField* must be dimensionless

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

arctan2(*other*)

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

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

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

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

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

arctanh()

Return the inverse hyperbolic tangent of the *PhysicalField*

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

The input *PhysicalField* must be dimensionless

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

ceil()

Return the smallest integer greater than or equal to the *PhysicalField*.

```
>>> print PhysicalField(2.2,"m").ceil()
3.0 m
```

conjugate()

Return the complex conjugate of the *PhysicalField*.

```
>>> print PhysicalField(2.2 - 3j,"ohm").conjugate() == PhysicalField(2.2 + 3j,"ohm")
True
```

convertToUnit(*unit*)

Changes the unit to *unit* and adjusts the value such that the combination is equivalent. The new unit is by a string containing its name. The new unit must be compatible with the previous unit of the object.

```
>>> e = PhysicalField('2.7 Hartree*Nav')
>>> e.convertToUnit('kcal/mol')
>>> print e
1694.27557621 kcal/mol
```

copy()

Make a duplicate.

```
>>> a = PhysicalField(1, unit = 'inch')
>>> b = a.copy()
```

The duplicate will not reflect changes made to the original

```
>>> a.convertToUnit('cm')
>>> print a
2.54 cm
>>> print b
1 inch
```

Likewise for arrays

```
>>> a = PhysicalField(numerix.array((0,1,2)), unit  = 'm')
>>> b = a.copy()
>>> a[0] = 3
>>> print a
[3 1 2] m
>>> print b
[0 1 2] m
```

cos()

Return the cosine of the *PhysicalField*

```
>>> print numerix.round_(PhysicalField(2*numerix.pi/6,"rad").cos(), 6)
0.5
>>> print numerix.round_(PhysicalField(60.,"deg").cos(), 6)
0.5
```

The units of the *PhysicalField* must be an angle

```
>>> PhysicalField(60.,"m").cos()
Traceback (most recent call last):
    ...
TypeError: Incompatible units
```

cosh()
Return the hyperbolic cosine of the *PhysicalField*

```
>>> PhysicalField(0.).cosh()
1.0
```

The units of the *PhysicalField* must be dimensionless

```
>>> PhysicalField(60.,"m").cosh()
Traceback (most recent call last):
    ...
TypeError: Incompatible units
```

divide(*other*)
Divide two physical quantities. The unit of the result is the unit of the first operand divided by the unit of the second.

```
>>> print PhysicalField(10., 'm') / PhysicalField(2., 's')
5.0 m/s
```

As a special case, if the result is dimensionless, the value is returned without units, rather than with a dimensionless unit of *1*. This facilitates passing physical quantities to packages such as Numeric that cannot use units, while ensuring the quantities have the desired units

```
>>> print (PhysicalField(1., 'inch')
...         / PhysicalField(1., 'mm'))
25.4
```

dot(*other*)
Return the dot product of *self* with *other*. The resulting unit is the product of the units of *self* and *other*.

```
>>> v = PhysicalField(((5.,6.),(7.,8.)), "m")
>>> print PhysicalField(((1.,2.),(3.,4.)), "m").dot(v)
[ 26.  44.] m**2
```

floor()
Return the largest integer less than or equal to the *PhysicalField*.

```
>>> print PhysicalField(2.2,"m").floor()
2.0 m
```

getNumericValue(**args*, ***kwds*)
Deprecated since version 3.0: use the `numericValue` property instead

getShape(**args*, ***kwds*)
Deprecated since version 3.0: use the `shape` property instead

getUnit(**args*, ***kwds*)
Deprecated since version 3.0: use the `unit` property instead

getsctype(*default=None*)
Returns the Numpy sctype of the underlying array.

```
>>> PhysicalField(1, 'm').getsctype() == numerix.NUMERIX.obj2sctype(numerix.array(1))
True
>>> PhysicalField(1., 'm').getsctype() == numerix.NUMERIX.obj2sctype(numerix.array(1.))
True
>>> PhysicalField((1,1.), 'm').getsctype() == numerix.NUMERIX.obj2sctype(numerix.array((1.,
True
```

inBaseUnits()
Return the quantity with all units reduced to their base SI elements.

```
>>> e = PhysicalField('2.7 Hartree*Nav')
>>> print e.inBaseUnits().allclose("7088849.01085 kg*m**2/s**2/mol")
1
```

inDimensionless()

Returns the numerical value of a dimensionless quantity.

```
>>> print PhysicalField(((2.,3.),(4.,5.))).inDimensionless()
[[ 2.  3.]
 [ 4.  5.]]
```

It's an error to convert a quantity with units

```
>>> print PhysicalField(((2.,3.),(4.,5.)),"m").inDimensionless()
Traceback (most recent call last):
    ...
TypeError: Incompatible units
```

inRadians()

Converts an angular quantity to radians and returns the numerical value.

```
>>> print PhysicalField(((2.,3.),(4.,5.)),"rad").inRadians()
[[ 2.  3.]
 [ 4.  5.]]
>>> print PhysicalField(((2.,3.),(4.,5.)),"deg").inRadians()
[[ 0.03490659  0.05235988]
 [ 0.06981317  0.08726646]]
```

As a special case, assumes a dimensionless quantity is already in radians.

```
>>> print PhysicalField(((2.,3.),(4.,5.))).inRadians()
[[ 2.  3.]
 [ 4.  5.]]
```

It's an error to convert a quantity with non-angular units

```
>>> print PhysicalField(((2.,3.),(4.,5.)),"m").inRadians()
Traceback (most recent call last):
    ...
TypeError: Incompatible units
```

inSIUnits()

Return the quantity with all units reduced to SI-compatible elements.

```
>>> e = PhysicalField('2.7 Hartree*Nav')
>>> print e.inSIUnits().allclose("7088849.01085 kg*m**2/s**2/mol")
1
```

inUnitsOf(**units*)

Returns one or more *PhysicalField* objects that express the same physical quantity in different units. The units are specified by strings containing their names. The units must be compatible with the unit of the object. If one unit is specified, the return value is a single *PhysicalField*.

```
>>> freeze = PhysicalField('0 degC')
>>> print freeze.inUnitsOf('degF').allclose("32.0 degF")
1
```

If several units are specified, the return value is a tuple of *PhysicalField* instances with with one element per unit such that the sum of all quantities in the tuple equals the the original quantity and all the values except

for the last one are integers. This is used to convert to irregular unit systems like hour/minute/second. The original object will not be changed.

```
>>> t = PhysicalField(314159., 's')
>>> print numerix.allclose([e.allclose(v) for (e, v) in zip(t.inUnitsOf('d','h','min','s'),
...                                                     ['3.0 d', '15.0 h', '15.0 min',
...                               True)
1
```

isCompatible (*unit*)

itemset (*value*)

Assign the value of a scalar array, performing appropriate conversions.

```
>>> a = PhysicalField(4.,"m")
>>> a.itemset(PhysicalField("6 ft"))
>>> print a.allclose("1.8288 m")
1
>>> a = PhysicalField(((3.,4.),(5.,6.)),"m")
>>> 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
```

itemsize

log ()

Return the natural logarithm of the *PhysicalField*

```
>>> print numerix.round_(PhysicalField(10).log(), 6)
2.302585
```

The input *PhysicalField* must be dimensionless

```
>>> print numerix.round_(PhysicalField("1. m").log(), 6)
Traceback (most recent call last):
    ...
TypeError: Incompatible units
```

log10 ()

Return the base-10 logarithm of the *PhysicalField*

```
>>> print numerix.round_(PhysicalField(10.).log10(), 6)
1.0
```

The input *PhysicalField* must be dimensionless

```
>>> print numerix.round_(PhysicalField("1. m").log10(), 6)
Traceback (most recent call last):
    ...
TypeError: Incompatible units
```

multiply (*other*)

Multiply two physical quantities. The unit of the result is the product of the units of the operands.

```
>>> print PhysicalField(10., 'N') * PhysicalField(10., 'm')
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 *1*. This facilitates passing physical quantities to packages such as Numeric that cannot use units, while ensuring the quantities have the desired units.

```
>>> print (PhysicalField(10., 's') * PhysicalField(2., 'Hz'))
20.0
```

numericValue

Return the *PhysicalField* without units, after conversion to base SI units.

```
>>> print numerix.round_(PhysicalField("1 inch").numericValue, 6)
0.0254
```

put (*indices*, *values*)

put is the opposite of *take*. The values of *self* at the locations specified in *indices* are set to the corresponding value of *values*.

The *indices* can be any integer sequence object with values suitable for indexing into the flat form of *self*. The *values* must be any sequence of values that can be converted to the typecode of *self*.

```
>>> f = PhysicalField((1.,2.,3.),"m")
>>> f.put((2,0), PhysicalField((2.,3.),"inch"))
>>> print f
[ 0.0762  2.      0.0508] m
```

The units of *values* must be compatible with *self*.

```
>>> f.put(1, PhysicalField(3,"kg"))
Traceback (most recent call last):
    ...
TypeError: Incompatible units
```

ravel ()

reshape (*shape*)

Changes the shape of *self* to that specified in *shape*

```
>>> print PhysicalField((1.,2.,3.,4.),"m").reshape((2,2))
[[ 1.  2.]
 [ 3.  4.]] m
```

The new shape must have the same size as the existing one.

```
>>> print PhysicalField((1.,2.,3.,4.),"m").reshape((2,3))
Traceback (most recent call last):
    ...
ValueError: total size of new array must be unchanged
```

setUnit (**args*, ***kwds*)

Deprecated since version 3.0: use the `unit` property instead

shape

Tuple of array dimensions.

sign ()

Return the sign of the quantity. The *unit* is unchanged.

```
>>> from fipy.tools.numerix import sign
>>> print sign(PhysicalField(((3.,-2.),(-1.,4.)), 'm'))
[[ 1. -1.]
 [-1.  1.]]
```

sin()
Return the sine of the *PhysicalField*

```
>>> print PhysicalField(numerix.pi/6,"rad").sin()
0.5
>>> print PhysicalField(30.,"deg").sin()
0.5
```

The units of the PhysicalField must be an angle

```
>>> PhysicalField(30.,"m").sin()
Traceback (most recent call last):
    ...
TypeError: Incompatible units
```

sinh()
Return the hyperbolic sine of the *PhysicalField*

```
>>> PhysicalField(0.).sinh()
0.0
```

The units of the *PhysicalField* must be dimensionless

```
>>> PhysicalField(60.,"m").sinh()
Traceback (most recent call last):
    ...
TypeError: Incompatible units
```

sqrt()
Return the square root of the *PhysicalField*

```
>>> print PhysicalField("100. m**2").sqrt()
10.0 m
```

The resulting unit must be integral

```
>>> print PhysicalField("100. m").sqrt()
Traceback (most recent call last):
    ...
TypeError: Illegal exponent
```

subtract(*other*)
Subtract two physical quantities, so long as their units are compatible. The unit of the result is the unit of the first operand.

```
>>> print PhysicalField(10., 'km') - PhysicalField(10., 'm')
9.99 km
>>> print PhysicalField(10., 'km') - PhysicalField(10., 'J')
Traceback (most recent call last):
    ...
TypeError: Incompatible units
```

sum(*index=0*)
Returns the sum of all of the elements in *self* along the specified axis (first axis by default).

```
>>> print PhysicalField(((1.,2.),(3.,4.)), "m").sum()
[ 4.  6.] m
>>> print PhysicalField(((1.,2.),(3.,4.)), "m").sum(1)
[ 3.  7.] m
```

take (*indices*, *axis=0*)

Return the elements of *self* specified by the elements of *indices*. The resulting *PhysicalField* array has the same units as the original.

```
>>> print PhysicalField((1.,2.,3.),"m").take((2,0))
[ 3.  1.] m
```

The optional third argument specifies the axis along which the selection occurs, and the default value (as in the example above) is 0, the first axis.

```
>>> print PhysicalField(((1.,2.,3.),(4.,5.,6.)),"m").take((2,0), axis = 1)
[[ 3.  1.]
 [ 6.  4.]] m
```

tan ()

Return the tangent of the *PhysicalField*

```
>>> numerix.round_(PhysicalField(numerix.pi/4,"rad").tan(), 6)
1.0
>>> numerix.round_(PhysicalField(45,"deg").tan(), 6)
1.0
```

The units of the *PhysicalField* must be an angle

```
>>> PhysicalField(45.,"m").tan()
Traceback (most recent call last):
    ...
TypeError: Incompatible units
```

tanh ()

Return the hyperbolic tangent of the *PhysicalField*

```
>>> print numerix.allclose(PhysicalField(1.).tanh(), 0.761594155956)
True
```

The units of the *PhysicalField* must be dimensionless

```
>>> PhysicalField(60.,"m").tanh()
Traceback (most recent call last):
    ...
TypeError: Incompatible units
```

tostring (*max_line_width=75*, *precision=8*, *suppress_small=False*, *separator=' '*)

Return human-readable form of a physical quantity

```
>>> p = PhysicalField(value = (3., 3.14159), unit = "eV")
>>> print p.tostring(precision = 3, separator = '|')
[ 3.    | 3.142] eV
```

unit

Return the unit object of *self*.

```
>>> PhysicalField("1 m").unit
<PhysicalUnit m>
```

class fipy.tools.dimensions.physicalField.**PhysicalUnit** (*names*, *factor*, *powers*, *offset=0*)

A *PhysicalUnit* represents the units of a *PhysicalField*.

This class is not generally not instantiated by users of this module, but rather it is created in the process of constructing a *PhysicalField*.

Parameters

- *names*: 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 numerix.round_(b.unit.conversionFactorTo(a.unit), 6)
25.4
```

Units must have the same fundamental SI units

```
>>> c = PhysicalField("1. K")
>>> c.unit.conversionFactorTo(a.unit)
Traceback (most recent call last):
    ...
TypeError: Incompatible units
```

If units have different offsets, they must have the same factor

```
>>> d = PhysicalField("1. degC")
>>> c.unit.conversionFactorTo(d.unit)
1.0
>>> e = PhysicalField("1. degF")
>>> c.unit.conversionFactorTo(e.unit)
Traceback (most recent call last):
    ...
TypeError: Unit conversion (K to degF) cannot be expressed as a simple multiplicative factor
```

conversionTupleTo(*other*)

Return a *tuple* of the multiplication factor and offset between two physical units

```
>>> a = PhysicalField("1. K").unit
>>> b = PhysicalField("1. degF").unit
>>> [str(numerix.round_(element,6)) for element in b.conversionTupleTo(a)]
['0.555556', '459.67']
```

isAngle()

Returns *True* if the unit is an angle

```
>>> PhysicalField("1. deg").unit.isAngle()
1
>>> PhysicalField("1. rad").unit.isAngle()
1
>>> PhysicalField("1. inch").unit.isAngle()
0
```

isCompatible(*other*)

Returns a list of which fundamental SI units are compatible between *self* and *other*

```
>>> a = PhysicalField("1. mm")
>>> b = PhysicalField("1. inch")
```

```
>>> print numerix.allclose(a.unit.isCompatible(b.unit),
...                        [True, True, True, True, True, True, True, True, True])
True
>>> c = PhysicalField("1. K")
>>> print numerix.allclose(a.unit.isCompatible(c.unit),
...                        [False, True, True, True, False, True, True, True, True])
True
```

isDimensionless()

Returns *True* if the unit is dimensionless

```
>>> PhysicalField("1. m/m").unit.isDimensionless()
1
>>> PhysicalField("1. inch").unit.isDimensionless()
0
```

isDimensionlessOrAngle()

Returns *True* if the unit is dimensionless or an angle

```
>>> PhysicalField("1. m/m").unit.isDimensionlessOrAngle()
1
>>> PhysicalField("1. deg").unit.isDimensionlessOrAngle()
1
>>> PhysicalField("1. rad").unit.isDimensionlessOrAngle()
1
>>> PhysicalField("1. inch").unit.isDimensionlessOrAngle()
0
```

isInverseAngle()

Returns *True* if the 1 divided by the unit is an angle

```
>>> PhysicalField("1. deg**-1").unit.isInverseAngle()
1
>>> PhysicalField("1. 1/rad").unit.isInverseAngle()
1
>>> PhysicalField("1. inch").unit.isInverseAngle()
0
```

name()

Return the name of the unit

```
>>> PhysicalField("1. m").unit.name()
'm'
>>> (PhysicalField("1. m") / PhysicalField("1. s")
...  / PhysicalField("1. s")).unit.name()
'm/s**2'
```

setName(*name*)

Set the name of the unit to *name*

```
>>> a = PhysicalField("1. m/s").unit
>>> a
<PhysicalUnit m/s>
>>> a.setName('meterpersecond')
>>> a
<PhysicalUnit meterpersecond>
```

27.12.3 performance Package

efficiencyTestGenerator Module

efficiencyTestHistory Module

efficiency_test Module

class fipy.tools.performance.efficiency_test.**Efficiency_test**(*dist*)
Bases: distutils.cmd.Command

Create and initialize a new Command object. Most importantly, invokes the 'initialize_options()' method, which is the real initializer and depends on the actual command being instantiated.

description = 'run FiPy efficiency tests'

finalize_options()

initialize_options()

run()

user_options = [('minimumelements=', None, 'minimum number of elements'), ('factor=', None, 'factor by which 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.

memoryLogger Module

class fipy.tools.performance.memoryLogger.**MemoryHighWaterThread**(*pid*, *sampleTime=1*)
Bases: threading.Thread

run()

stop()

class fipy.tools.performance.memoryLogger.**MemoryLogger**(*sampleTime=1*)

start()

stop()

memoryUsage Module

This python script is ripped from http://aspn.activestate.com/ASPN/Cookbook/Python/Recipe/286222/index_txt

Chapter 28

variables Package

28.1 variables Package

class fipy.variables.**Variable**(*value=0.0*, *unit=None*, *array=None*, *name=''*, *cached=1*)

Bases: object

Lazily evaluated quantity with units.

Using a Variable in a mathematical expression will create an automatic dependency Variable, e.g.,

```
>>> a = Variable(value=3)
>>> b = 4 * a
>>> b
(Variable(value=array(3)) * 4)
>>> b()
12
```

Changes to the value of a Variable will automatically trigger changes in any dependent Variable objects

```
>>> a.setValue(5)
>>> b
(Variable(value=array(5)) * 4)
>>> print b()
20
```

Create a *Variable*.

```
>>> Variable(value=3)
Variable(value=array(3))
>>> Variable(value=3, unit="m")
Variable(value=PhysicalField(3,'m'))
>>> Variable(value=3, unit="m", array=numerix.zeros((3,2), 'l'))
Variable(value=PhysicalField(array([[3, 3],
       [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=1e-05*, *atol=1e-08*)

```
>>> var = Variable((1, 1))
   >>> print var.allclose((1, 1))
   1
   >>> print var.allclose((1,))
   1
```

The following test is to check that the system does not run out of memory.

```
>>> from fipy.tools import numerix
>>> var = Variable(numerix.ones(10000))
>>> print var.allclose(numerix.zeros(10000, 'l'))
False
```

allequal(*other*)

any(*axis=None*)

```
>>> print Variable(value=0).any()
0
>>> print Variable(value=(0, 0, 1, 1)).any()
1
```

arccos(**args*, ***kwds*)
Deprecated since version 3.0: use `numerix.arccos()` instead

arccosh(**args*, ***kwds*)
Deprecated since version 3.0: use `numerix.arccosh()` instead

arcsin(**args*, ***kwds*)
Deprecated since version 3.0: use `numerix.arcsin()` instead

arcsinh(**args*, ***kwds*)
Deprecated since version 3.0: use `numerix.arcsinh()` instead

arctan(**args*, ***kwds*)
Deprecated since version 3.0: use `numerix.arctan()` instead

arctan2(**args*, ***kwds*)
Deprecated since version 3.0: use `numerix.arctan2()` instead

arctanh(**args*, ***kwds*)
Deprecated since version 3.0: use `numerix.arctanh()` instead

cacheMe(*recursive=False*)

ceil(**args*, ***kwds*)
Deprecated since version 3.0: use `numerix.ceil()` instead

conjugate(**args*, ***kwds*)
Deprecated since version 3.0: use `numerix.conjugate()` instead

constrain(*value*, *where=None*)

Constrain the *Variable* to have a *value* at an index or mask location specified by *where*.

```
>>> v = Variable((0,1,2,3))
>>> v.constrain(2, numerix.array((True, False, False, False)))
>>> print v
[2 1 2 3]
>>> v[:] = 10
>>> print v
[ 2 10 10 10]
>>> v.constrain(5, numerix.array((False, False, True, False)))
>>> print v
[ 2 10  5 10]
>>> v[:] = 6
>>> print v
[2 6 5 6]
>>> v.constrain(8)
>>> print v
[8 8 8 8]
>>> v[:] = 10
>>> print v
[8 8 8 8]
>>> del v.constraints[2]
>>> print v
[ 2 10  5 10]
```

```
>>> from fipy.variables.cellVariable import CellVariable
>>> from fipy.meshes import Grid2D
>>> m = Grid2D(nx=2, ny=2)
>>> x, y = m.cellCenters
>>> v = CellVariable(mesh=m, rank=1, value=(x, y))
>>> v.constrain(((0.,), (-1.,)), where=m.facesLeft)
>>> print v.faceValue
[[ 0.5  1.5  0.5  1.5  0.5  1.5  0.   1.   1.5  0.   1.   1.5]
 [ 0.5  0.5  1.   1.   1.5  1.5 -1.   0.5  0.5 -1.   1.5  1.5]]
```

Parameters

- *value*: the value of the constraint
- *where*: the constraint mask or index specifying the location of the constraint

constraints

copy()

Make an duplicate of the *Variable*

```
>>> a = Variable(value=3)
>>> b = a.copy()
>>> b
Variable(value=array(3))
```

The duplicate will not reflect changes made to the original

```
>>> a.setValue(5)
>>> b
Variable(value=array(3))
```

Check that this works for arrays.

```
>>> a = Variable(value=numerix.array((0,1,2)))
>>> b = a.copy()
>>> b
Variable(value=array([0, 1, 2]))
>>> a[1] = 3
>>> b
Variable(value=array([0, 1, 2]))
```

cos (**args*, ***kwds*)
Deprecated since version 3.0: use `numerix.cos()` instead

cosh (**args*, ***kwds*)
Deprecated since version 3.0: use `numerix.cosh()` instead

dontCacheMe (*recursive=False*)

dot (*other*, *opShape=None*, *operatorClass=None*, *axis=0*)

exp (**args*, ***kwds*)
Deprecated since version 3.0: use `numerix.exp()` instead

floor (**args*, ***kwds*)
Deprecated since version 3.0: use `numerix.floor()` instead

getMag (**args*, ***kwds*)
Deprecated since version 3.0: use the `mag` property instead

getName (**args*, ***kwds*)
Deprecated since version 3.0: use the `name` property instead

getNumericValue (**args*, ***kwds*)
Deprecated since version 3.0: use the `numericValue` property instead

getShape (**args*, ***kwds*)
Deprecated since version 3.0: use the `shape` property instead

getSubscribedVariables (**args*, ***kwds*)
Deprecated since version 3.0: use the `subscribedVariables` property instead

getUnit (**args*, ***kwds*)
Deprecated since version 3.0: use the `unit` property instead

getValue (**args*, ***kwds*)
Deprecated since version 3.0: use the `value` property instead

getsctype (*default=None*)
Returns the Numpy sctype of the underlying array.

```
>>> Variable(1).getsctype() == numerix.NUMERIX.obj2sctype(numerix.array(1))
True
>>> Variable(1.).getsctype() == numerix.NUMERIX.obj2sctype(numerix.array(1.))
True
>>> Variable((1,1.)).getsctype() == numerix.NUMERIX.obj2sctype(numerix.array((1., 1.)))
True
```

inBaseUnits ()
Return the value of the *Variable* with all units reduced to their base SI elements.

```
>>> e = Variable(value="2.7 Hartree*Nav")
>>> print e.inBaseUnits().allclose("7088849.01085 kg*m**2/s**2/mol")
1
```

inUnitsOf (**units*)

Returns one or more *Variable* objects that express the same physical quantity in different units. The units are specified by strings containing their names. The units must be compatible with the unit of the object. If one unit is specified, the return value is a single *Variable*.

```
>>> freeze = Variable('0 degC')
>>> print freeze.inUnitsOf('degF').allclose("32.0 degF")
1
```

If several units are specified, the return value is a tuple of *Variable* instances with with one element per unit such that the sum of all quantities in the tuple equals the the original quantity and all the values except for the last one are integers. This is used to convert to irregular unit systems like hour/minute/second. The original object will not be changed.

```
>>> t = Variable(value=314159., unit='s')
>>> print numerix.allclose([e.allclose(v) for (e, v) in zip(t.inUnitsOf('d','h','min','s'),
...                                                         ['3.0 d', '15.0 h', '15.0 min',
...                                    True)
1
```

itemset (*value*)

itemsize

log (**args*, ***kwds*)

Deprecated since version 3.0: use `numerix.log()` instead

log10 (**args*, ***kwds*)

Deprecated since version 3.0: use `numerix.log10()` instead

mag

max (*axis=None*)

min (*axis=None*)

name

numericValue

put (*indices*, *value*)

ravel ()

release (*constraint*)

Remove *constraint* from *self*

```
>>> v = Variable((0,1,2,3))
>>> v.constrain(2, numerix.array((True, False, False, False)))
>>> v[:] = 10
>>> from fipy.boundaryConditions.constraint import Constraint
>>> c1 = Constraint(5, numerix.array((False, False, True, False)))
>>> v.constrain(c1)
>>> v[:] = 6
>>> v.constrain(8)
>>> v[:] = 10
>>> del v.constraints[2]
>>> v.release(constraint=c1)
>>> print v
[ 2 10 10 10]
```

reshape (**args*, ***kwds*)

Deprecated since version 3.0: use `numerix.reshape()` instead

setName (**args*, ***kwds*)
Deprecated since version 3.0: use the `name` property instead

setUnit (**args*, ***kwds*)
Deprecated since version 3.0: use the `unit` property instead

setValue (*value*, *unit=None*, *where=None*)
Set the value of the Variable. Can take a masked array.

```
>>> a = Variable((1,2,3))
>>> a.setValue(5, where=(1, 0, 1))
>>> print a
[5 2 5]

>>> b = Variable((4,5,6))
>>> a.setValue(b, where=(1, 0, 1))
>>> print a
[4 2 6]
>>> print b
[4 5 6]
>>> a.value = 3
>>> print a
[3 3 3]

>>> b = numerix.array((3,4,5))
>>> a.value = b
>>> a[:] = 1
>>> print b
[3 4 5]

>>> a.setValue((4,5,6), where=(1, 0))
Traceback (most recent call last):
    ....
ValueError: shape mismatch: objects cannot be broadcast to a single shape
```

shape
Tuple of array dimensions.

```
>>> 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,)
```

sign (**args*, ***kwds*)
Deprecated since version 3.0: use `numerix.sign()` instead

sin (**args*, ***kwds*)
Deprecated since version 3.0: use `numerix.sin()` instead

sinh (**args*, ***kwds*)
Deprecated since version 3.0: use `numerix.sinh()` instead

sqrt (**args*, ***kwds*)
Deprecated since version 3.0: use `numerix.sqrt()` instead

```
>>> from fipy.meshes 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.]
```

subscribedVariables

sum (*axis=None*)

take (*ids*, *axis=0*)

tan (**args*, ***kwds*)
Deprecated since version 3.0: use `numerix.tan()` instead

tanh (**args*, ***kwds*)
Deprecated since version 3.0: use `numerix.tanh()` instead

tostring (*max_line_width=75*, *precision=8*, *suppress_small=False*, *separator=' '*)

unit
Return the unit object of *self*.

```
>>> Variable(value="1 m").unit
<PhysicalUnit m>
```

value
"Evaluate" the *Variable* and return its value (longhand)

```
>>> a = Variable(value=3)
>>> print a.value
3
>>> b = a + 4
>>> b
(Variable(value=array(3)) + 4)
>>> b.value
7
```

class `fipy.variables.`**CellVariable** (*mesh*, *name=''*, *value=0.0*, *rank=None*, *elementshape=None*, *unit=None*, *hasOld=0*)
Bases: `fipy.variables.meshVariable._MeshVariable`

Represents the field of values of a variable on a *Mesh*.

A *CellVariable* can be `pickled` to persistent storage (disk) for later use:

```
>>> from fipy.meshes import Grid2D
>>> mesh = Grid2D(dx = 1., dy = 1., nx = 10, ny = 10)

>>> var = CellVariable(mesh = mesh, value = 1., hasOld = 1, name = 'test')
>>> x, y = mesh.cellCenters
>>> var.value = (x * y)

>>> from fipy.tools import dump
>>> (f, filename) = dump.write(var, extension = '.gz')
>>> unPickledVar = dump.read(filename, f)
```

```
>>> print var.allclose(unPickledVar, atol = 1e-10, rtol = 1e-10)
1
```

arithmeticFaceValue

Returns a *FaceVariable* whose value corresponds to the arithmetic interpolation of the adjacent cells:

$$\phi_f = (\phi_1 - \phi_2)\frac{d_{f2}}{d_{12}} + \phi_2$$

```
>>> from fipy.meshes import Grid1D
>>> from fipy import numerix
>>> mesh = Grid1D(dx = (1., 1.))
>>> L = 1
>>> R = 2
>>> var = CellVariable(mesh = mesh, value = (L, R))
>>> faceValue = var.arithmeticFaceValue[mesh.interiorFaces.value]
>>> answer = (R - L) * (0.5 / 1.) + L
>>> print numerix.allclose(faceValue, answer, atol = 1e-10, rtol = 1e-10)
True
```

```
>>> mesh = Grid1D(dx = (2., 4.))
>>> var = CellVariable(mesh = mesh, value = (L, R))
>>> faceValue = var.arithmeticFaceValue[mesh.interiorFaces.value]
>>> answer = (R - L) * (1.0 / 3.0) + L
>>> print numerix.allclose(faceValue, answer, atol = 1e-10, rtol = 1e-10)
True
```

```
>>> mesh = Grid1D(dx = (10., 100.))
>>> var = CellVariable(mesh = mesh, value = (L, R))
>>> faceValue = var.arithmeticFaceValue[mesh.interiorFaces.value]
>>> answer = (R - L) * (5.0 / 55.0) + L
>>> print numerix.allclose(faceValue, answer, atol = 1e-10, rtol = 1e-10)
True
```

cellVolumeAverage

Return the cell-volume-weighted average of the *CellVariable*:

$$< \phi >_\text{vol} = \frac{\sum_\text{cells} \phi_\text{cell} V_\text{cell}}{\sum_\text{cells} V_\text{cell}}$$

```
>>> from fipy.meshes import Grid2D
>>> from fipy.variables.cellVariable import CellVariable
>>> mesh = Grid2D(nx = 3, ny = 1, dx = .5, dy = .1)
>>> var = CellVariable(value = (1, 2, 6), mesh = mesh)
>>> print var.cellVolumeAverage
3.0
```

constrain (*value*, *where=None*)

Constrains the *CellVariable* to *value* at a location specified by *where*.

```
>>> from fipy import *
>>> m = Grid1D(nx=3)
>>> v = CellVariable(mesh=m, value=m.cellCenters[0])
>>> v.constrain(0., where=m.facesLeft)
>>> v.faceGrad.constrain([1.], where=m.facesRight)
>>> print v.faceGrad
[[ 1.  1.  1.  1.]]
```

```
>>> print v.faceValue
[ 0.   1.   2.   2.5]
```

Changing the constraint changes the dependencies

```
>>> v.constrain(1., where=m.facesLeft)
>>> print v.faceGrad
[[-1.  1.  1.  1.]]
>>> print v.faceValue
[ 1.   1.   2.   2.5]
```

Constraints can be *Variable*

```
>>> c = Variable(0.)
>>> v.constrain(c, where=m.facesLeft)
>>> print v.faceGrad
[[ 1.  1.  1.  1.]]
>>> print v.faceValue
[ 0.   1.   2.   2.5]
>>> c.value = 1.
>>> print v.faceGrad
[[-1.  1.  1.  1.]]
>>> print v.faceValue
[ 1.   1.   2.   2.5]
```

Constraints can have a *Variable* mask.

```
>>> v = CellVariable(mesh=m)
>>> mask = FaceVariable(mesh=m, value=m.facesLeft)
>>> v.constrain(1., where=mask)
>>> print v.faceValue
[ 1.  0.  0.  0.]
>>> mask[:] = mask | m.facesRight
>>> print v.faceValue
[ 1.  0.  0.  1.]
```

copy()

faceGrad
Return $\nabla \phi$ as a rank-1 *FaceVariable* using differencing for the normal direction(second-order gradient).

faceGradAverage
Return $\nabla \phi$ as a rank-1 *FaceVariable* using averaging for the normal direction(second-order gradient)

faceValue
Returns a *FaceVariable* whose value corresponds to the arithmetic interpolation of the adjacent cells:

$$\phi_f = (\phi_1 - \phi_2)\frac{d_{f2}}{d_{12}} + \phi_2$$

```
>>> from fipy.meshes import Grid1D
>>> from fipy import numerix
>>> mesh = Grid1D(dx = (1., 1.))
>>> L = 1
>>> R = 2
>>> var = CellVariable(mesh = mesh, value = (L, R))
>>> faceValue = var.arithmeticFaceValue[mesh.interiorFaces.value]
>>> answer = (R - L) * (0.5 / 1.) + L
>>> print numerix.allclose(faceValue, answer, atol = 1e-10, rtol = 1e-10)
True
```

```
>>> mesh = Grid1D(dx = (2., 4.))
>>> var = CellVariable(mesh = mesh, value = (L, R))
>>> faceValue = var.arithmeticFaceValue[mesh.interiorFaces.value]
>>> answer = (R - L) * (1.0 / 3.0) + L
>>> print numerix.allclose(faceValue, answer, atol = 1e-10, rtol = 1e-10)
True

>>> mesh = Grid1D(dx = (10., 100.))
>>> var = CellVariable(mesh = mesh, value = (L, R))
>>> faceValue = var.arithmeticFaceValue[mesh.interiorFaces.value]
>>> answer = (R - L) * (5.0 / 55.0) + L
>>> print numerix.allclose(faceValue, answer, atol = 1e-10, rtol = 1e-10)
True
```

gaussGrad
Return $\frac{1}{V_P} \sum_f \vec{n} \phi_f A_f$ as a rank-1 *CellVariable* (first-order gradient).

getArithmeticFaceValue(**args*, ***kwds*)
Deprecated since version 3.0: use the `arithmeticFaceValue` property instead

getCellVolumeAverage(**args*, ***kwds*)
Deprecated since version 3.0: use the `cellVolumeAverage` property instead

getFaceGrad(**args*, ***kwds*)
Deprecated since version 3.0: use the `faceGrad` property instead

getFaceGradAverage(**args*, ***kwds*)
Deprecated since version 3.0: use the `faceGradAverage` property instead

getFaceValue(**args*, ***kwds*)
Deprecated since version 3.0: use the `arithmeticFaceValue` property instead

getGaussGrad(**args*, ***kwds*)
Deprecated since version 3.0: use the `gaussGrad` property instead

getGrad(**args*, ***kwds*)
Deprecated since version 3.0: use the `grad` property instead

getHarmonicFaceValue(**args*, ***kwds*)
Deprecated since version 3.0: use the `harmonicFaceValue` property instead

getLeastSquaresGrad(**args*, ***kwds*)
Deprecated since version 3.0: use the `leastSquaresGrad` property instead

getMinmodFaceValue(**args*, ***kwds*)
Deprecated since version 3.0: use the `minmodFaceValue` property instead

getOld(**args*, ***kwds*)
Deprecated since version 3.0: use the `old` property instead

globalValue
Concatenate and return values from all processors

When running on a single processor, the result is identical to `value`.

grad
Return $\nabla \phi$ as a rank-1 *CellVariable* (first-order gradient).

harmonicFaceValue

Returns a *FaceVariable* whose value corresponds to the harmonic interpolation of the adjacent cells:

$$\phi_f = \frac{\phi_1 \phi_2}{(\phi_2 - \phi_1) \frac{d_{f2}}{d_{12}} + \phi_1}$$

```
>>> from fipy.meshes import Grid1D
>>> from fipy import numerix
>>> mesh = Grid1D(dx = (1., 1.))
>>> L = 1
>>> R = 2
>>> var = CellVariable(mesh = mesh, value = (L, R))
>>> faceValue = var.harmonicFaceValue[mesh.interiorFaces.value]
>>> answer = L * R / ((R - L) * (0.5 / 1.) + L)
>>> print numerix.allclose(faceValue, answer, atol = 1e-10, rtol = 1e-10)
True

>>> mesh = Grid1D(dx = (2., 4.))
>>> var = CellVariable(mesh = mesh, value = (L, R))
>>> faceValue = var.harmonicFaceValue[mesh.interiorFaces.value]
>>> answer = L * R / ((R - L) * (1.0 / 3.0) + L)
>>> print numerix.allclose(faceValue, answer, atol = 1e-10, rtol = 1e-10)
True

>>> mesh = Grid1D(dx = (10., 100.))
>>> var = CellVariable(mesh = mesh, value = (L, R))
>>> faceValue = var.harmonicFaceValue[mesh.interiorFaces.value]
>>> answer = L * R / ((R - L) * (5.0 / 55.0) + L)
>>> print numerix.allclose(faceValue, answer, atol = 1e-10, rtol = 1e-10)
True
```

leastSquaresGrad

Return $\nabla \phi$, which is determined by solving for $\nabla \phi$ in the following matrix equation,

$$\nabla \phi \cdot \sum_f d_{AP}^2 \vec{n}_{AP} \otimes \vec{n}_{AP} = \sum_f d_{AP}^2 \left(\vec{n} \cdot \nabla \phi \right)_{AP}$$

The matrix equation is derived by minimizing the following least squares sum,

$$F\left(\phi_x, \phi_y\right) = \sqrt{\sum_f \left(d_{AP} \vec{n}_{AP} \cdot \nabla \phi - d_{AP} \left(\vec{n}_{AP} \cdot \nabla \phi \right)_{AP} \right)^2}$$

Tests

```
>>> from fipy import Grid2D
>>> m = Grid2D(nx=2, ny=2, dx=0.1, dy=2.0)
>>> print numerix.allclose(CellVariable(mesh=m, value=(0,1,3,6)).leastSquaresGrad.globalValu
...                                     [[8.0, 8.0, 24.0, 24.0],
...                                      [1.2, 2.0, 1.2, 2.0]])
True

>>> from fipy import Grid1D
>>> print numerix.allclose(CellVariable(mesh=Grid1D(dx=(2.0, 1.0, 0.5)),
...                                     value=(0, 1, 2)).leastSquaresGrad.globalValue, [[0.4
True
```

minmodFaceValue

Returns a *FaceVariable* with a value that is the minimum of the absolute values of the adjacent cells. If the

values are of opposite sign then the result is zero:

$$\phi_f = \begin{cases} \phi_1 & \text{when } |\phi_1| \le |\phi_2|, \\ \phi_2 & \text{when } |\phi_2| < |\phi_1|, \\ 0 & \text{when } \phi 1 \phi 2 < 0 \end{cases}$$

```
>>> from fipy import *
>>> print CellVariable(mesh=Grid1D(nx=2), value=(1, 2)).minmodFaceValue
[1 1 2]
>>> print CellVariable(mesh=Grid1D(nx=2), value=(-1, -2)).minmodFaceValue
[-1 -1 -2]
>>> print CellVariable(mesh=Grid1D(nx=2), value=(-1, 2)).minmodFaceValue
[-1  0  2]
```

old

Return the values of the *CellVariable* from the previous solution sweep.

Combinations of *CellVariable's* should also return old values.

```
>>> from fipy.meshes import Grid1D
>>> mesh = Grid1D(nx = 2)
>>> from fipy.variables.cellVariable import CellVariable
>>> var1 = CellVariable(mesh = mesh, value = (2, 3), hasOld = 1)
>>> var2 = CellVariable(mesh = mesh, value = (3, 4))
>>> v = var1 * var2
>>> print v
[ 6 12]
>>> var1.value = ((3,2))
>>> print v
[9 8]
>>> print v.old
[ 6 12]
```

The following small test is to correct for a bug when the operator does not just use variables.

```
>>> v1 = var1 * 3
>>> print v1
[9 6]
>>> print v1.old
[6 9]
```

release (*constraint*)

Remove *constraint* from *self*

```
>>> from fipy import *
>>> m = Grid1D(nx=3)
>>> v = CellVariable(mesh=m, value=m.cellCenters[0])
>>> c = Constraint(0., where=m.facesLeft)
>>> v.constrain(c)
>>> print v.faceValue
[ 0.   1.   2.   2.5]
>>> v.release(constraint=c)
>>> print v.faceValue
[ 0.5  1.   2.   2.5]
```

setValue (*value*, *unit=None*, *where=None*)

updateOld ()

Set the values of the previous solution sweep to the current values.

```
>>> from fipy import *
>>> v = CellVariable(mesh=Grid1D(), hasOld=False)
>>> v.updateOld()
Traceback (most recent call last):
    ...
AssertionError: The updateOld method requires the CellVariable to have an old value. Set has
```

class `fipy.variables.`**`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 * (mesh.dim,)*
- *unit*: the physical units of the *Variable*

`copy`()

`divergence`

```
>>> from fipy.meshes import Grid2D
>>> from fipy.variables.cellVariable import CellVariable
>>> mesh = Grid2D(nx=3, ny=2)
>>> var = CellVariable(mesh=mesh, value=range(3*2))
>>> print var.faceGrad.divergence
[ 4.  3.  2. -2. -3. -4.]
```

`getDivergence`(**args*, ***kwds*)

Deprecated since version 3.0: use the `divergence` property instead

`globalValue`

`setValue`(*value*, *unit=None*, *where=None*)

class `fipy.variables.`**`ScharfetterGummelFaceVariable`**(*var*, *boundaryConditions=()*)

Bases: `fipy.variables.cellToFaceVariable._CellToFaceVariable`

class `fipy.variables.`**`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 exisits on the circle between $-\pi$ and π

The following examples show how *ModularVariable* works. When subtracting the answer wraps back around the circle.

```
>>> from fipy.meshes import Grid1D
>>> mesh = Grid1D(nx = 2)
>>> from fipy.tools import numerix
>>> pi = numerix.pi
>>> v1 = ModularVariable(mesh = mesh, value = (2*pi/3, -2*pi/3))
>>> v2 = ModularVariable(mesh = mesh, value = -2*pi/3)
>>> print numerix.allclose(v2 - v1, (2*pi/3, 0))
1
```

Obtaining the arithmetic face value.

```
>>> print numerix.allclose(v1.arithmeticFaceValue, (2*pi/3, pi, -2*pi/3))
1
```

Obtaining the gradient.

```
>>> print numerix.allclose(v1.grad, ((pi/3, pi/3),))
1
```

Obtaining the gradient at the faces.

```
>>> print numerix.allclose(v1.faceGrad, ((0, 2*pi/3, 0),))
1
```

Obtaining the gradient at the faces but without modular arithmetic.

```
>>> print numerix.allclose(v1.faceGradNoMod, ((0, -4*pi/3, 0),))
1
```

arithmeticFaceValue

Returns a *FaceVariable* whose value corresponds to the arithmetic interpolation of the adjacent cells:

$$\phi_f = (\phi_1 - \phi_2)\frac{d_{f2}}{d_{12}} + \phi_2$$

Adjusted for a *ModularVariable*

faceGrad

Return $\nabla \phi$ as a rank-1 *FaceVariable* (second-order gradient). Adjusted for a *ModularVariable*

faceGradNoMod

Return $\nabla \phi$ as a rank-1 *FaceVariable* (second-order gradient). Not adjusted for a *ModularVariable*

getFaceGradNoMod(**args*, ***kwds*)

Deprecated since version 3.0: use the `faceGradNoMod` property instead

grad

Return $\nabla \phi$ as a rank-1 *CellVariable* (first-order gradient). Adjusted for a *ModularVariable*

updateOld()

Set the values of the previous solution sweep to the current values. Test case due to bug.

```
>>> from fipy.meshes import Grid1D
>>> mesh = Grid1D(nx = 1)
>>> var = ModularVariable(mesh=mesh, value=1., hasOld=1)
>>> var.updateOld()
>>> var[:] = 2
>>> answer = CellVariable(mesh=mesh, value=1.)
>>> print var.old.allclose(answer)
True
```

class fipy.variables.**BetaNoiseVariable**(*mesh*, *alpha*, *beta*, *name=''*, *hasOld=0*)

Bases: `fipy.variables.noiseVariable.NoiseVariable`

Represents a beta distribution of random numbers with the probability distribution

$$x^{\alpha-1}\frac{\beta^\alpha e^{-\beta x}}{\Gamma(\alpha)}$$

with a shape parameter α, a rate parameter β, and $\Gamma(z) = \int_0^\infty t^{z-1}e^{-t}\,dt$.

Seed the random module for the sake of deterministic test results.

```
>>> from fipy import numerix
>>> numerix.random.seed(1)
```

We generate noise on a uniform cartesian mesh

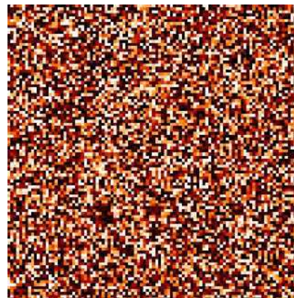
```
>>> from fipy.variables.variable import Variable
>>> alpha = Variable()
>>> beta = Variable()
>>> from fipy.meshes import Grid2D
>>> noise = BetaNoiseVariable(mesh = Grid2D(nx = 100, ny = 100), alpha = alpha, beta = beta)
```

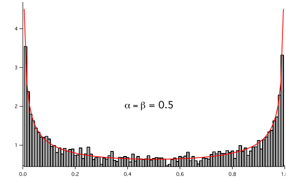
We histogram the root-volume-weighted noise distribution

```
>>> from fipy.variables.histogramVariable import HistogramVariable
>>> histogram = HistogramVariable(distribution = noise, dx = 0.01, nx = 100)
```

and compare to a Gaussian distribution

```
>>> from fipy.variables.cellVariable import CellVariable
>>> betadist = CellVariable(mesh = histogram.mesh)
>>> x = CellVariable(mesh=histogram.mesh, value=histogram.mesh.cellCenters[0])
>>> from scipy.special import gamma as Gamma
>>> betadist = ((Gamma(alpha + beta) / (Gamma(alpha) * Gamma(beta)))
...             * x**(alpha - 1) * (1 - x)**(beta - 1))

>>> if __name__ == '__main__':
...     from fipy import Viewer
...     viewer = Viewer(vars=noise, datamin=0, datamax=1)
...     histoplot = Viewer(vars=(histogram, betadist),
...                        datamin=0, datamax=1.5)

>>> from fipy.tools.numerix import arange

>>> for a in arange(0.5,5,0.5):
...     alpha.value = a
...     for b in arange(0.5,5,0.5):
...         beta.value = b
...         if __name__ == '__main__':
...             import sys
...             print >>sys.stderr, "alpha: %g, beta: %g" % (alpha, beta)
...             viewer.plot()
...             histoplot.plot()

>>> print abs(noise.faceGrad.divergence.cellVolumeAverage) < 5e-15
1
```





Parameters

- *mesh*: The mesh on which to define the noise.
- *alpha*: The parameter α.
- *beta*: The parameter β.

random()

class fipy.variables.**ExponentialNoiseVariable**(*mesh*, *mean=0.0*, *name=''*, *hasOld=0*)

Bases: fipy.variables.noiseVariable.NoiseVariable

Represents an exponential distribution of random numbers with the probability distribution

$$\mu^{-1}e^{-\frac{x}{\mu}}$$

with a mean parameter μ.

Seed the random module for the sake of deterministic test results.

```
>>> from fipy import numerix
>>> numerix.random.seed(1)
```

We generate noise on a uniform cartesian mesh

```
>>> from fipy.variables.variable import Variable
>>> mean = Variable()
>>> from fipy.meshes import Grid2D
>>> noise = ExponentialNoiseVariable(mesh = Grid2D(nx = 100, ny = 100), mean = mean)
```

We histogram the root-volume-weighted noise distribution

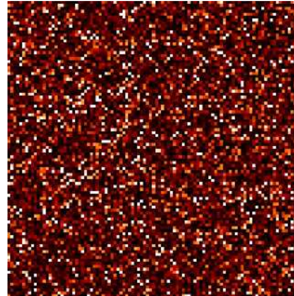
```
>>> from fipy.variables.histogramVariable import HistogramVariable
>>> histogram = HistogramVariable(distribution = noise, dx = 0.1, nx = 100)
```

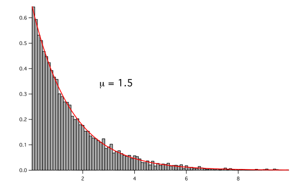
and compare to a Gaussian distribution

```
>>> from fipy.variables.cellVariable import CellVariable
>>> expdist = CellVariable(mesh = histogram.mesh)
>>> x = histogram.mesh.cellCenters[0]

>>> if __name__ == '__main__':
...     from fipy import Viewer
...     viewer = Viewer(vars=noise, datamin=0, datamax=5)
...     histoplot = Viewer(vars=(histogram, expdist),
...                        datamin=0, datamax=1.5)

>>> from fipy.tools.numerix import arange, exp

>>> for mu in arange(0.5,3,0.5):
...     mean.value = (mu)
...     expdist.value = ((1/mean)*exp(-x/mean))
...     if __name__ == '__main__':
...         import sys
```

```
...         print >>sys.stderr, "mean: %g" % mean
...         viewer.plot()
...         histoplot.plot()
```

```
>>> print abs(noise.faceGrad.divergence.cellVolumeAverage) < 5e-15
1
```





Parameters

- *mesh*: The mesh on which to define the noise.
- *mean*: The mean of the distribution μ.

random()

class fipy.variables.**GammaNoiseVariable**(*mesh*, *shape*, *rate*, *name=''*, *hasOld=0*)
Bases: fipy.variables.noiseVariable.NoiseVariable

Represents a gamma distribution of random numbers with the probability distribution

$$x^{\alpha-1}\frac{\beta^\alpha e^{-\beta x}}{\Gamma(\alpha)}$$

with a shape parameter α, a rate parameter β, and $\Gamma(z) = \int_0^\infty t^{z-1}e^{-t}\,dt$.

Seed the random module for the sake of deterministic test results.

```
>>> from fipy import numerix
>>> numerix.random.seed(1)
```

We generate noise on a uniform cartesian mesh

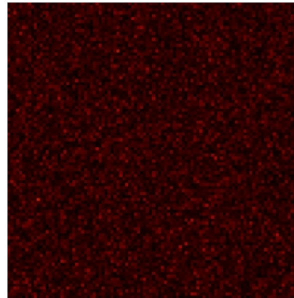
```
>>> from fipy.variables.variable import Variable
>>> alpha = Variable()
>>> beta = Variable()
>>> from fipy.meshes import Grid2D
>>> noise = GammaNoiseVariable(mesh = Grid2D(nx = 100, ny = 100), shape = alpha, rate = beta)
```

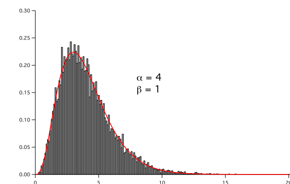
We histogram the root-volume-weighted noise distribution

```
>>> from fipy.variables.histogramVariable import HistogramVariable
>>> histogram = HistogramVariable(distribution = noise, dx = 0.1, nx = 300)
```

and compare to a Gaussian distribution

```
>>> from fipy.variables.cellVariable import CellVariable
>>> x = CellVariable(mesh=histogram.mesh, value=histogram.mesh.cellCenters[0])
>>> from scipy.special import gamma as Gamma
>>> from fipy.tools.numerix import exp
>>> gammadist = (x**(alpha - 1) * (beta**alpha * exp(-beta * x)) / Gamma(alpha))

>>> if __name__ == '__main__':
...     from fipy import Viewer
...     viewer = Viewer(vars=noise, datamin=0, datamax=30)
...     histoplot = Viewer(vars=(histogram, gammadist),
...                        datamin=0, datamax=1)

>>> from fipy.tools.numerix import arange

>>> for shape in arange(1,8,1):
...     alpha.value = shape
...     for rate in arange(0.5,2.5,0.5):
...         beta.value = rate
...         if __name__ == '__main__':
...             import sys
...             print >>sys.stderr, "alpha: %g, beta: %g" % (alpha, beta)
...             viewer.plot()
...             histoplot.plot()

>>> print abs(noise.faceGrad.divergence.cellVolumeAverage) < 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()

class fipy.variables.**GaussianNoiseVariable**(*mesh*, *name=''*, *mean=0.0*, *variance=1.0*, *hasOld=0*)

Bases: fipy.variables.noiseVariable.NoiseVariable

Represents a normal (Gaussian) distribution of random numbers with mean μ and variance $\langle\eta(\vec{r})\eta(\vec{r}\,')\rangle = \sigma^2$, which has the probability distribution

$$\frac{1}{\sigma\sqrt{2\pi}} \exp -\frac{(x-\mu)^2}{2\sigma^2}$$

For example, the variance of thermal noise that is uncorrelated in space and time is often expressed as

$$\langle\eta(\vec{r},t)\eta(\vec{r}\,',t')\rangle = M k_B T \delta(\vec{r}-\vec{r}\,')\delta(t-t')$$

which can be obtained with:

```
sigmaSqrd = Mobility * kBoltzmann * Temperature / (mesh.cellVolumes * timeStep)
GaussianNoiseVariable(mesh = mesh, variance = sigmaSqrd)
```

Note: If the time step will change as the simulation progresses, either through use of an adaptive iterator or by making manual changes at different stages, remember to declare *timeStep* as a *Variable* and to change its value with its *setValue()* method.

```
>>> import sys
>>> from fipy.tools.numerix import *

>>> mean = 0.
>>> variance = 4.
```

Seed the random module for the sake of deterministic test results.

```
>>> from fipy import numerix
>>> numerix.random.seed(3)
```

We generate noise on a non-uniform cartesian mesh with cell dimensions of x^2 and y^3.

```
>>> from fipy.meshes import Grid2D
>>> mesh = Grid2D(dx = arange(0.1, 5., 0.1)**2, dy = arange(0.1, 3., 0.1)**3)
>>> from fipy.variables.cellVariable import CellVariable
>>> volumes = CellVariable(mesh=mesh,value=mesh.cellVolumes)
>>> noise = GaussianNoiseVariable(mesh = mesh, mean = mean,
...                               variance = variance / volumes)
```

We histogram the root-volume-weighted noise distribution

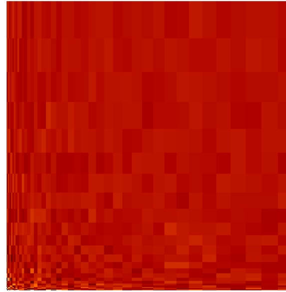
```
>>> from fipy.variables.histogramVariable import HistogramVariable
>>> histogram = HistogramVariable(distribution = noise * sqrt(volumes),
...                               dx = 0.1, nx = 600, offset = -30)
```

and compare to a Gaussian distribution

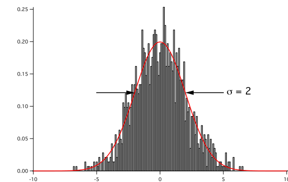
```
>>> gauss = CellVariable(mesh = histogram.mesh)
>>> x = histogram.mesh.cellCenters[0]
>>> gauss.value = ((1/(sqrt(variance * 2 * pi))) * exp(-(x - mean)**2 / (2 * variance)))

>>> if __name__ == '__main__':
...     from fipy.viewers import Viewer
...     viewer = Viewer(vars=noise,
...                     datamin=-5, datamax=5)
...     histoplot = Viewer(vars=(histogram, gauss))
```

```
>>> for i in range(10):
...     noise.scramble()
...     if __name__ == '__main__':
...         viewer.plot()
...         histoplot.plot()

>>> print abs(noise.faceGrad.divergence.cellVolumeAverage) < 5e-15
1
```

Note that the noise exhibits larger amplitude in the small cells than in the large ones



but that the root-volume-weighted histogram is Gaussian.



Parameters

- *mesh*: The mesh on which to define the noise.
- *mean*: The mean of the noise distrubution, μ.
- *variance*: The variance of the noise distribution, σ^2.

parallelRandom()

class fipy.variables.**UniformNoiseVariable**(*mesh*, *name=''*, *minimum=0.0*, *maximum=1.0*, *hasOld=0*)

Bases: fipy.variables.noiseVariable.NoiseVariable

Represents a uniform distribution of random numbers.

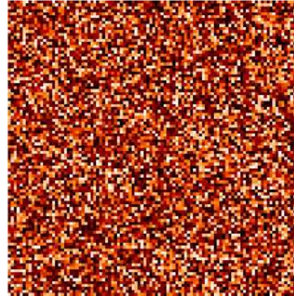
We generate noise on a uniform cartesian mesh

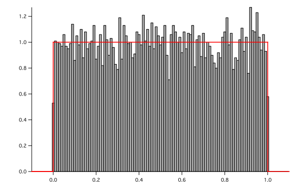
```
>>> from fipy.meshes import Grid2D
>>> noise = UniformNoiseVariable(mesh=Grid2D(nx=100, ny=100))
```

and histogram the noise

```
>>> from fipy.variables.histogramVariable import HistogramVariable
>>> histogram = HistogramVariable(distribution=noise, dx=0.01, nx=120, offset=-.1)

>>> if __name__ == '__main__':
...     from fipy import Viewer
...     viewer = Viewer(vars=noise,
...                     datamin=0, datamax=1)
...     histoplot = Viewer(vars=histogram)
```

```
>>> 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()

class fipy.variables.**HistogramVariable**(*distribution*, *dx=1.0*, *nx=None*, *offset=0.0*)
Bases: fipy.variables.cellVariable.CellVariable

Produces a histogram of the values of the supplied distribution.

Parameters

- *distribution*: The collection of values to sample.
- *dx*: the bin size
- *nx*: the number of bins
- *offset*: the position of the first bin

28.2 addOverFacesVariable Module

28.3 arithmeticCellToFaceVariable Module

28.4 betaNoiseVariable Module

class fipy.variables.betaNoiseVariable.**BetaNoiseVariable**(*mesh*, *alpha*, *beta*, *name=''*, *hasOld=0*)
Bases: fipy.variables.noiseVariable.NoiseVariable

Represents a beta distribution of random numbers with the probability distribution

$$x^{\alpha-1}\frac{\beta^\alpha e^{-\beta x}}{\Gamma(\alpha)}$$

with a shape parameter α, a rate parameter β, and $\Gamma(z) = \int_0^\infty t^{z-1}e^{-t}\,dt$.

Seed the random module for the sake of deterministic test results.

```
>>> from fipy import numerix
>>> numerix.random.seed(1)
```

We generate noise on a uniform cartesian mesh

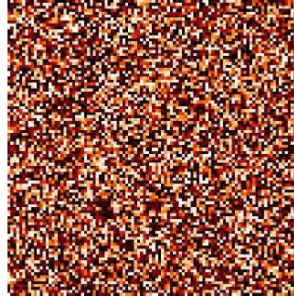
```
>>> from fipy.variables.variable import Variable
>>> alpha = Variable()
>>> beta = Variable()
>>> from fipy.meshes import Grid2D
>>> noise = BetaNoiseVariable(mesh = Grid2D(nx = 100, ny = 100), alpha = alpha, beta = beta)
```

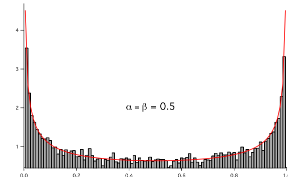
We histogram the root-volume-weighted noise distribution

```
>>> from fipy.variables.histogramVariable import HistogramVariable
>>> histogram = HistogramVariable(distribution = noise, dx = 0.01, nx = 100)
```

and compare to a Gaussian distribution

```
>>> from fipy.variables.cellVariable import CellVariable
>>> betadist = CellVariable(mesh = histogram.mesh)
>>> x = CellVariable(mesh=histogram.mesh, value=histogram.mesh.cellCenters[0])
>>> from scipy.special import gamma as Gamma
>>> betadist = ((Gamma(alpha + beta) / (Gamma(alpha) * Gamma(beta)))
...             * x**(alpha - 1) * (1 - x)**(beta - 1))

>>> if __name__ == '__main__':
...     from fipy import Viewer
...     viewer = Viewer(vars=noise, datamin=0, datamax=1)
...     histoplot = Viewer(vars=(histogram, betadist),
...                        datamin=0, datamax=1.5)

>>> from fipy.tools.numerix import arange

>>> for a in arange(0.5,5,0.5):
...     alpha.value = a
...     for b in arange(0.5,5,0.5):
...         beta.value = b
...         if __name__ == '__main__':
...             import sys
...             print >>sys.stderr, "alpha: %g, beta: %g" % (alpha, beta)
...             viewer.plot()
...             histoplot.plot()

>>> print abs(noise.faceGrad.divergence.cellVolumeAverage) < 5e-15
1
```





Parameters

- *mesh*: The mesh on which to define the noise.
- *alpha*: The parameter α.
- *beta*: The parameter β.

random()

28.5 `binaryOperatorVariable` Module

28.6 `cellToFaceVariable` Module

28.7 `cellVariable` Module

class `fipy.variables.cellVariable.`**`CellVariable`**(*mesh*, *name=''*, *value=0.0*, *rank=None*, *elementshape=None*, *unit=None*, *hasOld=0*)

Bases: `fipy.variables.meshVariable._MeshVariable`

Represents the field of values of a variable on a *Mesh*.

A *CellVariable* can be `pickled` to persistent storage (disk) for later use:

```
>>> from fipy.meshes import Grid2D
>>> mesh = Grid2D(dx = 1., dy = 1., nx = 10, ny = 10)

>>> var = CellVariable(mesh = mesh, value = 1., hasOld = 1, name = 'test')
>>> x, y = mesh.cellCenters
>>> var.value = (x * y)

>>> from fipy.tools import dump
>>> (f, filename) = dump.write(var, extension = '.gz')
>>> unPickledVar = dump.read(filename, f)

>>> print var.allclose(unPickledVar, atol = 1e-10, rtol = 1e-10)
1
```

arithmeticFaceValue

Returns a *FaceVariable* whose value corresponds to the arithmetic interpolation of the adjacent cells:

$$\phi_f = (\phi_1 - \phi_2)\frac{d_{f2}}{d_{12}} + \phi_2$$

```
>>> from fipy.meshes import Grid1D
>>> from fipy import numerix
>>> mesh = Grid1D(dx = (1., 1.))
>>> L = 1
>>> R = 2
>>> var = CellVariable(mesh = mesh, value = (L, R))
>>> faceValue = var.arithmeticFaceValue[mesh.interiorFaces.value]
>>> answer = (R - L) * (0.5 / 1.) + L
>>> print numerix.allclose(faceValue, answer, atol = 1e-10, rtol = 1e-10)
True
```

```
>>> mesh = Grid1D(dx = (2., 4.))
>>> var = CellVariable(mesh = mesh, value = (L, R))
>>> faceValue = var.arithmeticFaceValue[mesh.interiorFaces.value]
>>> answer = (R - L) * (1.0 / 3.0) + L
>>> print numerix.allclose(faceValue, answer, atol = 1e-10, rtol = 1e-10)
True
```

```
>>> mesh = Grid1D(dx = (10., 100.))
>>> var = CellVariable(mesh = mesh, value = (L, R))
>>> faceValue = var.arithmeticFaceValue[mesh.interiorFaces.value]
>>> answer = (R - L) * (5.0 / 55.0) + L
>>> print numerix.allclose(faceValue, answer, atol = 1e-10, rtol = 1e-10)
True
```

cellVolumeAverage

Return the cell-volume-weighted average of the *CellVariable*:

$$< \phi >_\text{vol} = \frac{\sum_\text{cells} \phi_\text{cell} V_\text{cell}}{\sum_\text{cells} V_\text{cell}}$$

```
>>> from fipy.meshes import Grid2D
>>> from fipy.variables.cellVariable import CellVariable
>>> mesh = Grid2D(nx = 3, ny = 1, dx = .5, dy = .1)
>>> var = CellVariable(value = (1, 2, 6), mesh = mesh)
>>> print var.cellVolumeAverage
3.0
```

constrain (*value*, *where=None*)

Constrains the *CellVariable* to *value* at a location specified by *where*.

```
>>> from fipy import *
>>> m = Grid1D(nx=3)
>>> v = CellVariable(mesh=m, value=m.cellCenters[0])
>>> v.constrain(0., where=m.facesLeft)
>>> v.faceGrad.constrain([1.], where=m.facesRight)
>>> print v.faceGrad
[[ 1.  1.  1.  1.]]
>>> print v.faceValue
[ 0.   1.   2.   2.5]
```

Changing the constraint changes the dependencies

```
>>> v.constrain(1., where=m.facesLeft)
>>> print v.faceGrad
[[-1.  1.  1.  1.]]
>>> print v.faceValue
[ 1.   1.   2.   2.5]
```

Constraints can be *Variable*

```
>>> c = Variable(0.)
>>> v.constrain(c, where=m.facesLeft)
>>> print v.faceGrad
[[ 1.  1.  1.  1.]]
>>> print v.faceValue
[ 0.   1.   2.   2.5]
>>> c.value = 1.
>>> print v.faceGrad
[[-1.  1.  1.  1.]]
>>> print v.faceValue
[ 1.   1.   2.   2.5]
```

Constraints can have a *Variable* mask.

```
>>> v = CellVariable(mesh=m)
>>> mask = FaceVariable(mesh=m, value=m.facesLeft)
>>> v.constrain(1., where=mask)
>>> print v.faceValue
[ 1.  0.  0.  0.]
>>> mask[:] = mask | m.facesRight
>>> print v.faceValue
[ 1.  0.  0.  1.]
```

copy()

faceGrad

Return $\nabla \phi$ as a rank-1 *FaceVariable* using differencing for the normal direction(second-order gradient).

faceGradAverage

Return $\nabla \phi$ as a rank-1 *FaceVariable* using averaging for the normal direction(second-order gradient)

faceValue

Returns a *FaceVariable* whose value corresponds to the arithmetic interpolation of the adjacent cells:

$$\phi_f = (\phi_1 - \phi_2)\frac{d_{f2}}{d_{12}} + \phi_2$$

```
>>> from fipy.meshes import Grid1D
>>> from fipy import numerix
>>> mesh = Grid1D(dx = (1., 1.))
>>> L = 1
>>> R = 2
>>> var = CellVariable(mesh = mesh, value = (L, R))
>>> faceValue = var.arithmeticFaceValue[mesh.interiorFaces.value]
>>> answer = (R - L) * (0.5 / 1.) + L
>>> print numerix.allclose(faceValue, answer, atol = 1e-10, rtol = 1e-10)
True
```

```
>>> mesh = Grid1D(dx = (2., 4.))
>>> var = CellVariable(mesh = mesh, value = (L, R))
>>> faceValue = var.arithmeticFaceValue[mesh.interiorFaces.value]
>>> answer = (R - L) * (1.0 / 3.0) + L
>>> print numerix.allclose(faceValue, answer, atol = 1e-10, rtol = 1e-10)
True

>>> mesh = Grid1D(dx = (10., 100.))
>>> var = CellVariable(mesh = mesh, value = (L, R))
>>> faceValue = var.arithmeticFaceValue[mesh.interiorFaces.value]
>>> answer = (R - L) * (5.0 / 55.0) + L
>>> print numerix.allclose(faceValue, answer, atol = 1e-10, rtol = 1e-10)
True
```

gaussGrad
Return $\frac{1}{V_P} \sum_f \vec{n} \phi_f A_f$ as a rank-1 *CellVariable* (first-order gradient).

getArithmeticFaceValue(**args*, ***kwds*)
Deprecated since version 3.0: use the `arithmeticFaceValue` property instead

getCellVolumeAverage(**args*, ***kwds*)
Deprecated since version 3.0: use the `cellVolumeAverage` property instead

getFaceGrad(**args*, ***kwds*)
Deprecated since version 3.0: use the `faceGrad` property instead

getFaceGradAverage(**args*, ***kwds*)
Deprecated since version 3.0: use the `faceGradAverage` property instead

getFaceValue(**args*, ***kwds*)
Deprecated since version 3.0: use the `arithmeticFaceValue` property instead

getGaussGrad(**args*, ***kwds*)
Deprecated since version 3.0: use the `gaussGrad` property instead

getGrad(**args*, ***kwds*)
Deprecated since version 3.0: use the `grad` property instead

getHarmonicFaceValue(**args*, ***kwds*)
Deprecated since version 3.0: use the `harmonicFaceValue` property instead

getLeastSquaresGrad(**args*, ***kwds*)
Deprecated since version 3.0: use the `leastSquaresGrad` property instead

getMinmodFaceValue(**args*, ***kwds*)
Deprecated since version 3.0: use the `minmodFaceValue` property instead

getOld(**args*, ***kwds*)
Deprecated since version 3.0: use the `old` property instead

globalValue
Concatenate and return values from all processors

When running on a single processor, the result is identical to `value`.

grad
Return $\nabla \phi$ as a rank-1 *CellVariable* (first-order gradient).

harmonicFaceValue

Returns a *FaceVariable* whose value corresponds to the harmonic interpolation of the adjacent cells:

$$\phi_f = \frac{\phi_1 \phi_2}{(\phi_2 - \phi_1)\frac{d_{f2}}{d_{12}} + \phi_1}$$

```
>>> from fipy.meshes import Grid1D
>>> from fipy import numerix
>>> mesh = Grid1D(dx = (1., 1.))
>>> L = 1
>>> R = 2
>>> var = CellVariable(mesh = mesh, value = (L, R))
>>> faceValue = var.harmonicFaceValue[mesh.interiorFaces.value]
>>> answer = L * R / ((R - L) * (0.5 / 1.) + L)
>>> print numerix.allclose(faceValue, answer, atol = 1e-10, rtol = 1e-10)
True
```

```
>>> mesh = Grid1D(dx = (2., 4.))
>>> var = CellVariable(mesh = mesh, value = (L, R))
>>> faceValue = var.harmonicFaceValue[mesh.interiorFaces.value]
>>> answer = L * R / ((R - L) * (1.0 / 3.0) + L)
>>> print numerix.allclose(faceValue, answer, atol = 1e-10, rtol = 1e-10)
True
```

```
>>> mesh = Grid1D(dx = (10., 100.))
>>> var = CellVariable(mesh = mesh, value = (L, R))
>>> faceValue = var.harmonicFaceValue[mesh.interiorFaces.value]
>>> answer = L * R / ((R - L) * (5.0 / 55.0) + L)
>>> print numerix.allclose(faceValue, answer, atol = 1e-10, rtol = 1e-10)
True
```

leastSquaresGrad

Return $\nabla\phi$, which is determined by solving for $\nabla\phi$ in the following matrix equation,

$$\nabla\phi \cdot \sum_f d_{AP}^2 \vec{n}_{AP} \otimes \vec{n}_{AP} = \sum_f d_{AP}^2 \left(\vec{n} \cdot \nabla\phi\right)_{AP}$$

The matrix equation is derived by minimizing the following least squares sum,

$$F\left(\phi_x, \phi_y\right) = \sqrt{\sum_f \left(d_{AP} \vec{n}_{AP} \cdot \nabla\phi - d_{AP}\left(\vec{n}_{AP} \cdot \nabla\phi\right)_{AP}\right)^2}$$

Tests

```
>>> from fipy import Grid2D
>>> m = Grid2D(nx=2, ny=2, dx=0.1, dy=2.0)
>>> print numerix.allclose(CellVariable(mesh=m, value=(0,1,3,6)).leastSquaresGrad.globalValu
...                                     [[8.0, 8.0, 24.0, 24.0],
...                                      [1.2, 2.0, 1.2, 2.0]])
True
```

```
>>> from fipy import Grid1D
>>> print numerix.allclose(CellVariable(mesh=Grid1D(dx=(2.0, 1.0, 0.5)),
...                                     value=(0, 1, 2)).leastSquaresGrad.globalValue, [[0.4
True
```

minmodFaceValue

Returns a *FaceVariable* with a value that is the minimum of the absolute values of the adjacent cells. If the

values are of opposite sign then the result is zero:

$$\phi_f = \begin{cases} \phi_1 & \text{when } |\phi_1| \le |\phi_2|, \\ \phi_2 & \text{when } |\phi_2| < |\phi_1|, \\ 0 & \text{when } \phi 1 \phi 2 < 0 \end{cases}$$

```
>>> from fipy import *
>>> print CellVariable(mesh=Grid1D(nx=2), value=(1, 2)).minmodFaceValue
[1 1 2]
>>> print CellVariable(mesh=Grid1D(nx=2), value=(-1, -2)).minmodFaceValue
[-1 -1 -2]
>>> print CellVariable(mesh=Grid1D(nx=2), value=(-1, 2)).minmodFaceValue
[-1  0  2]
```

old

Return the values of the *CellVariable* from the previous solution sweep.

Combinations of *CellVariable's* should also return old values.

```
>>> from fipy.meshes import Grid1D
>>> mesh = Grid1D(nx = 2)
>>> from fipy.variables.cellVariable import CellVariable
>>> var1 = CellVariable(mesh = mesh, value = (2, 3), hasOld = 1)
>>> var2 = CellVariable(mesh = mesh, value = (3, 4))
>>> v = var1 * var2
>>> print v
[ 6 12]
>>> var1.value = ((3,2))
>>> print v
[9 8]
>>> print v.old
[ 6 12]
```

The following small test is to correct for a bug when the operator does not just use variables.

```
>>> v1 = var1 * 3
>>> print v1
[9 6]
>>> print v1.old
[6 9]
```

release (*constraint*)

Remove *constraint* from *self*

```
>>> from fipy import *
>>> m = Grid1D(nx=3)
>>> v = CellVariable(mesh=m, value=m.cellCenters[0])
>>> c = Constraint(0., where=m.facesLeft)
>>> v.constrain(c)
>>> print v.faceValue
[ 0.   1.   2.   2.5]
>>> v.release(constraint=c)
>>> print v.faceValue
[ 0.5  1.   2.   2.5]
```

setValue (*value*, *unit=None*, *where=None*)

updateOld ()

Set the values of the previous solution sweep to the current values.

```
>>> from fipy import *
>>> v = CellVariable(mesh=Grid1D(), hasOld=False)
>>> v.updateOld()
Traceback (most recent call last):
    ...
AssertionError: The updateOld method requires the CellVariable to have an old value. Set has
```

28.8 `cellVolumeAverageVariable` Module

28.9 `constant` Module

28.10 `constraintMask` Module

28.11 `coupledCellVariable` Module

28.12 `exponentialNoiseVariable` Module

class `fipy.variables.exponentialNoiseVariable.`**`ExponentialNoiseVariable`**(*mesh*, *mean=0.0*, *name=''*, *hasOld=0*)

Bases: `fipy.variables.noiseVariable.NoiseVariable`

Represents an exponential distribution of random numbers with the probability distribution

$$\mu^{-1}e^{-\frac{x}{\mu}}$$

with a mean parameter μ.

Seed the random module for the sake of deterministic test results.

```
>>> from fipy import numerix
>>> numerix.random.seed(1)
```

We generate noise on a uniform cartesian mesh

```
>>> from fipy.variables.variable import Variable
>>> mean = Variable()
>>> from fipy.meshes import Grid2D
>>> noise = ExponentialNoiseVariable(mesh = Grid2D(nx = 100, ny = 100), mean = mean)
```

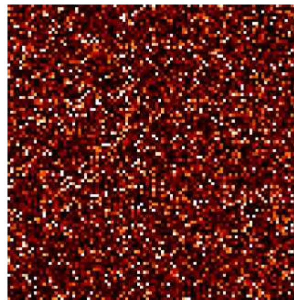
We histogram the root-volume-weighted noise distribution

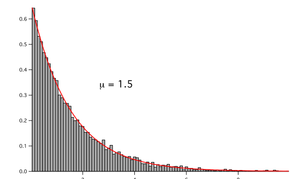
```
>>> from fipy.variables.histogramVariable import HistogramVariable
>>> histogram = HistogramVariable(distribution = noise, dx = 0.1, nx = 100)
```

and compare to a Gaussian distribution

```
>>> from fipy.variables.cellVariable import CellVariable
>>> expdist = CellVariable(mesh = histogram.mesh)
>>> x = histogram.mesh.cellCenters[0]
```

```
>>> if __name__ == '__main__':
...     from fipy import Viewer
...     viewer = Viewer(vars=noise, datamin=0, datamax=5)
...     histoplot = Viewer(vars=(histogram, expdist),
...                        datamin=0, datamax=1.5)

>>> from fipy.tools.numerix import arange, exp

>>> for mu in arange(0.5,3,0.5):
...     mean.value = (mu)
...     expdist.value = ((1/mean)*exp(-x/mean))
...     if __name__ == '__main__':
...         import sys
...         print >>sys.stderr, "mean: %g" % mean
...         viewer.plot()
...         histoplot.plot()

>>> print abs(noise.faceGrad.divergence.cellVolumeAverage) < 5e-15
1
```





Parameters

- *mesh*: The mesh on which to define the noise.
- *mean*: The mean of the distribution μ.

random()

28.13 faceGradContributionsVariable Module

28.14 faceGradVariable Module

28.15 faceVariable Module

class fipy.variables.faceVariable.**FaceVariable**(*mesh*, *name=''*, *value=0.0*, *rank=None*, *elementshape=None*, *unit=None*, *cached=1*)

Bases: fipy.variables.meshVariable._MeshVariable

Parameters

- *mesh*: 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 * (mesh.dim,)*
- *unit*: the physical units of the *Variable*

copy()

divergence

```
>>> from fipy.meshes import Grid2D
>>> from fipy.variables.cellVariable import CellVariable
>>> mesh = Grid2D(nx=3, ny=2)
>>> var = CellVariable(mesh=mesh, value=range(3*2))
>>> print var.faceGrad.divergence
[ 4.  3.  2. -2. -3. -4.]
```

getDivergence(**args*, ***kwds*)
Deprecated since version 3.0: use the `divergence` property instead

globalValue

setValue(*value*, *unit=None*, *where=None*)

28.16 `gammaNoiseVariable` Module

class `fipy.variables.gammaNoiseVariable.`**GammaNoiseVariable**(*mesh*, *shape*, *rate*, *name=''*, *hasOld=0*)
Bases: `fipy.variables.noiseVariable.NoiseVariable`

Represents a gamma distribution of random numbers with the probability distribution

$$x^{\alpha-1}\frac{\beta^\alpha e^{-\beta x}}{\Gamma(\alpha)}$$

with a shape parameter α, a rate parameter β, and $\Gamma(z) = \int_0^\infty t^{z-1}e^{-t}\,dt$.

Seed the random module for the sake of deterministic test results.

```
>>> from fipy import numerix
>>> numerix.random.seed(1)
```

We generate noise on a uniform cartesian mesh

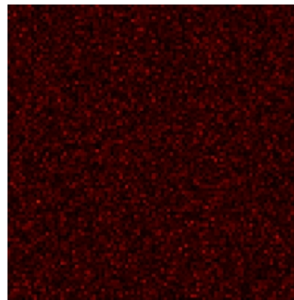
```
>>> from fipy.variables.variable import Variable
>>> alpha = Variable()
>>> beta = Variable()
>>> from fipy.meshes import Grid2D
>>> noise = GammaNoiseVariable(mesh = Grid2D(nx = 100, ny = 100), shape = alpha, rate = beta)
```

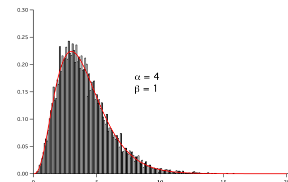
We histogram the root-volume-weighted noise distribution

```
>>> from fipy.variables.histogramVariable import HistogramVariable
>>> histogram = HistogramVariable(distribution = noise, dx = 0.1, nx = 300)
```

and compare to a Gaussian distribution

```
>>> from fipy.variables.cellVariable import CellVariable
>>> x = CellVariable(mesh=histogram.mesh, value=histogram.mesh.cellCenters[0])
>>> from scipy.special import gamma as Gamma
>>> from fipy.tools.numerix import exp
>>> gammadist = (x**(alpha - 1) * (beta**alpha * exp(-beta * x)) / Gamma(alpha))

>>> if __name__ == '__main__':
...     from fipy import Viewer
...     viewer = Viewer(vars=noise, datamin=0, datamax=30)
...     histoplot = Viewer(vars=(histogram, gammadist),
...                        datamin=0, datamax=1)

>>> from fipy.tools.numerix import arange

>>> for shape in arange(1,8,1):
...     alpha.value = shape
...     for rate in arange(0.5,2.5,0.5):
...         beta.value = rate
...         if __name__ == '__main__':
...             import sys
...             print >>sys.stderr, "alpha: %g, beta: %g" % (alpha, beta)
...             viewer.plot()
...             histoplot.plot()

>>> print abs(noise.faceGrad.divergence.cellVolumeAverage) < 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()

28.17 `gaussCellGradVariable` Module

28.18 `gaussianNoiseVariable` Module

class `fipy.variables.gaussianNoiseVariable.`**`GaussianNoiseVariable`**(*mesh*, *name=''*, *mean=0.0*, *variance=1.0*, *hasOld=0*)

Bases: `fipy.variables.noiseVariable.NoiseVariable`

Represents a normal (Gaussian) distribution of random numbers with mean μ and variance $\langle\eta(\vec{r})\eta(\vec{r}\,')\rangle = \sigma^2$, which has the probability distribution

$$\frac{1}{\sigma\sqrt{2\pi}}\exp -\frac{(x-\mu)^2}{2\sigma^2}$$

For example, the variance of thermal noise that is uncorrelated in space and time is often expressed as

$$\langle\eta(\vec{r},t)\eta(\vec{r}\,',t')\rangle = Mk_BT\delta(\vec{r}-\vec{r}\,')\delta(t-t')$$

which can be obtained with:

```
sigmaSqrd = Mobility * kBoltzmann * Temperature / (mesh.cellVolumes * timeStep)
GaussianNoiseVariable(mesh = mesh, variance = sigmaSqrd)
```

Note: If the time step will change as the simulation progresses, either through use of an adaptive iterator or by making manual changes at different stages, remember to declare *timeStep* as a *Variable* and to change its value with its *setValue()* method.

```
>>> import sys
>>> from fipy.tools.numerix import *

>>> mean = 0.
>>> variance = 4.
```

Seed the random module for the sake of deterministic test results.

```
>>> from fipy import numerix
>>> numerix.random.seed(3)
```

We generate noise on a non-uniform cartesian mesh with cell dimensions of x^2 and y^3.

```
>>> from fipy.meshes import Grid2D
>>> mesh = Grid2D(dx = arange(0.1, 5., 0.1)**2, dy = arange(0.1, 3., 0.1)**3)
>>> from fipy.variables.cellVariable import CellVariable
>>> volumes = CellVariable(mesh=mesh,value=mesh.cellVolumes)
>>> noise = GaussianNoiseVariable(mesh = mesh, mean = mean,
...                               variance = variance / volumes)
```

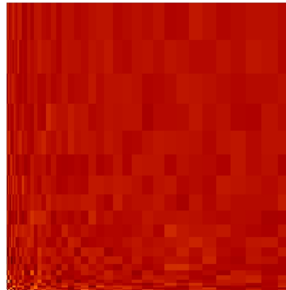
We histogram the root-volume-weighted noise distribution

```
>>> from fipy.variables.histogramVariable import HistogramVariable
>>> histogram = HistogramVariable(distribution = noise * sqrt(volumes),
...                               dx = 0.1, nx = 600, offset = -30)
```

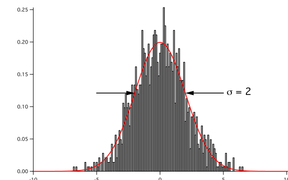
and compare to a Gaussian distribution

```
>>> gauss = CellVariable(mesh = histogram.mesh)
>>> x = histogram.mesh.cellCenters[0]
>>> gauss.value = ((1/(sqrt(variance * 2 * pi))) * exp(-(x - mean)**2 / (2 * variance)))

>>> if __name__ == '__main__':
...     from fipy.viewers import Viewer
...     viewer = Viewer(vars=noise,
...                     datamin=-5, datamax=5)
...     histoplot = Viewer(vars=(histogram, gauss))

>>> for i in range(10):
...     noise.scramble()
...     if __name__ == '__main__':
...         viewer.plot()
...         histoplot.plot()

>>> print abs(noise.faceGrad.divergence.cellVolumeAverage) < 5e-15
1
```

Note that the noise exhibits larger amplitude in the small cells than in the large ones



but that the root-volume-weighted histogram is Gaussian.



Parameters

- *mesh*: The mesh on which to define the noise.
- *mean*: The mean of the noise distrubution, μ.
- *variance*: The variance of the noise distribution, σ^2.

parallelRandom()

28.19 `harmonicCellToFaceVariable` Module

28.20 `histogramVariable` Module

class `fipy.variables.histogramVariable.`**`HistogramVariable`**(*distribution*, *dx=1.0*, *nx=None*, *offset=0.0*)

Bases: `fipy.variables.cellVariable.CellVariable`

Produces a histogram of the values of the supplied distribution.

Parameters

- *distribution*: The collection of values to sample.
- *dx*: the bin size
- *nx*: the number of bins
- *offset*: the position of the first bin

28.21 `leastSquaresCellGradVariable` Module

28.22 `meshVariable` Module

28.23 `minmodCellToFaceVariable` Module

28.24 `modCellGradVariable` Module

28.25 `modCellToFaceVariable` Module

28.26 `modFaceGradVariable` Module

28.27 `modPhysicalField` Module

28.28 `modularVariable` Module

class `fipy.variables.modularVariable.`**`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 exisits on the circle between $-\pi$ and π

The following examples show how *ModularVariable* works. When subtracting the answer wraps back around the circle.

```
>>> from fipy.meshes import Grid1D
>>> mesh = Grid1D(nx = 2)
>>> from fipy.tools import numerix
>>> pi = numerix.pi
```

```
>>> v1 = ModularVariable(mesh = mesh, value = (2*pi/3, -2*pi/3))
>>> v2 = ModularVariable(mesh = mesh, value = -2*pi/3)
>>> print numerix.allclose(v2 - v1, (2*pi/3, 0))
1
```

Obtaining the arithmetic face value.

```
>>> print numerix.allclose(v1.arithmeticFaceValue, (2*pi/3, pi, -2*pi/3))
1
```

Obtaining the gradient.

```
>>> print numerix.allclose(v1.grad, ((pi/3, pi/3),))
1
```

Obtaining the gradient at the faces.

```
>>> print numerix.allclose(v1.faceGrad, ((0, 2*pi/3, 0),))
1
```

Obtaining the gradient at the faces but without modular arithmetic.

```
>>> print numerix.allclose(v1.faceGradNoMod, ((0, -4*pi/3, 0),))
1
```

arithmeticFaceValue
Returns a *FaceVariable* whose value corresponds to the arithmetic interpolation of the adjacent cells:

$$\phi_f = (\phi_1 - \phi_2)\frac{d_{f2}}{d_{12}} + \phi_2$$

Adjusted for a *ModularVariable*

faceGrad
Return $\nabla \phi$ as a rank-1 *FaceVariable* (second-order gradient). Adjusted for a *ModularVariable*

faceGradNoMod
Return $\nabla \phi$ as a rank-1 *FaceVariable* (second-order gradient). Not adjusted for a *ModularVariable*

getFaceGradNoMod(**args*, ***kwds*)
Deprecated since version 3.0: use the `faceGradNoMod` property instead

grad
Return $\nabla \phi$ as a rank-1 *CellVariable* (first-order gradient). Adjusted for a *ModularVariable*

updateOld()
Set the values of the previous solution sweep to the current values. Test case due to bug.

```
>>> from fipy.meshes import Grid1D
>>> mesh = Grid1D(nx = 1)
>>> var = ModularVariable(mesh=mesh, value=1., hasOld=1)
>>> var.updateOld()
>>> var[:] = 2
>>> answer = CellVariable(mesh=mesh, value=1.)
>>> print var.old.allclose(answer)
True
```

28.29 `noiseVariable` Module

class `fipy.variables.noiseVariable.`**NoiseVariable**(*mesh*, *name=''*, *hasOld=0*)
Bases: `fipy.variables.cellVariable.CellVariable`

Attention: This class is abstract. Always create one of its subclasses.

A generic base class for sources of noise distributed over the cells of a mesh.

In the event that the noise should be conserved, use:

```
<Specific>NoiseVariable(...).faceGrad.divergence
```

The *seed()* and *get_seed()* functions of the *fipy.tools.numerix.random* module can be set and query the random number generated used by all *NoiseVariable* objects.

copy()
Copy the value of the *NoiseVariable* to a static *CellVariable*.

parallelRandom()

random()

scramble()
Generate a new random distribution.

28.30 `operatorVariable` Module

28.31 `scharfetterGummelFaceVariable` Module

class `fipy.variables.scharfetterGummelFaceVariable.`**`ScharfetterGummelFaceVariable`**(*var*, *boundaryConditions=()*)

Bases: `fipy.variables.cellToFaceVariable._CellToFaceVariable`

28.32 `test` Module

Test numeric implementation of the mesh

28.33 `unaryOperatorVariable` Module

28.34 `uniformNoiseVariable` Module

class `fipy.variables.uniformNoiseVariable.`**`UniformNoiseVariable`**(*mesh*, *name=''*, *minimum=0.0*, *maximum=1.0*, *hasOld=0*)

Bases: `fipy.variables.noiseVariable.NoiseVariable`

Represents a uniform distribution of random numbers.

We generate noise on a uniform cartesian mesh

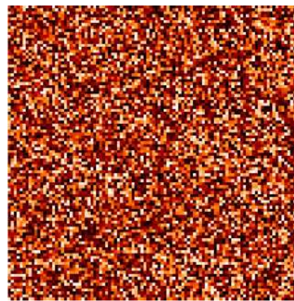
```
>>> from fipy.meshes import Grid2D
>>> noise = UniformNoiseVariable(mesh=Grid2D(nx=100, ny=100))
```

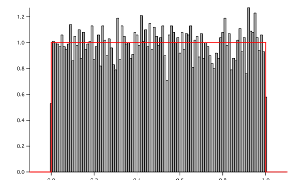
and histogram the noise

```
>>> from fipy.variables.histogramVariable import HistogramVariable
>>> histogram = HistogramVariable(distribution=noise, dx=0.01, nx=120, offset=-.1)
```

```
>>> if __name__ == '__main__':
...     from fipy import Viewer
...     viewer = Viewer(vars=noise,
...                     datamin=0, datamax=1)
...     histoplot = Viewer(vars=histogram)
```

```
>>> 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()

28.35 `variable` Module

class `fipy.variables.variable.`**`Variable`**(*value=0.0*, *unit=None*, *array=None*, *name=''*, *cached=1*)

Bases: `object`

Lazily evaluated quantity with units.

Using a `Variable` in a mathematical expression will create an automatic dependency `Variable`, e.g.,

```
>>> a = Variable(value=3)
>>> b = 4 * a
>>> b
(Variable(value=array(3)) * 4)
>>> b()
12
```

Changes to the value of a `Variable` will automatically trigger changes in any dependent `Variable` objects

```
>>> a.setValue(5)
>>> b
(Variable(value=array(5)) * 4)
>>> print b()
20
```

Create a *Variable*.

```
>>> Variable(value=3)
Variable(value=array(3))
>>> Variable(value=3, unit="m")
Variable(value=PhysicalField(3,'m'))
>>> Variable(value=3, unit="m", array=numerix.zeros((3,2), 'l'))
Variable(value=PhysicalField(array([[3, 3],
       [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=1e-05*, *atol=1e-08*)

```
>>> var = Variable((1, 1))
   >>> print var.allclose((1, 1))
   1
   >>> print var.allclose((1,))
   1
```

The following test is to check that the system does not run out of memory.

```
>>> from fipy.tools import numerix
>>> var = Variable(numerix.ones(10000))
>>> print var.allclose(numerix.zeros(10000, 'l'))
False
```

allequal(*other*)

any(*axis=None*)

```
>>> print Variable(value=0).any()
0
>>> print Variable(value=(0, 0, 1, 1)).any()
1
```

arccos(**args*, ***kwds*)
Deprecated since version 3.0: use `numerix.arccos()` instead

arccosh(**args*, ***kwds*)
Deprecated since version 3.0: use `numerix.arccosh()` instead

arcsin(**args*, ***kwds*)
Deprecated since version 3.0: use `numerix.arcsin()` instead

arcsinh(**args*, ***kwds*)
Deprecated since version 3.0: use `numerix.arcsinh()` instead

arctan(**args*, ***kwds*)
Deprecated since version 3.0: use `numerix.arctan()` instead

arctan2(**args*, ***kwds*)
Deprecated since version 3.0: use `numerix.arctan2()` instead

arctanh(**args*, ***kwds*)
Deprecated since version 3.0: use `numerix.arctanh()` instead

cacheMe(*recursive=False*)

ceil(**args*, ***kwds*)
Deprecated since version 3.0: use `numerix.ceil()` instead

conjugate(**args*, ***kwds*)
Deprecated since version 3.0: use `numerix.conjugate()` instead

constrain(*value*, *where=None*)
Constrain the *Variable* to have a *value* at an index or mask location specified by *where*.

```
>>> v = Variable((0,1,2,3))
>>> v.constrain(2, numerix.array((True, False, False, False)))
>>> print v
[2 1 2 3]
>>> v[:] = 10
>>> print v
[ 2 10 10 10]
>>> v.constrain(5, numerix.array((False, False, True, False)))
>>> print v
[ 2 10  5 10]
>>> v[:] = 6
>>> print v
[2 6 5 6]
>>> v.constrain(8)
>>> print v
[8 8 8 8]
>>> v[:] = 10
>>> print v
[8 8 8 8]
>>> del v.constraints[2]
```

```
>>> print v
[ 2 10  5 10]

>>> from fipy.variables.cellVariable import CellVariable
>>> from fipy.meshes import Grid2D
>>> m = Grid2D(nx=2, ny=2)
>>> x, y = m.cellCenters
>>> v = CellVariable(mesh=m, rank=1, value=(x, y))
>>> v.constrain(((0.,), (-1.,)), where=m.facesLeft)
>>> print v.faceValue
[[ 0.5  1.5  0.5  1.5  0.5  1.5  0.   1.   1.5  0.   1.   1.5]
 [ 0.5  0.5  1.   1.   1.5  1.5 -1.   0.5  0.5 -1.   1.5  1.5]]
```

Parameters

- *value*: the value of the constraint
- *where*: the constraint mask or index specifying the location of the constraint

constraints

copy()

Make an duplicate of the *Variable*

```
>>> a = Variable(value=3)
>>> b = a.copy()
>>> b
Variable(value=array(3))
```

The duplicate will not reflect changes made to the original

```
>>> a.setValue(5)
>>> b
Variable(value=array(3))
```

Check that this works for arrays.

```
>>> a = Variable(value=numerix.array((0,1,2)))
>>> b = a.copy()
>>> b
Variable(value=array([0, 1, 2]))
>>> a[1] = 3
>>> b
Variable(value=array([0, 1, 2]))
```

cos(**args*, ***kwds*)

Deprecated since version 3.0: use `numerix.cos()` instead

cosh(**args*, ***kwds*)

Deprecated since version 3.0: use `numerix.cosh()` instead

dontCacheMe(*recursive=False*)

dot(*other*, *opShape=None*, *operatorClass=None*, *axis=0*)

exp(**args*, ***kwds*)

Deprecated since version 3.0: use `numerix.exp()` instead

floor(**args*, ***kwds*)

Deprecated since version 3.0: use `numerix.floor()` instead

getMag (**args*, ***kwds*)
Deprecated since version 3.0: use the `mag` property instead

getName (**args*, ***kwds*)
Deprecated since version 3.0: use the `name` property instead

getNumericValue (**args*, ***kwds*)
Deprecated since version 3.0: use the `numericValue` property instead

getShape (**args*, ***kwds*)
Deprecated since version 3.0: use the `shape` property instead

getSubscribedVariables (**args*, ***kwds*)
Deprecated since version 3.0: use the `subscribedVariables` property instead

getUnit (**args*, ***kwds*)
Deprecated since version 3.0: use the `unit` property instead

getValue (**args*, ***kwds*)
Deprecated since version 3.0: use the `value` property instead

getsctype (*default=None*)
Returns the Numpy sctype of the underlying array.

```
>>> Variable(1).getsctype() == numerix.NUMERIX.obj2sctype(numerix.array(1))
True
>>> Variable(1.).getsctype() == numerix.NUMERIX.obj2sctype(numerix.array(1.))
True
>>> Variable((1,1.)).getsctype() == numerix.NUMERIX.obj2sctype(numerix.array((1., 1.)))
True
```

inBaseUnits ()
Return the value of the *Variable* with all units reduced to their base SI elements.

```
>>> e = Variable(value="2.7 Hartree*Nav")
>>> print e.inBaseUnits().allclose("7088849.01085 kg*m**2/s**2/mol")
1
```

inUnitsOf (**units*)
Returns one or more *Variable* objects that express the same physical quantity in different units. The units are specified by strings containing their names. The units must be compatible with the unit of the object. If one unit is specified, the return value is a single *Variable*.

```
>>> freeze = Variable('0 degC')
>>> print freeze.inUnitsOf('degF').allclose("32.0 degF")
1
```

If several units are specified, the return value is a tuple of *Variable* instances with with one element per unit such that the sum of all quantities in the tuple equals the the original quantity and all the values except for the last one are integers. This is used to convert to irregular unit systems like hour/minute/second. The original object will not be changed.

```
>>> t = Variable(value=314159., unit='s')
>>> print numerix.allclose([e.allclose(v) for (e, v) in zip(t.inUnitsOf('d','h','min','s'),
...                                                         ['3.0 d', '15.0 h', '15.0 min',
...                                 True)
1
```

itemset (*value*)

itemsize

log (**args*, ***kwds*)
Deprecated since version 3.0: use `numerix.log()` instead

log10 (**args*, ***kwds*)
Deprecated since version 3.0: use `numerix.log10()` instead

mag

max (*axis=None*)

min (*axis=None*)

name

numericValue

put (*indices*, *value*)

ravel ()

release (*constraint*)
Remove *constraint* from *self*

```
>>> v = Variable((0,1,2,3))
>>> v.constrain(2, numerix.array((True, False, False, False)))
>>> v[:] = 10
>>> from fipy.boundaryConditions.constraint import Constraint
>>> c1 = Constraint(5, numerix.array((False, False, True, False)))
>>> v.constrain(c1)
>>> v[:] = 6
>>> v.constrain(8)
>>> v[:] = 10
>>> del v.constraints[2]
>>> v.release(constraint=c1)
>>> print v
[ 2 10 10 10]
```

reshape (**args*, ***kwds*)
Deprecated since version 3.0: use `numerix.reshape()` instead

setName (**args*, ***kwds*)
Deprecated since version 3.0: use the `name` property instead

setUnit (**args*, ***kwds*)
Deprecated since version 3.0: use the `unit` property instead

setValue (*value*, *unit=None*, *where=None*)
Set the value of the Variable. Can take a masked array.

```
>>> a = Variable((1,2,3))
>>> a.setValue(5, where=(1, 0, 1))
>>> print a
[5 2 5]

>>> b = Variable((4,5,6))
>>> a.setValue(b, where=(1, 0, 1))
>>> print a
[4 2 6]
>>> print b
[4 5 6]
>>> a.value = 3
>>> print a
[3 3 3]
```

```
>>> b = numerix.array((3,4,5))
>>> a.value = b
>>> a[:] = 1
>>> print b
[3 4 5]

>>> a.setValue((4,5,6), where=(1, 0))
Traceback (most recent call last):
    ....
ValueError: shape mismatch: objects cannot be broadcast to a single shape
```

shape

Tuple of array dimensions.

```
>>> 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,)
```

sign(**args, **kwds*)

Deprecated since version 3.0: use `numerix.sign()` instead

sin(**args, **kwds*)

Deprecated since version 3.0: use `numerix.sin()` instead

sinh(**args, **kwds*)

Deprecated since version 3.0: use `numerix.sinh()` instead

sqrt(**args, **kwds*)

Deprecated since version 3.0: use `numerix.sqrt()` instead

```
>>> from fipy.meshes 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.]
```

subscribedVariables

sum(*axis=None*)

take(*ids, axis=0*)

tan(**args, **kwds*)

Deprecated since version 3.0: use `numerix.tan()` instead

tanh(**args, **kwds*)

Deprecated since version 3.0: use `numerix.tanh()` instead

tostring(*max_line_width=75, precision=8, suppress_small=False, separator=' '*)

unit

Return the unit object of *self*.

```
>>> Variable(value="1 m").unit
<PhysicalUnit m>
```

value

"Evaluate" the *Variable* and return its value (longhand)

```
>>> a = Variable(value=3)
>>> print a.value
3
>>> b = a + 4
>>> b
(Variable(value=array(3)) + 4)
>>> b.value
7
```

Chapter 29

viewers Package

29.1 `viewers` Package

`fipy.viewers.`**`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.

`fipy.viewers.`**`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.

`fipy.viewers.`**`MatplotlibViewer`**(*vars*, *title=None*, *limits={}*, *cmap=None*, *colorbar='vertical'*, *axes=None*, ***kwlimits*)
Generic function for creating a *MatplotlibViewer*.

The *MatplotlibViewer* factory will search the module tree and return an instance of the first *MatplotlibViewer* it finds of the correct dimension and rank.

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.

cmap the colormap. Defaults to *matplotlib.cm.jet*

colorbar plot a colorbar in specified orientation if not *None*

axes if not *None*, *vars* will be plotted into this Matplotlib *Axes* object

It is possible to view different Variables against different Matplotlib *Axes*

```
>>> from matplotlib import pylab
>>> from fipy import *

>>> pylab.ion()
>>> fig = pylab.figure()

>>> ax1 = pylab.subplot((221))
>>> ax2 = pylab.subplot((223))
>>> ax3 = pylab.subplot((224))

>>> k = Variable(name="k", value=0.)

>>> mesh1 = Grid1D(nx=100)
>>> x, = mesh1.cellCenters
>>> xVar = CellVariable(mesh=mesh1, name="x", value=x)
>>> viewer1 = MatplotlibViewer(vars=(numerix.sin(0.1 * k * xVar), numerix.cos(0.1 * k * xVar / n
...                                limits={'xmin': 10, 'xmax': 90},
...                                datamin=-0.9, datamax=2.0,
...                                title="Grid1D test",
...                                axes=ax1,
...                                legend=None)

>>> mesh2 = Grid2D(nx=50, ny=100, dx=0.1, dy=0.01)
>>> x, y = mesh2.cellCenters
>>> xyVar = CellVariable(mesh=mesh2, name="x y", value=x * y)
>>> viewer2 = MatplotlibViewer(vars=numerix.sin(k * xyVar),
...                                limits={'ymin': 0.1, 'ymax': 0.9},
...                                datamin=-0.9, datamax=2.0,
...                                title="Grid2D test",
...                                axes=ax2,
...                                colorbar=None)

>>> mesh3 = (Grid2D(nx=5, ny=10, dx=0.1, dy=0.1)
...          + (Tri2D(nx=5, ny=5, dx=0.1, dy=0.1)
...             + ((0.5,), (0.2,))))
>>> x, y = mesh3.cellCenters
>>> xyVar = CellVariable(mesh=mesh3, name="x y", value=x * y)
>>> viewer3 = MatplotlibViewer(vars=numerix.sin(k * xyVar),
...                                limits={'ymin': 0.1, 'ymax': 0.9},
...                                datamin=-0.9, datamax=2.0,
...                                title="Irregular 2D test",
...                                axes=ax3,
...                                cmap = pylab.cm.OrRd)

>>> viewer = MultiViewer(viewers=(viewer1, viewer2, viewer3))
>>> for kval in range(10):
```

```
...     k.setValue(kval)
...     viewer.plot()

>>> viewer._promptForOpinion()
```

class `fipy.viewers.`**`Matplotlib1DViewer`**(*vars*, *title=None*, *xlog=False*, *ylog=False*, *limits={}*, *legend='upper left'*, *axes=None*, ***kwlimits*)

Bases: `fipy.viewers.matplotlibViewer.matplotlibViewer.AbstractMatplotlibViewer`

Displays a y vs. x plot of one or more 1D *CellVariable* objects using Matplotlib.

```
>>> from fipy import *
>>> mesh = Grid1D(nx=100)
>>> x, = mesh.cellCenters
>>> xVar = CellVariable(mesh=mesh, name="x", value=x)
>>> k = Variable(name="k", value=0.)
>>> viewer = Matplotlib1DViewer(vars=(numerix.sin(k * xVar), numerix.cos(k * xVar / numerix.pi))
...                 limits={'xmin': 10, 'xmax': 90},
...                 datamin=-0.9, datamax=2.0,
...                 title="Matplotlib1DViewer test")
>>> for kval in numerix.arange(0,0.3,0.03):
...     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*).

legend place a legend at the specified position, if not *None*

axes if not *None*, *vars* will be plotted into this Matplotlib *Axes* object

`log`

logarithmic data scaling

class `fipy.viewers.`**`Matplotlib2DGridViewer`**(*vars*, *title=None*, *limits={}*, *cmap=None*, *colorbar='vertical'*, *axes=None*, *figaspect='auto'*, ***kwlimits*)

Bases: `fipy.viewers.matplotlibViewer.matplotlib2DViewer.AbstractMatplotlib2DViewer`

Displays an image plot of a 2D *CellVariable* object using Matplotlib.

```
>>> from fipy import *
>>> mesh = Grid2D(nx=50, ny=100, dx=0.1, dy=0.01)
>>> x, y = mesh.cellCenters
>>> xyVar = CellVariable(mesh=mesh, name="x y", value=x * y)
>>> k = Variable(name="k", value=0.)
>>> viewer = Matplotlib2DGridViewer(vars=numerix.sin(k * xyVar),
...                 limits={'ymin': 0.1, 'ymax': 0.9},
...                 datamin=-0.9, datamax=2.0,
```

```
...                  title="Matplotlib2DGridViewer test")
>>> 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 *matplotlib.cm.jet*

xmin, xmax, ymin, ymax, datamin, datamax displayed range of data. Any limit set to a (default) value of *None* will autoscale.

colorbar plot a colorbar in specified orientation if not *None*

axes if not *None*, *vars* will be plotted into this Matplotlib *Axes* object

figaspect desired aspect ratio of figure. If arg is a number, use that aspect ratio. If arg is 'auto', the aspect ratio will be determined from the Variable's mesh.

class fipy.viewers.**Matplotlib2DGridContourViewer**(*vars*, *title=None*, *limits={}*, *cmap=None*, *colorbar='vertical'*, *axes=None*, *figaspect='auto'*, ***kwlimits*)

Bases: fipy.viewers.matplotlibViewer.matplotlib2DViewer.AbstractMatplotlib2DViewer

Displays a contour plot of a 2D *CellVariable* object.

The *Matplotlib2DGridContourViewer* plots a 2D *CellVariable* using Matplotlib.

```
>>> from fipy import *
>>> mesh = Grid2D(nx=50, ny=100, dx=0.1, dy=0.01)
>>> x, y = mesh.cellCenters
>>> xyVar = CellVariable(mesh=mesh, name="x y", value=x * y)
>>> k = Variable(name="k", value=0.)
>>> viewer = Matplotlib2DGridContourViewer(vars=numerix.sin(k * xyVar),
...                  limits={'ymin': 0.1, 'ymax': 0.9},
...                  datamin=-0.9, datamax=2.0,
...                  title="Matplotlib2DGridContourViewer test")
>>> 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.

cmap the colormap. Defaults to *matplotlib.cm.jet*

colorbar plot a colorbar in specified orientation if not *None*

axes if not *None*, *vars* will be plotted into this Matplotlib *Axes* object

figaspect desired aspect ratio of figure. If arg is a number, use that aspect ratio. If arg is 'auto', the aspect ratio will be determined from the Variable's mesh.

class `fipy.viewers.`**`Matplotlib2DViewer`**(*vars*, *title=None*, *limits={}*, *cmap=None*, *colorbar='vertical'*, *axes=None*, *figaspect='auto'*, ***kwlimits*)

Bases: `fipy.viewers.matplotlibViewer.matplotlib2DViewer.AbstractMatplotlib2DViewer`

Displays a contour plot of a 2D *CellVariable* object.

The *Matplotlib2DViewer* plots a 2D *CellVariable* using Matplotlib.

```
>>> from fipy import *
>>> mesh = (Grid2D(nx=5, ny=10, dx=0.1, dy=0.1)
...         + (Tri2D(nx=5, ny=5, dx=0.1, dy=0.1)
...          + ((0.5,), (0.2,))))
>>> x, y = mesh.cellCenters
>>> xyVar = CellVariable(mesh=mesh, name="x y", value=x * y)
>>> k = Variable(name="k", value=0.)
>>> viewer = Matplotlib2DViewer(vars=numerix.sin(k * xyVar),
...                     limits={'ymin': 0.1, 'ymax': 0.9},
...                     datamin=-0.9, datamax=2.0,
...                     title="Matplotlib2DViewer test")
>>> 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 *matplotlib.cm.jet*

xmin, xmax, ymin, ymax, datamin, datamax displayed range of data. Any limit set to a (default) value of *None* will autoscale.

colorbar plot a colorbar in specified orientation if not *None*

axes if not *None*, *vars* will be plotted into this Matplotlib *Axes* object

figaspect desired aspect ratio of figure. If arg is a number, use that aspect ratio. If arg is 'auto', the aspect ratio will be determined from the Variable's mesh.

class `fipy.viewers.`**`MatplotlibVectorViewer`**(*vars*, *title=None*, *scale=None*, *sparsity=None*, *log=False*, *limits={}*, *axes=None*, *figaspect='auto'*, ***kwlimits*)

Bases: `fipy.viewers.matplotlibViewer.matplotlib2DViewer.AbstractMatplotlib2DViewer`

Displays a vector plot of a 2D rank-1 *CellVariable* or *FaceVariable* object using Matplotlib

```
>>> from fipy import *
>>> mesh = Grid2D(nx=50, ny=100, dx=0.1, dy=0.01)
>>> x, y = mesh.cellCenters
>>> xyVar = CellVariable(mesh=mesh, name="x y", value=x * y)
```

```
>>> k = Variable(name="k", value=1.)
>>> viewer = MatplotlibVectorViewer(vars=numerix.sin(k * xyVar).grad,
...                 title="MatplotlibVectorViewer test")
>>> for kval in numerix.arange(1, 10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()

>>> viewer = MatplotlibVectorViewer(vars=numerix.sin(k * xyVar).faceGrad,
...                 title="MatplotlibVectorViewer test")
>>> for kval in numerix.arange(1, 10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()

>>> for sparsity in numerix.arange(5000, 0, -500):
...     viewer.quiver(sparsity=sparsity)
...     viewer.plot()
>>> viewer._promptForOpinion()

>>> from fipy import *
>>> mesh = (Grid2D(nx=5, ny=10, dx=0.1, dy=0.1)
...         + (Tri2D(nx=5, ny=5, dx=0.1, dy=0.1)
...          + ((0.5,), (0.2,))))
>>> x, y = mesh.cellCenters
>>> xyVar = CellVariable(mesh=mesh, name="x y", value=x * y)
>>> k = Variable(name="k", value=1.)
>>> viewer = MatplotlibVectorViewer(vars=numerix.sin(k * xyVar).grad,
...                 title="MatplotlibVectorViewer test")
>>> for kval in numerix.arange(1, 10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()

>>> viewer = MatplotlibVectorViewer(vars=numerix.sin(k * xyVar).faceGrad,
...                 title="MatplotlibVectorViewer test")
>>> for kval in numerix.arange(1, 10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()
```

Creates a *Matplotlib2DViewer*.

Parameters

vars 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)

log if *True*, arrow length goes at the base-10 logarithm of the magnitude

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.

axes if not *None*, *vars* will be plotted into this Matplotlib *Axes* object

figaspect desired aspect ratio of figure. If arg is a number, use that aspect ratio. If arg is 'auto', the aspect ratio will be determined from the Variable's mesh.

quiver (*sparsity=None*, *scale=None*)

class fipy.viewers.**MayaviClient** (*vars*, *title=None*, *daemon_file=None*, *fps=1.0*, ***kwlimits*)
Bases: fipy.viewers.viewer.AbstractViewer

The *MayaviClient* uses the Mayavi python plotting package.

```
>>> from fipy import *
>>> mesh = Grid1D(nx=100)
>>> x, = mesh.cellCenters
>>> xVar = CellVariable(mesh=mesh, name="x", value=x)
>>> k = Variable(name="k", value=0.)
>>> viewer = MayaviClient(vars=(numerix.sin(k * xVar), numerix.cos(k * xVar / numerix.pi)),
...                       limits={'xmin': 10, 'xmax': 90},
...                       datamin=-0.9, datamax=2.0,
...                       title="MayaviClient test")
>>> for kval in numerix.arange(0,0.3,0.03):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()
```

```
>>> from fipy import *
>>> mesh = Grid2D(nx=50, ny=100, dx=0.1, dy=0.01)
>>> x, y = mesh.cellCenters
>>> xyVar = CellVariable(mesh=mesh, name="x y", value=x * y)
>>> k = Variable(name="k", value=0.)
>>> viewer = MayaviClient(vars=numerix.sin(k * xyVar),
...                       limits={'ymin': 0.1, 'ymax': 0.9},
...                       datamin=-0.9, datamax=2.0,
...                       title="MayaviClient test")
>>> for kval in range(10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()
```

```
>>> from fipy import *
>>> mesh = (Grid2D(nx=5, ny=10, dx=0.1, dy=0.1)
...         + (Tri2D(nx=5, ny=5, dx=0.1, dy=0.1)
...          + ((0.5,), (0.2,))))
>>> x, y = mesh.cellCenters
>>> xyVar = CellVariable(mesh=mesh, name="x y", value=x * y)
>>> k = Variable(name="k", value=0.)
>>> viewer = MayaviClient(vars=numerix.sin(k * xyVar),
...                       limits={'ymin': 0.1, 'ymax': 0.9},
...                       datamin=-0.9, datamax=2.0,
...                       title="MayaviClient test")
>>> for kval in range(10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()
```

```
>>> from fipy import *
>>> mesh = Grid3D(nx=50, ny=100, nz=10, dx=0.1, dy=0.01, dz=0.1)
>>> x, y, z = mesh.cellCenters
>>> xyzVar = CellVariable(mesh=mesh, name=r"x y z", value=x * y * z)
```

```
>>> k = Variable(name="k", value=0.)
>>> viewer = MayaviClient(vars=numerix.sin(k * xyzVar),
...                       limits={'ymin': 0.1, 'ymax': 0.9},
...                       datamin=-0.9, datamax=2.0,
...                       title="MayaviClient test")
>>> for kval in range(10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()
```

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*)

class fipy.viewers.**MultiViewer**(*viewers*)
Bases: fipy.viewers.viewer.AbstractViewer

Treat a collection of different viewers (such for different 2D plots or 1D plots with different axes) as a single viewer that will *plot()* all subviewers simultaneously.

Parameters

viewers [list] the viewers to bind together

getViewers(**args*, ***kwds*)
Deprecated since version 3.0: use the `viewers` property instead

plot()

setLimits(*limits={}*, ***kwlimits*)

class fipy.viewers.**TSVViewer**(*vars*, *title=None*, *limits={}*, ***kwlimits*)
Bases: fipy.viewers.viewer.AbstractViewer

"Views" one or more variables in tab-separated-value format.

Output is a list of coordinates and variable values at each cell center.

File contents will be, e.g.:

```
title
x       y       ...     var0    var2    ...
0.0     0.0     ...     3.14    1.41    ...
1.0     0.0     ...     2.72    0.866   ...
:
:
```

Creates a *TSVViewer*.

Any cell centers that lie outside the limits provided will not be included. Any values that lie outside the *datamin* or *datamax* will be replaced with *nan*.

All variables must have the same mesh.

It tries to do something reasonable with rank-1 *CellVariable* and *FaceVariable* objects.

Parameters

vars 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 import Grid1D
>>> m = Grid1D(nx = 3, dx = 0.4)
>>> from fipy.variables.cellVariable import CellVariable
>>> v = CellVariable(mesh = m, name = "var", value = (0, 2, 5))
>>> TSVViewer(vars = (v, v.grad)).plot()
x       var     var_gauss_grad_x
0.2     0       2.5
0.6     2       6.25
1       5       3.75
```

```
>>> from fipy.meshes import Grid2D
>>> m = Grid2D(nx = 2, dx = .1, ny = 2, dy = 0.3)
>>> v = CellVariable(mesh = m, name = "var", value = (0, 2, -2, 5))
>>> TSVViewer(vars = (v, v.grad)).plot()
x       y       var     var_gauss_grad_x        var_gauss_grad_y
0.05    0.15    0       10      -3.33333333333333
0.15    0.15    2       10      5
0.05    0.45    -2      35      -3.33333333333333
0.15    0.45    5       35      5
```

Parameters

filename If not *None*, the name of a file to save the image into.

fipy.viewers.**VTKViewer** (*vars*, *title=None*, *limits={}*, ***kwlimits*)

Generic function for creating a *VTKViewer*.

The *VTKViewer* factory will search the module tree and return an instance of the first *VTKViewer* it finds of the correct dimension and rank.

Parameters

vars 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 `fipy.viewers.`**`VTKCellViewer`**(*vars*, *title=None*, *limits={}*, ***kwlimits*)

Bases: `fipy.viewers.vtkViewer.vtkViewer.VTKViewer`

Renders *CellVariable* data in VTK format

Creates a VTKViewer

Parameters

vars 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 `fipy.viewers.`**`VTKFaceViewer`**(*vars*, *title=None*, *limits={}*, ***kwlimits*)

Bases: `fipy.viewers.vtkViewer.vtkViewer.VTKViewer`

Renders *_MeshVariable* data in VTK format

Creates a VTKViewer

Parameters

vars 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.

exception `fipy.viewers.`**`MeshDimensionError`**

Bases: `exceptions.IndexError`

class `fipy.viewers.`**`DummyViewer`**(*vars*, *title=None*, ***kwlimits*)

Bases: `fipy.viewers.viewer.AbstractViewer`

Create a *AbstractViewer* object.

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.

`plot`(*filename=None*)

`fipy.viewers.`**`Viewer`**(*vars*, *title=None*, *limits={}*, *FIPY_VIEWER=None*, ***kwlimits*)

Generic function for creating a *Viewer*.

The *Viewer* factory will search the module tree and return an instance of the first *Viewer* it finds that supports the dimensions of *vars*. Setting the '*FIPY_VIEWER*' environment variable to either '*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.

29.2 `multiViewer` Module

class `fipy.viewers.multiViewer.`**`MultiViewer`**(*viewers*)

Bases: `fipy.viewers.viewer.AbstractViewer`

Treat a collection of different viewers (such for different 2D plots or 1D plots with different axes) as a single viewer that will *plot()* all subviewers simultaneously.

Parameters

viewers [list] the viewers to bind together

`getViewers`(**args*, ***kwds*)

Deprecated since version 3.0: use the `viewers` property instead

`plot`()

`setLimits`(*limits={}*, ***kwlimits*)

29.3 `test` Module

Test implementation of the viewers

29.4 `testinteractive` Module

Interactively test the viewers

29.5 `tsvViewer` Module

class `fipy.viewers.tsvViewer.`**`TSVViewer`**(*vars*, *title=None*, *limits={}*, ***kwlimits*)
Bases: `fipy.viewers.viewer.AbstractViewer`

"Views" one or more variables in tab-separated-value format.

Output is a list of coordinates and variable values at each cell center.

File contents will be, e.g.:

```
title
x       y       ...     var0    var2    ...
0.0     0.0     ...     3.14    1.41    ...
1.0     0.0     ...     2.72    0.866   ...
:
:
```

Creates a *TSVViewer*.

Any cell centers that lie outside the limits provided will not be included. Any values that lie outside the *datamin* or *datamax* will be replaced with *nan*.

All variables must have the same mesh.

It tries to do something reasonable with rank-1 *CellVariable* and *FaceVariable* objects.

Parameters

vars 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 import Grid1D
>>> m = Grid1D(nx = 3, dx = 0.4)
>>> from fipy.variables.cellVariable import CellVariable
>>> v = CellVariable(mesh = m, name = "var", value = (0, 2, 5))
>>> TSVViewer(vars = (v, v.grad)).plot()
x       var     var_gauss_grad_x
0.2     0       2.5
0.6     2       6.25
1       5       3.75
```

```
>>> from fipy.meshes import Grid2D
>>> m = Grid2D(nx = 2, dx = .1, ny = 2, dy = 0.3)
>>> v = CellVariable(mesh = m, name = "var", value = (0, 2, -2, 5))
>>> TSVViewer(vars = (v, v.grad)).plot()
x       y       var     var_gauss_grad_x        var_gauss_grad_y
0.05    0.15    0       10      -3.33333333333333
0.15    0.15    2       10      5
0.05    0.45    -2      35      -3.33333333333333
0.15    0.45    5       35      5
```

Parameters

filename If not *None*, the name of a file to save the image into.

29.6 `viewer` Module

class `fipy.viewers.viewer.`**`AbstractViewer`**(*vars*, *title=None*, ***kwlimits*)

Bases: `object`

> **Attention:** This class is abstract. Always create one of its subclasses.

Create a *AbstractViewer* object.

Parameters

vars 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.

`getVars`(**args*, ***kwds*)

Get the Variables Deprecated since version 3.0: use the `vars` property instead

Deprecated since version Use: `fipy.viewers.viewer.AbstractViewer.vars` instead

`plot`(*filename=None*)

Update the display of the viewed variables.

Parameters

filename If not *None*, the name of a file to save the image into.

`plotMesh`(*filename=None*)

Display a representation of the mesh

Parameters

filename If not *None*, the name of a file to save the image into.

`setLimits`(*limits={}*, ***kwlimits*)

Update the limits.

Parameters

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.

29.7 Subpackages

29.7.1 gistViewer Package

`gistViewer` Package

`fipy.viewers.gistViewer.`**`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.

`colorbar` Module

`gist1DViewer` Module

`gist2DViewer` Module

`gistVectorViewer` Module

`gistViewer` Module

`test` Module

Test numeric implementation of the mesh

29.7.2 gnuplotViewer Package

`gnuplotViewer` Package

`fipy.viewers.gnuplotViewer.`**`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.

`gnuplot1DViewer` Module

`gnuplot2DViewer` Module

`gnuplotViewer` Module

`test` Module

Test numeric implementation of the mesh

29.7.3 matplotlibViewer Package

`matplotlibViewer` Package

`fipy.viewers.matplotlibViewer.`**`MatplotlibViewer`**(*vars*, *title=None*, *limits={}*, *cmap=None*, *colorbar='vertical'*, *axes=None*, ***kwlimits*)

Generic function for creating a *MatplotlibViewer*.

The *MatplotlibViewer* factory will search the module tree and return an instance of the first *MatplotlibViewer* it finds of the correct dimension and rank.

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.

cmap the colormap. Defaults to *matplotlib.cm.jet*

colorbar plot a colorbar in specified orientation if not *None*

axes if not *None*, *vars* will be plotted into this Matplotlib *Axes* object

It is possible to view different `Variable`s against different Matplotlib *Axes*

```
>>> from matplotlib import pylab
>>> from fipy import *

>>> pylab.ion()
>>> fig = pylab.figure()

>>> ax1 = pylab.subplot((221))
>>> ax2 = pylab.subplot((223))
>>> ax3 = pylab.subplot((224))

>>> k = Variable(name="k", value=0.)
```

```
>>> mesh1 = Grid1D(nx=100)
>>> x, = mesh1.cellCenters
>>> xVar = CellVariable(mesh=mesh1, name="x", value=x)
>>> viewer1 = MatplotlibViewer(vars=(numerix.sin(0.1 * k * xVar), numerix.cos(0.1 * k * xVar / n
...                            limits={'xmin': 10, 'xmax': 90},
...                            datamin=-0.9, datamax=2.0,
...                            title="Grid1D test",
...                            axes=ax1,
...                            legend=None)

>>> mesh2 = Grid2D(nx=50, ny=100, dx=0.1, dy=0.01)
>>> x, y = mesh2.cellCenters
>>> xyVar = CellVariable(mesh=mesh2, name="x y", value=x * y)
>>> viewer2 = MatplotlibViewer(vars=numerix.sin(k * xyVar),
...                            limits={'ymin': 0.1, 'ymax': 0.9},
...                            datamin=-0.9, datamax=2.0,
...                            title="Grid2D test",
...                            axes=ax2,
...                            colorbar=None)

>>> mesh3 = (Grid2D(nx=5, ny=10, dx=0.1, dy=0.1)
...          + (Tri2D(nx=5, ny=5, dx=0.1, dy=0.1)
...             + ((0.5,), (0.2,))))
>>> x, y = mesh3.cellCenters
>>> xyVar = CellVariable(mesh=mesh3, name="x y", value=x * y)
>>> viewer3 = MatplotlibViewer(vars=numerix.sin(k * xyVar),
...                            limits={'ymin': 0.1, 'ymax': 0.9},
...                            datamin=-0.9, datamax=2.0,
...                            title="Irregular 2D test",
...                            axes=ax3,
...                            cmap = pylab.cm.OrRd)

>>> viewer = MultiViewer(viewers=(viewer1, viewer2, viewer3))
>>> for kval in range(10):
...     k.setValue(kval)
...     viewer.plot()

>>> viewer._promptForOpinion()
```

class fipy.viewers.matplotlibViewer.**Matplotlib1DViewer**(*vars*, *title=None*, *xlog=False*, *ylog=False*, *limits={}*, *legend='upper left'*, *axes=None*, ***kwlimits*)

Bases: fipy.viewers.matplotlibViewer.matplotlibViewer.AbstractMatplotlibViewer

Displays a y vs. x plot of one or more 1D *CellVariable* objects using Matplotlib.

```
>>> from fipy import *
>>> mesh = Grid1D(nx=100)
>>> x, = mesh.cellCenters
>>> xVar = CellVariable(mesh=mesh, name="x", value=x)
>>> k = Variable(name="k", value=0.)
>>> viewer = Matplotlib1DViewer(vars=(numerix.sin(k * xVar), numerix.cos(k * xVar / numerix.pi))
...                 limits={'xmin': 10, 'xmax': 90},
...                 datamin=-0.9, datamax=2.0,
...                 title="Matplotlib1DViewer test")
>>> for kval in numerix.arange(0,0.3,0.03):
...     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*).

legend place a legend at the specified position, if not *None*

axes if not *None*, *vars* will be plotted into this Matplotlib *Axes* object

log
logarithmic data scaling

class `fipy.viewers.matplotlibViewer.`**`Matplotlib2DGridViewer`**(*vars*, *title=None*, *limits={}*, *cmap=None*, *colorbar='vertical'*, *axes=None*, *figaspect='auto'*, ***kwlimits*)

Bases: `fipy.viewers.matplotlibViewer.matplotlib2DViewer.AbstractMatplotlib2DViewer`

Displays an image plot of a 2D *CellVariable* object using Matplotlib.

```
>>> from fipy import *
>>> mesh = Grid2D(nx=50, ny=100, dx=0.1, dy=0.01)
>>> x, y = mesh.cellCenters
>>> xyVar = CellVariable(mesh=mesh, name="x y", value=x * y)
>>> k = Variable(name="k", value=0.)
>>> viewer = Matplotlib2DGridViewer(vars=numerix.sin(k * xyVar),
...                 limits={'ymin': 0.1, 'ymax': 0.9},
...                 datamin=-0.9, datamax=2.0,
...                 title="Matplotlib2DGridViewer test")
>>> 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 *matplotlib.cm.jet*

xmin, xmax, ymin, ymax, datamin, datamax displayed range of data. Any limit set to a (default) value of *None* will autoscale.

colorbar plot a colorbar in specified orientation if not *None*

axes if not *None*, *vars* will be plotted into this Matplotlib *Axes* object

figaspect desired aspect ratio of figure. If arg is a number, use that aspect ratio. If arg is 'auto', the aspect ratio will be determined from the Variable's mesh.

class `fipy.viewers.matplotlibViewer.`**`Matplotlib2DGridContourViewer`**(*vars*, *title=None*, *limits={}*, *cmap=None*, *colorbar='vertical'*, *axes=None*, *figaspect='auto'*, ***kwlimits*)

Bases: `fipy.viewers.matplotlibViewer.matplotlib2DViewer.AbstractMatplotlib2DViewer`

Displays a contour plot of a 2D *CellVariable* object.

The *Matplotlib2DGridContourViewer* plots a 2D *CellVariable* using Matplotlib.

```
>>> from fipy import *
>>> mesh = Grid2D(nx=50, ny=100, dx=0.1, dy=0.01)
>>> x, y = mesh.cellCenters
>>> xyVar = CellVariable(mesh=mesh, name="x y", value=x * y)
>>> k = Variable(name="k", value=0.)
>>> viewer = Matplotlib2DGridContourViewer(vars=numerix.sin(k * xyVar),
...                 limits={'ymin': 0.1, 'ymax': 0.9},
...                 datamin=-0.9, datamax=2.0,
...                 title="Matplotlib2DGridContourViewer test")
>>> 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.

cmap the colormap. Defaults to *matplotlib.cm.jet*

colorbar plot a colorbar in specified orientation if not *None*

axes if not *None*, *vars* will be plotted into this Matplotlib *Axes* object

figaspect desired aspect ratio of figure. If arg is a number, use that aspect ratio. If arg is 'auto', the aspect ratio will be determined from the Variable's mesh.

class `fipy.viewers.matplotlibViewer.`**`Matplotlib2DViewer`**(*vars*, *title=None*, *limits={}*, *cmap=None*, *colorbar='vertical'*, *axes=None*, *figaspect='auto'*, ***kwlimits*)

Bases: `fipy.viewers.matplotlibViewer.matplotlib2DViewer.AbstractMatplotlib2DViewer`

Displays a contour plot of a 2D *CellVariable* object.

The *Matplotlib2DViewer* plots a 2D *CellVariable* using Matplotlib.

```
>>> from fipy import *
>>> mesh = (Grid2D(nx=5, ny=10, dx=0.1, dy=0.1)
...         + (Tri2D(nx=5, ny=5, dx=0.1, dy=0.1)
...          + ((0.5,), (0.2,))))
>>> x, y = mesh.cellCenters
>>> xyVar = CellVariable(mesh=mesh, name="x y", value=x * y)
>>> k = Variable(name="k", value=0.)
>>> viewer = Matplotlib2DViewer(vars=numerix.sin(k * xyVar),
...                 limits={'ymin': 0.1, 'ymax': 0.9},
...                 datamin=-0.9, datamax=2.0,
...                 title="Matplotlib2DViewer test")
>>> 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 *matplotlib.cm.jet*

xmin, xmax, ymin, ymax, datamin, datamax displayed range of data. Any limit set to a (default) value of *None* will autoscale.

colorbar plot a colorbar in specified orientation if not *None*

axes if not *None*, *vars* will be plotted into this Matplotlib *Axes* object

figaspect desired aspect ratio of figure. If arg is a number, use that aspect ratio. If arg is 'auto', the aspect ratio will be determined from the Variable's mesh.

class `fipy.viewers.matplotlibViewer.`**`MatplotlibVectorViewer`**(*vars*, *title=None*, *scale=None*, *sparsity=None*, *log=False*, *limits={}*, *axes=None*, *figaspect='auto'*, ***kwlimits*)

Bases: `fipy.viewers.matplotlibViewer.matplotlib2DViewer.AbstractMatplotlib2DViewer`

Displays a vector plot of a 2D rank-1 *CellVariable* or *FaceVariable* object using Matplotlib

```
>>> from fipy import *
>>> mesh = Grid2D(nx=50, ny=100, dx=0.1, dy=0.01)
>>> x, y = mesh.cellCenters
>>> xyVar = CellVariable(mesh=mesh, name="x y", value=x * y)
>>> k = Variable(name="k", value=1.)
>>> viewer = MatplotlibVectorViewer(vars=numerix.sin(k * xyVar).grad,
...                 title="MatplotlibVectorViewer test")
>>> for kval in numerix.arange(1, 10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()
```

```
>>> viewer = MatplotlibVectorViewer(vars=numerix.sin(k * xyVar).faceGrad,
...                 title="MatplotlibVectorViewer test")
>>> for kval in numerix.arange(1, 10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()

>>> for sparsity in numerix.arange(5000, 0, -500):
...     viewer.quiver(sparsity=sparsity)
...     viewer.plot()
>>> viewer._promptForOpinion()

>>> from fipy import *
>>> mesh = (Grid2D(nx=5, ny=10, dx=0.1, dy=0.1)
...         + (Tri2D(nx=5, ny=5, dx=0.1, dy=0.1)
...          + ((0.5,), (0.2,))))
>>> x, y = mesh.cellCenters
>>> xyVar = CellVariable(mesh=mesh, name="x y", value=x * y)
>>> k = Variable(name="k", value=1.)
>>> viewer = MatplotlibVectorViewer(vars=numerix.sin(k * xyVar).grad,
...                 title="MatplotlibVectorViewer test")
>>> for kval in numerix.arange(1, 10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()

>>> viewer = MatplotlibVectorViewer(vars=numerix.sin(k * xyVar).faceGrad,
...                 title="MatplotlibVectorViewer test")
>>> for kval in numerix.arange(1, 10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()
```

Creates a *Matplotlib2DViewer*.

Parameters

vars 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)

log if *True*, arrow length goes at the base-10 logarithm of the magnitude

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.

axes if not *None*, *vars* will be plotted into this Matplotlib *Axes* object

figaspect desired aspect ratio of figure. If arg is a number, use that aspect ratio. If arg is 'auto', the aspect ratio will be determined from the Variable's mesh.

quiver(*sparsity=None*, *scale=None*)

`matplotlib1DViewer` Module

class `fipy.viewers.matplotlibViewer.matplotlib1DViewer.`**`Matplotlib1DViewer`**(*vars*, *title=None*, *xlog=False*, *ylog=False*, *limits={}*, *legend='upper left'*, *axes=None*, ***kwlimits*)

Bases: `fipy.viewers.matplotlibViewer.matplotlibViewer.AbstractMatplotlibViewer`

Displays a y vs. x plot of one or more 1D *CellVariable* objects using Matplotlib.

```
>>> from fipy import *
>>> mesh = Grid1D(nx=100)
>>> x, = mesh.cellCenters
>>> xVar = CellVariable(mesh=mesh, name="x", value=x)
>>> k = Variable(name="k", value=0.)
>>> viewer = Matplotlib1DViewer(vars=(numerix.sin(k * xVar), numerix.cos(k * xVar / numerix.pi))
...                     limits={'xmin': 10, 'xmax': 90},
...                     datamin=-0.9, datamax=2.0,
...                     title="Matplotlib1DViewer test")
>>> for kval in numerix.arange(0,0.3,0.03):
...     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*).

legend place a legend at the specified position, if not *None*

axes if not *None*, *vars* will be plotted into this Matplotlib *Axes* object

`log`

logarithmic data scaling

`matplotlib2DContourViewer` Module

`matplotlib2DGridContourViewer` Module

class `fipy.viewers.matplotlibViewer.matplotlib2DGridContourViewer.`**`Matplotlib2DGridContourView`**

Bases: `fipy.viewers.matplotlibViewer.matplotlib2DViewer.AbstractMatplotlib2DViewer`

Displays a contour plot of a 2D *CellVariable* object.

The *Matplotlib2DGridContourViewer* plots a 2D *CellVariable* using Matplotlib.

```
>>> from fipy import *
>>> mesh = Grid2D(nx=50, ny=100, dx=0.1, dy=0.01)
>>> x, y = mesh.cellCenters
>>> xyVar = CellVariable(mesh=mesh, name="x y", value=x * y)
>>> k = Variable(name="k", value=0.)
>>> viewer = Matplotlib2DGridContourViewer(vars=numerix.sin(k * xyVar),
...                 limits={'ymin': 0.1, 'ymax': 0.9},
...                 datamin=-0.9, datamax=2.0,
...                 title="Matplotlib2DGridContourViewer test")
>>> 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.

cmap the colormap. Defaults to *matplotlib.cm.jet*

colorbar plot a colorbar in specified orientation if not *None*

axes if not *None*, *vars* will be plotted into this Matplotlib *Axes* object

figaspect desired aspect ratio of figure. If arg is a number, use that aspect ratio. If arg is 'auto', the aspect ratio will be determined from the Variable's mesh.

`matplotlib2DGridViewer` Module

class `fipy.viewers.matplotlibViewer.matplotlib2DGridViewer.`**`Matplotlib2DGridViewer`**(*vars, title=None, limits={}, cmap=None, colorbar='vertical', axes=None, figaspect='auto', **kwlimits*)

Bases: `fipy.viewers.matplotlibViewer.matplotlib2DViewer.AbstractMatplotlib2DViewer`

Displays an image plot of a 2D *CellVariable* object using Matplotlib.

```
>>> from fipy import *
>>> mesh = Grid2D(nx=50, ny=100, dx=0.1, dy=0.01)
>>> x, y = mesh.cellCenters
>>> xyVar = CellVariable(mesh=mesh, name="x y", value=x * y)
>>> k = Variable(name="k", value=0.)
>>> viewer = Matplotlib2DGridViewer(vars=numerix.sin(k * xyVar),
...                   limits={'ymin': 0.1, 'ymax': 0.9},
...                   datamin=-0.9, datamax=2.0,
...                   title="Matplotlib2DGridViewer test")
>>> 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 *matplotlib.cm.jet*

xmin, xmax, ymin, ymax, datamin, datamax displayed range of data. Any limit set to a (default) value of *None* will autoscale.

colorbar plot a colorbar in specified orientation if not *None*

axes if not *None*, *vars* will be plotted into this Matplotlib *Axes* object

figaspect desired aspect ratio of figure. If arg is a number, use that aspect ratio. If arg is 'auto', the aspect ratio will be determined from the Variable's mesh.

matplotlib2DViewer Module

class fipy.viewers.matplotlibViewer.matplotlib2DViewer.**Matplotlib2DViewer**(*vars*, *title=None*, *limits={}*, *cmap=None*, *colorbar='vertical'*, *axes=None*, *figaspect='auto'*, ***kwlimits*)

Bases: fipy.viewers.matplotlibViewer.matplotlib2DViewer.AbstractMatplotlib2DViewer

Displays a contour plot of a 2D *CellVariable* object.

The *Matplotlib2DViewer* plots a 2D *CellVariable* using Matplotlib.

```
>>> from fipy import *
>>> mesh = (Grid2D(nx=5, ny=10, dx=0.1, dy=0.1)
...         + (Tri2D(nx=5, ny=5, dx=0.1, dy=0.1)
...          + ((0.5,), (0.2,))))
>>> x, y = mesh.cellCenters
>>> xyVar = CellVariable(mesh=mesh, name="x y", value=x * y)
>>> k = Variable(name="k", value=0.)
>>> viewer = Matplotlib2DViewer(vars=numerix.sin(k * xyVar),
...                     limits={'ymin': 0.1, 'ymax': 0.9},
...                     datamin=-0.9, datamax=2.0,
...                     title="Matplotlib2DViewer test")
>>> 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 *matplotlib.cm.jet*

xmin, xmax, ymin, ymax, datamin, datamax displayed range of data. Any limit set to a (default) value of *None* will autoscale.

colorbar plot a colorbar in specified orientation if not *None*

axes if not *None*, *vars* will be plotted into this Matplotlib *Axes* object

figaspect desired aspect ratio of figure. If arg is a number, use that aspect ratio. If arg is 'auto', the aspect ratio will be determined from the Variable's mesh.

`matplotlibSparseMatrixViewer` Module

class `fipy.viewers.matplotlibViewer.matplotlibSparseMatrixViewer.`**`MatplotlibSparseMatrixViewer`**

`plot` (*matrix*, *RHSvector*, *log='auto'*)

`matplotlibVectorViewer` Module

class `fipy.viewers.matplotlibViewer.matplotlibVectorViewer.`**`MatplotlibVectorViewer`**(*vars*, *title=None*, *scale=None*, *sparsity=None*, *log=False*, *limits={}*, *axes=None*, *figaspect='auto'*, ***kwlimits*)

Bases: `fipy.viewers.matplotlibViewer.matplotlib2DViewer.AbstractMatplotlib2DViewer`

Displays a vector plot of a 2D rank-1 *CellVariable* or *FaceVariable* object using Matplotlib

```
>>> from fipy import *
>>> mesh = Grid2D(nx=50, ny=100, dx=0.1, dy=0.01)
>>> x, y = mesh.cellCenters
>>> xyVar = CellVariable(mesh=mesh, name="x y", value=x * y)
>>> k = Variable(name="k", value=1.)
>>> viewer = MatplotlibVectorViewer(vars=numerix.sin(k * xyVar).grad,
...                 title="MatplotlibVectorViewer test")
>>> for kval in numerix.arange(1, 10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()
```

```
>>> viewer = MatplotlibVectorViewer(vars=numerix.sin(k * xyVar).faceGrad,
...                 title="MatplotlibVectorViewer test")
>>> for kval in numerix.arange(1, 10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()
```

```
>>> for sparsity in numerix.arange(5000, 0, -500):
...     viewer.quiver(sparsity=sparsity)
...     viewer.plot()
>>> viewer._promptForOpinion()
```

```
>>> from fipy import *
>>> mesh = (Grid2D(nx=5, ny=10, dx=0.1, dy=0.1)
...         + (Tri2D(nx=5, ny=5, dx=0.1, dy=0.1)
...          + ((0.5,), (0.2,))))
>>> x, y = mesh.cellCenters
>>> xyVar = CellVariable(mesh=mesh, name="x y", value=x * y)
```

```
>>> k = Variable(name="k", value=1.)
>>> viewer = MatplotlibVectorViewer(vars=numerix.sin(k * xyVar).grad,
...                 title="MatplotlibVectorViewer test")
>>> for kval in numerix.arange(1, 10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()

>>> viewer = MatplotlibVectorViewer(vars=numerix.sin(k * xyVar).faceGrad,
...                 title="MatplotlibVectorViewer test")
>>> for kval in numerix.arange(1, 10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()
```

Creates a *Matplotlib2DViewer*.

Parameters

vars 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)

log if *True*, arrow length goes at the base-10 logarithm of the magnitude

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.

axes if not *None*, *vars* will be plotted into this Matplotlib *Axes* object

figaspect desired aspect ratio of figure. If arg is a number, use that aspect ratio. If arg is 'auto', the aspect ratio will be determined from the Variable's mesh.

quiver(*sparsity=None*, *scale=None*)

`matplotlibViewer` Module

class fipy.viewers.matplotlibViewer.matplotlibViewer.**AbstractMatplotlibViewer**(*vars*, *title=None*, *figaspect=1.0*, *cmap=None*, *colorbar=None*, *axes=None*, *log=False*, ***kwlimits*)

Bases: `fipy.viewers.viewer.AbstractViewer`

Attention: This class is abstract. Always create one of its subclasses.

The *AbstractMatplotlibViewer* is the base class for the viewers that use the Matplotlib python plotting package.

Create a *AbstractMatplotlibViewer*.

Parameters

vars a *CellVariable* or tuple of *CellVariable* objects to plot

title displayed at the top of the *Viewer* window

figaspect desired aspect ratio of figure. If arg is a number, use that aspect ratio. If arg is an array, figaspect will determine the width and height for a figure that would fit array preserving aspect ratio.

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.

cmap the colormap. Defaults to *matplotlib.cm.jet*

colorbar plot a colorbar in specified orientation if not *None*

axes if not *None*, *vars* will be plotted into this Matplotlib *Axes* object

log whether to logarithmically scale the data

figaspect (*figaspect*)

log
logarithmic data scaling

plot (*filename=None*)

test Module

Test numeric implementation of the mesh

29.7.4 mayaviViewer Package

mayaviViewer Package

class fipy.viewers.mayaviViewer.**MayaviClient** (*vars*, *title=None*, *daemon_file=None*, *fps=1.0*, ***kwlimits*)

Bases: fipy.viewers.viewer.AbstractViewer

The *MayaviClient* uses the Mayavi python plotting package.

```
>>> from fipy import *
>>> mesh = Grid1D(nx=100)
>>> x, = mesh.cellCenters
>>> xVar = CellVariable(mesh=mesh, name="x", value=x)
>>> k = Variable(name="k", value=0.)
>>> viewer = MayaviClient(vars=(numerix.sin(k * xVar), numerix.cos(k * xVar / numerix.pi)),
...                       limits={'xmin': 10, 'xmax': 90},
...                       datamin=-0.9, datamax=2.0,
...                       title="MayaviClient test")
>>> for kval in numerix.arange(0,0.3,0.03):
...     k.setValue(kval)
```

```
...     viewer.plot()
>>> viewer._promptForOpinion()

>>> from fipy import *
>>> mesh = Grid2D(nx=50, ny=100, dx=0.1, dy=0.01)
>>> x, y = mesh.cellCenters
>>> xyVar = CellVariable(mesh=mesh, name="x y", value=x * y)
>>> k = Variable(name="k", value=0.)
>>> viewer = MayaviClient(vars=numerix.sin(k * xyVar),
...                  limits={'ymin': 0.1, 'ymax': 0.9},
...                  datamin=-0.9, datamax=2.0,
...                  title="MayaviClient test")
>>> for kval in range(10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()

>>> from fipy import *
>>> mesh = (Grid2D(nx=5, ny=10, dx=0.1, dy=0.1)
...         + (Tri2D(nx=5, ny=5, dx=0.1, dy=0.1)
...          + ((0.5,), (0.2,))))
>>> x, y = mesh.cellCenters
>>> xyVar = CellVariable(mesh=mesh, name="x y", value=x * y)
>>> k = Variable(name="k", value=0.)
>>> viewer = MayaviClient(vars=numerix.sin(k * xyVar),
...                  limits={'ymin': 0.1, 'ymax': 0.9},
...                  datamin=-0.9, datamax=2.0,
...                  title="MayaviClient test")
>>> for kval in range(10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()

>>> from fipy import *
>>> mesh = Grid3D(nx=50, ny=100, nz=10, dx=0.1, dy=0.01, dz=0.1)
>>> x, y, z = mesh.cellCenters
>>> xyzVar = CellVariable(mesh=mesh, name=r"x y z", value=x * y * z)
>>> k = Variable(name="k", value=0.)
>>> viewer = MayaviClient(vars=numerix.sin(k * xyzVar),
...                      limits={'ymin': 0.1, 'ymax': 0.9},
...                      datamin=-0.9, datamax=2.0,
...                      title="MayaviClient test")
>>> for kval in range(10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()
```

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*)

mayaviClient Module

class fipy.viewers.mayaviViewer.mayaviClient.**MayaviClient** (*vars*, *title=None*, *daemon_file=None*, *fps=1.0*, ***kwlimits*)

Bases: fipy.viewers.viewer.AbstractViewer

The *MayaviClient* uses the Mayavi python plotting package.

```
>>> from fipy import *
>>> mesh = Grid1D(nx=100)
>>> x, = mesh.cellCenters
>>> xVar = CellVariable(mesh=mesh, name="x", value=x)
>>> k = Variable(name="k", value=0.)
>>> viewer = MayaviClient(vars=(numerix.sin(k * xVar), numerix.cos(k * xVar / numerix.pi)),
...                       limits={'xmin': 10, 'xmax': 90},
...                       datamin=-0.9, datamax=2.0,
...                       title="MayaviClient test")
>>> for kval in numerix.arange(0,0.3,0.03):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()
```

```
>>> from fipy import *
>>> mesh = Grid2D(nx=50, ny=100, dx=0.1, dy=0.01)
>>> x, y = mesh.cellCenters
>>> xyVar = CellVariable(mesh=mesh, name="x y", value=x * y)
>>> k = Variable(name="k", value=0.)
>>> viewer = MayaviClient(vars=numerix.sin(k * xyVar),
...                       limits={'ymin': 0.1, 'ymax': 0.9},
...                       datamin=-0.9, datamax=2.0,
...                       title="MayaviClient test")
>>> for kval in range(10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()
```

```
>>> from fipy import *
>>> mesh = (Grid2D(nx=5, ny=10, dx=0.1, dy=0.1)
...         + (Tri2D(nx=5, ny=5, dx=0.1, dy=0.1)
...          + ((0.5,), (0.2,))))
>>> x, y = mesh.cellCenters
>>> xyVar = CellVariable(mesh=mesh, name="x y", value=x * y)
>>> k = Variable(name="k", value=0.)
>>> viewer = MayaviClient(vars=numerix.sin(k * xyVar),
...                       limits={'ymin': 0.1, 'ymax': 0.9},
...                       datamin=-0.9, datamax=2.0,
...                       title="MayaviClient test")
>>> for kval in range(10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()
```

```
>>> from fipy import *
>>> mesh = Grid3D(nx=50, ny=100, nz=10, dx=0.1, dy=0.01, dz=0.1)
>>> x, y, z = mesh.cellCenters
>>> xyzVar = CellVariable(mesh=mesh, name=r"x y z", value=x * y * z)
>>> k = Variable(name="k", value=0.)
>>> viewer = MayaviClient(vars=numerix.sin(k * xyzVar),
...                       limits={'ymin': 0.1, 'ymax': 0.9},
...                       datamin=-0.9, datamax=2.0,
...                       title="MayaviClient test")
>>> for kval in range(10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()
```

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*)

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 exampe in the mayavi distribution.

This script is to be run like so:

```
$ mayavi2 -x mayaviDaemon.py <options>
```

Or:

```
$ python mayaviDaemon.py <options>
```

Run:

```
$ python mayaviDaemon.py --help
```

to see available options.

class fipy.viewers.mayaviViewer.mayaviDaemon.**MayaviDaemon**

Bases: mayavi.plugins.app.Mayavi

Given a file name and a mayavi2 data reader object, this class polls the file for any changes and automatically updates the mayavi pipeline.

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.

test Module

Test numeric implementation of the mesh

29.7.5 vtkViewer Package

vtkViewer Package

fipy.viewers.vtkViewer.**VTKViewer**(*vars*, *title=None*, *limits={}*, ***kwlimits*)
Generic function for creating a *VTKViewer*.

The *VTKViewer* factory will search the module tree and return an instance of the first *VTKViewer* it finds of the correct dimension and rank.

Parameters

vars 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 fipy.viewers.vtkViewer.**VTKCellViewer**(*vars*, *title=None*, *limits={}*, ***kwlimits*)
Bases: fipy.viewers.vtkViewer.vtkViewer.VTKViewer

Renders *CellVariable* data in VTK format

Creates a VTKViewer

Parameters

vars 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 `fipy.viewers.vtkViewer.`**`VTKFaceViewer`**(*vars*, *title=None*, *limits={}*, ***kwlimits*)

Bases: `fipy.viewers.vtkViewer.vtkViewer.VTKViewer`

Renders *_MeshVariable* data in VTK format

Creates a VTKViewer

Parameters

vars 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.

`test` Module

Test numeric implementation of the mesh

`vtkCellViewer` Module

class `fipy.viewers.vtkViewer.vtkCellViewer.`**`VTKCellViewer`**(*vars*, *title=None*, *limits={}*, ***kwlimits*)

Bases: `fipy.viewers.vtkViewer.vtkViewer.VTKViewer`

Renders *CellVariable* data in VTK format

Creates a VTKViewer

Parameters

vars 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.

`vtkFaceViewer` Module

class `fipy.viewers.vtkViewer.vtkFaceViewer.`**`VTKFaceViewer`**(*vars*, *title=None*, *limits={}*, ***kwlimits*)

Bases: `fipy.viewers.vtkViewer.vtkViewer.VTKViewer`

Renders *_MeshVariable* data in VTK format

Creates a VTKViewer

Parameters

vars 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.

`vtkViewer` Module

class `fipy.viewers.vtkViewer.vtkViewer.`**`VTKViewer`**(*vars*, *title=None*, *limits={}*, ***kwlimits*)

Bases: `fipy.viewers.viewer.AbstractViewer`

Renders *_MeshVariable* data in VTK format

Creates a VTKViewer

Parameters

vars 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.

`plot`(*filename=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.

[Elder:2011p2811] Elder, K R, K Thornton, and J J Hoyt. 2011. The Kirkendall effect in the phase field crystal model. *Philos Mag* 91, 151–164.

[ferziger] Ferziger, J H, and M Peric. (1996) *Computational Methods for Fluid Dynamics*. : Springer.

[ElPhFI] Guyer, J E, et al. 2004. Phase field modeling of electrochemistry I: Equilibrium. *pre* 69, 021603.

[ElPhFII] Guyer, J E, et al. 2004. Phase field modeling of electrochemistry II: Kinetics. *pre* 69, 021604.

[Hangarter:2011p2795] Hangarter, C M, et al. 2011. Three dimensionally structured interdigitated back contact thin film heterojunction solar cells. *Journal of Applied Physics* 109, 073514.

[NIST:damascene:2001] Josell, D, et al. 2001. Superconformal Electrodeposition in Submicron Features. *prl* 87, 016102.

[NIST:damascene:2005] Josell, D, D Wheeler, and T P Moffat. 2006. Gold superfill in submicrometer trenches: Experiment and prediction. *Journal of The Electrochemical Society* 153, C11-C18.

[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.

[NIST:leveler:2005] Moffat, T P, et al. 2006. Curvature enhanced adsorbate coverage model for electrodeposition. *Journal of The Electrochemical Society* 153, C127-C132.

[patankar] Patankar, S V. (1980) *Numerical Heat Transfer and Fluid Flow*. : Taylor and Francis.

[DiveIntoPython] Pilgrim, Mark. (2004) *Dive Into Python*. : Apress.

[NumericalRecipes] Press, William H, et al. (1999) *Numerical Recipes in C: the Art of Scientific Computing*. : Cambridge University Press.

[PythonTutorial] van Rossum, Guido. . Python Tutorial.

[Saylor:2011p2794] Saylor, David M, et al. 2011. Predicting microstructure development during casting of drug-eluting coatings. *Acta Biomater* 7, 604–613.

[versteegMalalasekera] Versteeg, H K, and W Malalasekera. (1995) *An Introduction to Computational Fluid Dynamics*. : Longman Scientific and Technical.

[InstallingPythonModules] Ward, Greg. . Installing Python Modules.

[Warren:1995] Warren, J A, and W J Boettinger. 1995. Prediction of Dendritic Growth and Microsegregation in a Binary Alloy using the Phase Field Method. *Acta Metallurgica et Materialia* 43, 689-703.

[WarrenPolycrystal] Warren, James A, et al. 2003. Extending Phase Field Models of Solidification to Polycrystalline Materials. *Acta Materialia* 51, 6035-6058.

[Wheeler:1992] Wheeler, A A, W J Boettinger, and G B McF. 1992. Phase-field model for isothermal phase transitions in binary alloys. *pra* 45, 7424–7439.

[NIST:damascene:2003] Wheeler, D, D Josell, and T P Moffat. 2003. Modeling Superconformal Electrodeposition Using The Level Set Method. *Journal of The Electrochemical Society* 150, C302-C310.

[PhysRevE.82.051601] Wheeler, Daniel, James A Warren, and William J Boettinger. 2010. Modeling the early stages of reactive wetting. *Phys. Rev. E* 82, 051601.

Python Module Index

e

examples.cahnHilliard.mesh2DCoupled, 157
examples.cahnHilliard.sphere, 160
examples.convection.exponential1D.mesh1D, 85
examples.convection.exponential1DSource.mesh1D, 86
examples.convection.robin, 87
examples.convection.source, 89
examples.diffusion.anisotropy, 82
examples.diffusion.circle, 71
examples.diffusion.coupled, 67
examples.diffusion.electrostatics, 76
examples.diffusion.mesh1D, 55
examples.diffusion.mesh20x20, 69
examples.diffusion.nthOrder.input4thOrder1D, 80
examples.flow.stokesCavity, 163
examples.levelSet.advection.circle, 138
examples.levelSet.advection.mesh1D, 137
examples.levelSet.distanceFunction.circle, 136
examples.levelSet.distanceFunction.mesh1D, 135
examples.levelSet.electroChem.gold, 144
examples.levelSet.electroChem.howToWriteAScript, 149
examples.levelSet.electroChem.leveler, 145
examples.levelSet.electroChem.simpleTrenchSystem, 141
examples.phase.anisotropy, 113
examples.phase.binaryCoupled, 99
examples.phase.impingement.mesh20x20, 120
examples.phase.impingement.mesh40x1, 117
examples.phase.polyxtal, 123
examples.phase.polyxtalCoupled, 129
examples.phase.quaternary, 107
examples.phase.simple, 91
examples.reactiveWetting.liquidVapor1D, 169
examples.updating.update0_1to1_0, 180
examples.updating.update1_0to2_0, 176
examples.updating.update2_0to3_0, 175

f

fipy.boundaryConditions, 191
fipy.boundaryConditions.boundaryCondition, 192
fipy.boundaryConditions.constraint, 192
fipy.boundaryConditions.fixedFlux, 192
fipy.boundaryConditions.fixedValue, 192
fipy.boundaryConditions.nthOrderBoundaryCondition, 193
fipy.boundaryConditions.test, 193
fipy.matrices.offsetSparseMatrix, 195
fipy.matrices.pysparseMatrix, 195
fipy.matrices.scipyMatrix, 195
fipy.matrices.sparseMatrix, 195
fipy.matrices.test, 195
fipy.matrices.trilinosMatrix, 195
fipy.meshes, 197
fipy.meshes.abstractMesh, 205
fipy.meshes.builders, 225
fipy.meshes.builders.abstractGridBuilder, 225
fipy.meshes.builders.grid1DBuilder, 225
fipy.meshes.builders.grid2DBuilder, 225
fipy.meshes.builders.grid3DBuilder, 225
fipy.meshes.builders.periodicGrid1DBuilder, 225
fipy.meshes.builders.utilityClasses, 225
fipy.meshes.cylindricalGrid1D, 209
fipy.meshes.cylindricalGrid2D, 210
fipy.meshes.cylindricalUniformGrid1D, 210
fipy.meshes.cylindricalUniformGrid2D, 211
fipy.meshes.factoryMeshes, 211
fipy.meshes.gmshMesh, 213

fipy.meshes.grid1D, 218
fipy.meshes.grid2D, 218
fipy.meshes.grid3D, 219
fipy.meshes.mesh, 219
fipy.meshes.mesh1D, 219
fipy.meshes.mesh2D, 220
fipy.meshes.numMesh.cylindricalGrid1D, 225
fipy.meshes.numMesh.cylindricalGrid2D, 225
fipy.meshes.numMesh.cylindricalUniformGrid1D, 225
fipy.meshes.numMesh.cylindricalUniformGrid2D, 225
fipy.meshes.numMesh.deprecatedWarning, 225
fipy.meshes.numMesh.gmshImport, 226
fipy.meshes.numMesh.grid1D, 226
fipy.meshes.numMesh.grid2D, 226
fipy.meshes.numMesh.grid3D, 226
fipy.meshes.numMesh.periodicGrid1D, 226
fipy.meshes.numMesh.periodicGrid2D, 226
fipy.meshes.numMesh.skewedGrid2D, 226
fipy.meshes.numMesh.tri2D, 226
fipy.meshes.numMesh.uniformGrid1D, 226
fipy.meshes.numMesh.uniformGrid2D, 226
fipy.meshes.numMesh.uniformGrid3D, 226
fipy.meshes.periodicGrid1D, 220
fipy.meshes.periodicGrid2D, 221
fipy.meshes.representations.abstractRepresentation, 226
fipy.meshes.representations.gridRepresentation, 226
fipy.meshes.representations.meshRepresentation, 226
fipy.meshes.skewedGrid2D, 222
fipy.meshes.test, 222
fipy.meshes.topologies.abstractTopology, 226
fipy.meshes.topologies.gridTopology, 226
fipy.meshes.topologies.meshTopology, 226
fipy.meshes.tri2D, 223
fipy.meshes.uniformGrid, 223
fipy.meshes.uniformGrid1D, 223
fipy.meshes.uniformGrid2D, 224
fipy.meshes.uniformGrid3D, 224
fipy.models, 227
fipy.models.levelSet, 242
fipy.models.levelSet.advection, 257
fipy.models.levelSet.advection.advectionEquation, 258
fipy.models.levelSet.advection.advectionTerm, 258
fipy.models.levelSet.advection.higherOrderAdvection 258
fipy.models.levelSet.advection.higherOrderAdvection 258
fipy.models.levelSet.distanceFunction, 258
fipy.models.levelSet.distanceFunction.distanceVaria 262
fipy.models.levelSet.distanceFunction.levelSetDiffu 265
fipy.models.levelSet.distanceFunction.levelSetDiffu 266
fipy.models.levelSet.electroChem, 266
fipy.models.levelSet.electroChem.gapFillMesh, 268
fipy.models.levelSet.electroChem.metalIonDiffusionE 270
fipy.models.levelSet.electroChem.metalIonSourceVari 272
fipy.models.levelSet.electroChem.test, 272
fipy.models.levelSet.surfactant, 272
fipy.models.levelSet.surfactant.adsorbingSurfactant 281
fipy.models.levelSet.surfactant.convectionCoeff, 285
fipy.models.levelSet.surfactant.lines, 285
fipy.models.levelSet.surfactant.matplotlibSurfactan 286
fipy.models.levelSet.surfactant.mayaviSurfactantVie 287
fipy.models.levelSet.surfactant.surfactantBulkDiffu 288
fipy.models.levelSet.surfactant.surfactantEquation, 290
fipy.models.levelSet.surfactant.surfactantVariable, 290
fipy.models.levelSet.test, 257
fipy.models.test, 242
fipy.solvers, 293
fipy.solvers.pyAMG, 296
fipy.solvers.pyAMG.linearCGSSolver, 298
fipy.solvers.pyAMG.linearGeneralSolver, 298
fipy.solvers.pyAMG.linearGMRESSolver, 298
fipy.solvers.pyAMG.linearLUSolver, 299
fipy.solvers.pyAMG.linearPCGSolver, 299
fipy.solvers.pyAMG.preconditioners, 299
fipy.solvers.pyAMG.preconditioners.smoothedAggregat 299
fipy.solvers.pysparse, 299

fipy.solvers.pysparse.linearCGSSolver, 301
fipy.solvers.pysparse.linearGMRESSolver, 301
fipy.solvers.pysparse.linearJORSolver, 302
fipy.solvers.pysparse.linearLUSolver, 302
fipy.solvers.pysparse.linearPCGSolver, 302
fipy.solvers.pysparse.preconditioners, 303
fipy.solvers.pysparse.preconditioners.jacobiPreconditioner, 303
fipy.solvers.pysparse.preconditioners.preconditioner, 303
fipy.solvers.pysparse.preconditioners.ssorPreconditioner, 304
fipy.solvers.pysparse.pysparseSolver, 303
fipy.solvers.pysparseMatrixSolver, 295
fipy.solvers.scipy, 304
fipy.solvers.scipy.linearBicgstabSolver, 305
fipy.solvers.scipy.linearCGSSolver, 305
fipy.solvers.scipy.linearGMRESSolver, 306
fipy.solvers.scipy.linearLUSolver, 306
fipy.solvers.scipy.linearPCGSolver, 306
fipy.solvers.scipy.scipyKrylovSolver, 307
fipy.solvers.scipy.scipySolver, 307
fipy.solvers.solver, 295
fipy.solvers.test, 296
fipy.solvers.trilinos, 307
fipy.solvers.trilinos.linearBicgstabSolver, 310
fipy.solvers.trilinos.linearCGSSolver, 310
fipy.solvers.trilinos.linearGMRESSolver, 310
fipy.solvers.trilinos.linearLUSolver, 311
fipy.solvers.trilinos.linearPCGSolver, 311
fipy.solvers.trilinos.preconditioners, 313
fipy.solvers.trilinos.preconditioners.domDecompPreconditioner, 315
fipy.solvers.trilinos.preconditioners.icPreconditioner, 315
fipy.solvers.trilinos.preconditioners.jacobiPreconditioner, 315
fipy.solvers.trilinos.preconditioners.multilevelDDM 315
fipy.solvers.trilinos.preconditioners.multilevelDDP 315
fipy.solvers.trilinos.preconditioners.multilevelNSS 315
fipy.solvers.trilinos.preconditioners.multilevelSAP 315
fipy.solvers.trilinos.preconditioners.multilevelSGS 316
fipy.solvers.trilinos.preconditioners.multilevelSol 316
fipy.solvers.trilinos.preconditioners.preconditione 316
fipy.solvers.trilinos.trilinosAztecOOSolver, 311
fipy.solvers.trilinos.trilinosMLTest, 312
fipy.solvers.trilinos.trilinosNonlinearSolver, 313
fipy.solvers.trilinos.trilinosSolver, 313
fipy.steppers, 317
fipy.steppers.pidStepper, 318
fipy.steppers.pseudoRKQSStepper, 318
fipy.steppers.stepper, 318
fipy.terms, 319
fipy.terms.abstractBinaryTerm, 329
fipy.terms.abstractConvectionTerm, 329
fipy.terms.abstractDiffusionTerm, 329
fipy.terms.abstractUpwindConvectionTerm, 329
fipy.terms.asymmetricConvectionTerm, 329
fipy.terms.binaryTerm, 329
fipy.terms.cellTerm, 329
fipy.terms.centralDiffConvectionTerm, 329
fipy.terms.coupledBinaryTerm, 330
fipy.terms.diffusionTerm, 330
fipy.terms.diffusionTermCorrection, 331
fipy.terms.diffusionTermNoCorrection, 331
fipy.terms.explicitDiffusionTerm, 331
fipy.terms.explicitSourceTerm, 331
fipy.terms.explicitUpwindConvectionTerm, 331
fipy.terms.exponentialConvectionTerm, 332
fipy.terms.faceTerm, 334
fipy.terms.hybridConvectionTerm, 334
fipy.terms.implicitDiffusionTerm, 335
fipy.terms.implicitSourceTerm, 335
fipy.terms.nonDiffusionTerm, 335
fipy.terms.powerLawConvectionTerm, 335

fipy.terms.residualTerm, 336
fipy.terms.sourceTerm, 337
fipy.terms.term, 337
fipy.terms.test, 339
fipy.terms.transientTerm, 339
fipy.terms.unaryTerm, 340
fipy.terms.upwindConvectionTerm, 340
fipy.terms.vanLeerConvectionTerm, 342
fipy.tests, 343
fipy.tests.doctestPlus, 343
fipy.tests.lateImportTest, 344
fipy.tests.testBase, 344
fipy.tests.testClass, 344
fipy.tests.testProgram, 344
fipy.tools, 345
fipy.tools.comms.commWrapper, 365
fipy.tools.comms.dummyComm, 366
fipy.tools.comms.mpi4pyCommWrapper, 366
fipy.tools.comms.serialCommWrapper, 366
fipy.tools.copy_script, 355
fipy.tools.debug, 355
fipy.tools.decorators, 355
fipy.tools.dimensions.DictWithDefault, 367
fipy.tools.dimensions.NumberDict, 367
fipy.tools.dimensions.physicalField, 367
fipy.tools.dump, 356
fipy.tools.inline, 357
fipy.tools.numerix, 357
fipy.tools.parser, 364
fipy.tools.performance.efficiency_test, 381
fipy.tools.performance.efficiencyTestGenerator, 381
fipy.tools.performance.efficiencyTestHistory, 381
fipy.tools.performance.memoryLeak, 381
fipy.tools.performance.memoryLogger, 381
fipy.tools.performance.memoryUsage, 381
fipy.tools.test, 364
fipy.tools.vector, 364
fipy.tools.vitals, 365
fipy.variables, 383
fipy.variables.addOverFacesVariable, 403
fipy.variables.arithmeticCellToFaceVariable, 403
fipy.variables.betaNoiseVariable, 403
fipy.variables.binaryOperatorVariable, 405
fipy.variables.cellToFaceVariable, 405
fipy.variables.cellVariable, 405
fipy.variables.cellVolumeAverageVariable, 411
fipy.variables.constant, 411
fipy.variables.constraintMask, 411
fipy.variables.coupledCellVariable, 411
fipy.variables.exponentialNoiseVariable, 411
fipy.variables.faceGradContributionsVariable, 412
fipy.variables.faceGradVariable, 412
fipy.variables.faceVariable, 412
fipy.variables.gammaNoiseVariable, 413
fipy.variables.gaussCellGradVariable, 415
fipy.variables.gaussianNoiseVariable, 415
fipy.variables.harmonicCellToFaceVariable, 417
fipy.variables.histogramVariable, 417
fipy.variables.leastSquaresCellGradVariable, 417
fipy.variables.meshVariable, 417
fipy.variables.minmodCellToFaceVariable, 417
fipy.variables.modCellGradVariable, 417
fipy.variables.modCellToFaceVariable, 417
fipy.variables.modFaceGradVariable, 417
fipy.variables.modPhysicalField, 417
fipy.variables.modularVariable, 417
fipy.variables.noiseVariable, 418
fipy.variables.operatorVariable, 419
fipy.variables.scharfetterGummelFaceVariable, 419
fipy.variables.test, 419
fipy.variables.unaryOperatorVariable, 419
fipy.variables.uniformNoiseVariable, 419
fipy.variables.variable, 420
fipy.viewers, 429
fipy.viewers.gistViewer, 442
fipy.viewers.gistViewer.gist1DViewer, 442
fipy.viewers.gistViewer.gist2DViewer, 442
fipy.viewers.gistViewer.gistVectorViewer, 442
fipy.viewers.gistViewer.gistViewer, 442
fipy.viewers.gistViewer.test, 442
fipy.viewers.gnuplotViewer, 442
fipy.viewers.gnuplotViewer.gnuplot1DViewer, 443
fipy.viewers.gnuplotViewer.gnuplot2DViewer, 443
fipy.viewers.gnuplotViewer.gnuplotViewer, 443
fipy.viewers.gnuplotViewer.test, 443

fipy.viewers.matplotlibViewer, 443
fipy.viewers.matplotlibViewer.matplotlib1DViewer, 449
fipy.viewers.matplotlibViewer.matplotlib2DGridContourViewer, 450
fipy.viewers.matplotlibViewer.matplotlib2DGridViewer, 451
fipy.viewers.matplotlibViewer.matplotlib2DViewer, 452
fipy.viewers.matplotlibViewer.matplotlibSparseMatrixViewer, 453
fipy.viewers.matplotlibViewer.matplotlibVectorViewer, 453
fipy.viewers.matplotlibViewer.matplotlibViewer, 454
fipy.viewers.matplotlibViewer.test, 455
fipy.viewers.mayaviViewer, 455
fipy.viewers.mayaviViewer.mayaviClient, 457
fipy.viewers.mayaviViewer.mayaviDaemon, 458
fipy.viewers.mayaviViewer.test, 459
fipy.viewers.multiViewer, 439
fipy.viewers.test, 439
fipy.viewers.testinteractive, 439
fipy.viewers.tsvViewer, 440
fipy.viewers.viewer, 441
fipy.viewers.vtkViewer, 459
fipy.viewers.vtkViewer.test, 460
fipy.viewers.vtkViewer.vtkCellViewer, 460
fipy.viewers.vtkViewer.vtkFaceViewer, 460
fipy.viewers.vtkViewer.vtkViewer, 461

p

package.subpackage, 187
package.subpackage.base, 187
package.subpackage.object, 188

Index

Symbols

–inline
    command line option, 22
–no-pysparse
    command line option, 22
–pyamg
    command line option, 23
–pysparse
    command line option, 22
–scipy
    command line option, 22
–trilinos
    command line option, 22
:math:‘\pi‘, 114, 119, 121
:module: viewers, 112

A

AbstractBaseClassError, 319
AbstractMatplotlibViewer (class in fipy.viewers.matplotlibViewer.matplotlibViewer), 454
AbstractMesh (class in fipy.meshes.abstractMesh), 205
AbstractViewer (class in fipy.viewers.viewer), 441
add() (fipy.tools.dimensions.physicalField.PhysicalField method), 370
add() (fipy.tools.PhysicalField method), 346
AdsorbingSurfactantEquation (class in fipy.models), 232
AdsorbingSurfactantEquation (class in fipy.models.levelSet), 247
AdsorbingSurfactantEquation (class in fipy.models.levelSet.surfactant), 274
AdsorbingSurfactantEquation (class in fipy.models.levelSet.surfactant.adsorbingSurfactantEquation), 281
all() (fipy.tools.comms.commWrapper.CommWrapper method), 365
all() (fipy.tools.comms.mpi4pyCommWrapper.Mpi4pyCommWrapper method), 366
all() (fipy.variables.Variable method), 383
all() (fipy.variables.variable.Variable method), 421
all() (in module fipy.tools.numerix), 359, 360
allclose() (fipy.tools.comms.commWrapper.CommWrapper method), 365
allclose() (fipy.tools.comms.mpi4pyCommWrapper.Mpi4pyCommWrapper method), 366
allclose() (fipy.tools.dimensions.physicalField.PhysicalField method), 370
allclose() (fipy.tools.PhysicalField method), 346
allclose() (fipy.variables.Variable method), 384
allclose() (fipy.variables.variable.Variable method), 421
allclose() (in module fipy.tools.numerix), 358, 360
allequal() (fipy.tools.comms.commWrapper.CommWrapper method), 365
allequal() (fipy.tools.comms.mpi4pyCommWrapper.Mpi4pyCommWrapper method), 366
allequal() (fipy.tools.dimensions.physicalField.PhysicalField method), 370
allequal() (fipy.tools.PhysicalField method), 346
allequal() (fipy.variables.Variable method), 384
allequal() (fipy.variables.variable.Variable method), 421
allequal() (in module fipy.tools.numerix), 360
allgather() (fipy.tools.comms.commWrapper.CommWrapper method), 365
allgather() (fipy.tools.comms.mpi4pyCommWrapper.Mpi4pyCommWrapper method), 366
any() (fipy.tools.comms.commWrapper.CommWrapper method), 366
any() (fipy.tools.comms.mpi4pyCommWrapper.Mpi4pyCommWrapper method), 366
any() (fipy.variables.Variable method), 384
any() (fipy.variables.variable.Variable method), 422
appendChild() (fipy.tools.Vitals method), 354
appendChild() (fipy.tools.vitals.Vitals method), 365
appendInfo() (fipy.tools.Vitals method), 354
appendInfo() (fipy.tools.vitals.Vitals method), 365
arccos() (fipy.tools.dimensions.physicalField.PhysicalField method), 370
arccos() (fipy.tools.PhysicalField method), 346
arccos() (fipy.variables.Variable method), 384
arccos() (fipy.variables.variable.Variable method), 422
arccosh() (fipy.tools.dimensions.physicalField.PhysicalField method), 370
arccosh() (fipy.tools.PhysicalField method), 346

arccosh() (fipy.variables.Variable method), 384
arccosh() (fipy.variables.variable.Variable method), 422
arcsin() (fipy.tools.dimensions.physicalField.PhysicalField method), 371
arcsin() (fipy.tools.PhysicalField method), 346
arcsin() (fipy.variables.Variable method), 384
arcsin() (fipy.variables.variable.Variable method), 422
arcsinh() (fipy.variables.Variable method), 384
arcsinh() (fipy.variables.variable.Variable method), 422
arctan, 114
arctan() (fipy.tools.dimensions.physicalField.PhysicalField method), 371
arctan() (fipy.tools.PhysicalField method), 347
arctan() (fipy.variables.Variable method), 384
arctan() (fipy.variables.variable.Variable method), 422
arctan2, 114
arctan2() (fipy.tools.dimensions.physicalField.PhysicalField method), 371
arctan2() (fipy.tools.PhysicalField method), 347
arctan2() (fipy.variables.Variable method), 384
arctan2() (fipy.variables.variable.Variable method), 422
arctanh() (fipy.tools.dimensions.physicalField.PhysicalField method), 371
arctanh() (fipy.tools.PhysicalField method), 347
arctanh() (fipy.variables.Variable method), 384
arctanh() (fipy.variables.variable.Variable method), 422
arithmeticFaceValue (fipy.variables.CellVariable attribute), 390
arithmeticFaceValue (fipy.variables.cellVariable.CellVariable attribute), 405
arithmeticFaceValue (fipy.variables.ModularVariable attribute), 396
arithmeticFaceValue (fipy.variables.modularVariable.ModularVariable attribute), 418
array, 104
aspect2D (fipy.meshes.abstractMesh.AbstractMesh attribute), 206

B

Barrier() (fipy.tools.comms.commWrapper.CommWrapper method), 365
Barrier() (fipy.tools.comms.dummyComm.DummyComm method), 366
Base (class in package.subpackage.base), 187
bcast() (fipy.tools.comms.commWrapper.CommWrapper method), 366
bcast() (fipy.tools.comms.mpi4pyCommWrapper.Mpi4pyCommWrapper method), 366
BetaNoiseVariable (class in fipy.variables), 396
BetaNoiseVariable (class in fipy.variables.betaNoiseVariable), 403
BoundaryCondition (class in fipy.boundaryConditions.boundaryCondition), 192
buildAdvectionEquation() (in module fipy.models), 239
buildAdvectionEquation() (in module fipy.models.levelSet), 254
buildAdvectionEquation() (in module fipy.models.levelSet.advection), 257
buildAdvectionEquation() (in module fipy.models.levelSet.advection.advectionEquation), 258
Buildbot, **49**
buildHigherOrderAdvectionEquation, 153
buildHigherOrderAdvectionEquation() (in module fipy.models), 239
buildHigherOrderAdvectionEquation() (in module fipy.models.levelSet), 254
buildHigherOrderAdvectionEquation() (in module fipy.models.levelSet.advection), 257
buildHigherOrderAdvectionEquation() (in module fipy.models.levelSet.advection.higherOrderAdvectionEquation), 258
buildMetalIonDiffusionEquation, 153
buildMetalIonDiffusionEquation() (in module fipy.models), 239
buildMetalIonDiffusionEquation() (in module fipy.models.levelSet), 254
buildMetalIonDiffusionEquation() (in module fipy.models.levelSet.electroChem), 266
buildMetalIonDiffusionEquation() (in module fipy.models.levelSet.electroChem.metalIonDiffusionEquation), 270
buildSurfactantBulkDiffusionEquation, 153
buildSurfactantBulkDiffusionEquation() (in module fipy.models), 236
buildSurfactantBulkDiffusionEquation() (in module fipy.models.levelSet), 251
buildSurfactantBulkDiffusionEquation() (in module fipy.models.levelSet.surfactant), 278
buildSurfactantBulkDiffusionEquation() (in module fipy.models.levelSet.surfactant.surfactantBulkDiffusionEquation), 288

C

cacheMatrix, 165
cacheMatrix() (fipy.terms.term.Term method), 337
cacheMe() (fipy.variables.Variable method), 384
cacheMe() (fipy.variables.variable.Variable method), 422
cacheRHSvector, 165
cacheRHSvector() (fipy.terms.term.Term method), 337
calcDistanceFunction() (fipy.models.DistanceVariable method), 229
calcDistanceFunction() (fipy.models.levelSet.distanceFunction.DistanceVar method), 261
calcDistanceFunction() (fipy.models.levelSet.distanceFunction.distanceVari method), 264

calcDistanceFunction() (fipy.models.levelSet.DistanceVariable method), 244
ceil() (fipy.tools.dimensions.physicalField.PhysicalField method), 371
ceil() (fipy.tools.PhysicalField method), 347
ceil() (fipy.variables.Variable method), 384
ceil() (fipy.variables.variable.Variable method), 422
cellCenters (fipy.meshes.abstractMesh.AbstractMesh attribute), 206
cellCenters (fipy.meshes.cylindricalGrid2D.CylindricalGrid2D attribute), 210
cellCenters (fipy.meshes.PeriodicGrid1D attribute), 199
cellCenters (fipy.meshes.periodicGrid1D.PeriodicGrid1D attribute), 221
cellDistanceVectors (fipy.meshes.abstractMesh.AbstractMesh attribute), 206
cellFaceIDs (fipy.meshes.abstractMesh.AbstractMesh attribute), 206
cellInterfaceAreas (fipy.models.DistanceVariable attribute), 229
cellInterfaceAreas (fipy.models.levelSet.distanceFunction.DistanceVariable attribute), 261
cellInterfaceAreas (fipy.models.levelSet.distanceFunction.distanceVariable.DistanceVariable attribute), 264
cellInterfaceAreas (fipy.models.levelSet.DistanceVariable attribute), 244
CellTerm (class in fipy.terms.cellTerm), 329
cellToFaceDistanceVectors (fipy.meshes.abstractMesh.AbstractMesh attribute), 206
CellVariable, 81, 87, 92, 100, 108, 118, 121, 151, 164, 181, 183
CellVariable (class in fipy.variables), 389
CellVariable (class in fipy.variables.cellVariable), 405
cellVolumeAverage (fipy.variables.CellVariable attribute), 390
cellVolumeAverage (fipy.variables.cellVariable.CellVariable attribute), 406
cellVolumes (fipy.meshes.abstractMesh.AbstractMesh attribute), 206
cellVolumes (fipy.meshes.cylindricalGrid2D.CylindricalGrid2D attribute), 210
cellVolumes (fipy.meshes.cylindricalUniformGrid1D.CylindricalUniformGrid1D attribute), 211
cellVolumes (fipy.meshes.cylindricalUniformGrid2D.CylindricalUniformGrid2D attribute), 211
CentralDifferenceConvectionTerm (class in fipy.terms), 321
CentralDifferenceConvectionTerm (class in fipy.terms.centralDiffConvectionTerm), 329
clip_data() (fipy.viewers.mayaviViewer.mayaviDaemon.MayaviDaemon method), 458
command line option
–inline, 22
–no-pysparse, 22
–pyamg, 23
–pysparse, 22
–scipy, 22
–trilinos, 22
CommWrapper (class in fipy.tools.comms.commWrapper), 365
conjugate() (fipy.tools.dimensions.physicalField.PhysicalField method), 372
conjugate() (fipy.tools.PhysicalField method), 347
conjugate() (fipy.variables.Variable method), 384
conjugate() (fipy.variables.variable.Variable method), 422
constrain() (fipy.variables.CellVariable method), 390
constrain() (fipy.variables.cellVariable.CellVariable method), 406
constrain() (fipy.variables.Variable method), 384
constrain() (fipy.variables.variable.Variable method), 422
Constraint (class in fipy.boundaryConditions), 191
Constraint (class in fipy.boundaryConditions.constraint), 192
constraints (fipy.variables.Variable attribute), 385
constraints (fipy.variables.variable.Variable attribute), 423
ConvectionTerm (in module fipy.terms), 319
conversionFactorTo() (fipy.tools.dimensions.physicalField.PhysicalUnit method), 379
conversionTupleTo() (fipy.tools.dimensions.physicalField.PhysicalUnit method), 379
convertToUnit() (fipy.tools.dimensions.physicalField.PhysicalField method), 372
convertToUnit() (fipy.tools.PhysicalField method), 348
copy() (fipy.models.levelSet.surfactant.SurfactantVariable method), 273
copy() (fipy.models.levelSet.surfactant.surfactantVariable.SurfactantVariable method), 291
copy() (fipy.models.levelSet.SurfactantVariable method), 246
copy() (fipy.models.SurfactantVariable method), 231
copy() (fipy.terms.term.Term method), 337
copy() (fipy.tools.dimensions.physicalField.PhysicalField method), 372
copy() (fipy.tools.PhysicalField method), 348
copy() (fipy.variables.CellVariable method), 391
copy() (fipy.variables.cellVariable.CellVariable method), 407
copy() (fipy.variables.FaceVariable method), 395
copy() (fipy.variables.faceVariable.FaceVariable method), 413
copy() (fipy.variables.noiseVariable.NoiseVariable method), 419
copy() (fipy.variables.Variable method), 385
copy() (fipy.variables.variable.Variable method), 423
Copy_script (class in fipy.tools.copy_script), 355
cos() (fipy.tools.dimensions.physicalField.PhysicalField method), 372

cos() (fipy.tools.PhysicalField method), 348
cos() (fipy.variables.Variable method), 386
cos() (fipy.variables.variable.Variable method), 423
cosh() (fipy.tools.dimensions.physicalField.PhysicalField method), 372
cosh() (fipy.tools.PhysicalField method), 348
cosh() (fipy.variables.Variable method), 386
cosh() (fipy.variables.variable.Variable method), 423
CylindricalGrid1D (class in fipy.meshes.cylindricalGrid1D), 209
CylindricalGrid1D() (in module fipy.meshes), 198
CylindricalGrid1D() (in module fipy.meshes.factoryMeshes), 213
CylindricalGrid2D (class in fipy.meshes.cylindricalGrid2D), 210
CylindricalGrid2D() (in module fipy.meshes), 198
CylindricalGrid2D() (in module fipy.meshes.factoryMeshes), 212
CylindricalUniformGrid1D (class in fipy.meshes.cylindricalUniformGrid1D), 210
CylindricalUniformGrid2D (class in fipy.meshes.cylindricalUniformGrid2D), 211

D

DefaultAsymmetricSolver (in module fipy.solvers), 293
DefaultAsymmetricSolver (in module fipy.solvers.pyAMG), 296
DefaultAsymmetricSolver (in module fipy.solvers.pysparse), 300
DefaultAsymmetricSolver (in module fipy.solvers.scipy), 304
DefaultAsymmetricSolver (in module fipy.solvers.trilinos), 307
DefaultSolver (in module fipy.solvers), 293
DefaultSolver (in module fipy.solvers.pyAMG), 296
DefaultSolver (in module fipy.solvers.pysparse), 299
DefaultSolver (in module fipy.solvers.scipy), 304
DefaultSolver (in module fipy.solvers.trilinos), 307
description (fipy.tools.copy_script.Copy_script attribute), 355
description (fipy.tools.performance.efficiency_test.Efficiency_test attribute), 381
dictToXML() (fipy.tools.Vitals method), 354
dictToXML() (fipy.tools.vitals.Vitals method), 365
DiffusionTerm (class in fipy.terms), 320
DiffusionTerm (class in fipy.terms.diffusionTerm), 330
DiffusionTermCorrection (class in fipy.terms), 320
DiffusionTermCorrection (class in fipy.terms.diffusionTerm), 331
DiffusionTermCorrection (class in fipy.terms.diffusionTermCorrection), 331
DiffusionTermNoCorrection (class in fipy.terms), 320, 321
DiffusionTermNoCorrection (class in fipy.terms.diffusionTerm), 331
DiffusionTermNoCorrection (class in fipy.terms.diffusionTermNoCorrection), 331
DistanceVariable, 151
DistanceVariable (class in fipy.models), 227
DistanceVariable (class in fipy.models.levelSet), 242
DistanceVariable (class in fipy.models.levelSet.distanceFunction), 258
DistanceVariable (class in fipy.models.levelSet.distanceFunction.distanceVariable), 262
divergence (fipy.variables.FaceVariable attribute), 395
divergence (fipy.variables.faceVariable.FaceVariable attribute), 413
divide() (fipy.tools.dimensions.physicalField.PhysicalField method), 373
divide() (fipy.tools.PhysicalField method), 349
DomDecompPreconditioner (class in fipy.solvers.trilinos), 309
DomDecompPreconditioner (class in fipy.solvers.trilinos.preconditioners), 314
DomDecompPreconditioner (class in fipy.solvers.trilinos.preconditioners.domDecompPreconditioner), 315
dontCacheMe() (fipy.variables.Variable method), 386
dontCacheMe() (fipy.variables.variable.Variable method), 423
dot() (fipy.tools.dimensions.physicalField.PhysicalField method), 373
dot() (fipy.tools.PhysicalField method), 349
dot() (fipy.variables.Variable method), 386
dot() (fipy.variables.variable.Variable method), 423
dot() (in module fipy.tools.numerix), 357, 360
DummyComm (class in fipy.tools.comms.dummyComm), 366
DummySolver (in module fipy.solvers), 293
DummySolver (in module fipy.solvers.pyAMG), 296
DummySolver (in module fipy.solvers.pysparse), 299
DummySolver (in module fipy.solvers.scipy), 304
DummySolver (in module fipy.solvers.trilinos), 307
DummyViewer (class in fipy.viewers), 438
DYLD_LIBRARY_PATH, 17

E

Efficiency_test (class in fipy.tools.performance.efficiency_test), 381
electrolyteMask (fipy.models.levelSet.electroChem.gapFillMesh.TrenchMes attribute), 270
electrolyteMask (fipy.models.levelSet.electroChem.TrenchMesh attribute), 268

electrolyteMask (fipy.models.levelSet.TrenchMesh attribute), 257
electrolyteMask (fipy.models.TrenchMesh attribute), 242
environment variable
DYLD_LIBRARY_PATH, 17
FIPY_DISPLAY_MATRIX, 23
FIPY_INCLUDE_NUMERIX_ALL, 23, 175
FIPY_INLINE, 23
FIPY_INLINE_COMMENT, 23
FIPY_SOLVERS, 15, 22, 23
FIPY_VERBOSE_SOLVER, 23
FIPY_VIEWER, 23
LD_LIBRARY_PATH, 17
PYTHONPATH, 13, 17
examples.cahnHilliard.mesh2DCoupled (module), 157
examples.cahnHilliard.sphere (module), 160
examples.convection.exponential1D.mesh1D (module), 85
examples.convection.exponential1DSource.mesh1D (module), 86
examples.convection.robin (module), 87
examples.convection.source (module), 89
examples.diffusion.anisotropy (module), 82
examples.diffusion.circle (module), 71
examples.diffusion.coupled (module), 67
examples.diffusion.electrostatics (module), 76
examples.diffusion.mesh1D (module), 55
examples.diffusion.mesh20x20 (module), 69
examples.diffusion.nthOrder.input4thOrder1D (module), 80
examples.flow.stokesCavity (module), 163
examples.levelSet.advection.circle (module), 138
examples.levelSet.advection.mesh1D (module), 137
examples.levelSet.distanceFunction.circle (module), 136
examples.levelSet.distanceFunction.mesh1D (module), 135
examples.levelSet.electroChem.gold (module), 144
examples.levelSet.electroChem.howToWriteAScript (module), 149
examples.levelSet.electroChem.leveler (module), 145
examples.levelSet.electroChem.simpleTrenchSystem (module), 141
examples.phase.anisotropy (module), 113
examples.phase.binaryCoupled (module), 99
examples.phase.impingement.mesh20x20 (module), 120
examples.phase.impingement.mesh40x1 (module), 117
examples.phase.polyxtal (module), 123
examples.phase.polyxtalCoupled (module), 129
examples.phase.quaternary (module), 107
examples.phase.simple (module), 91
examples.reactiveWetting.liquidVapor1D (module), 169
examples.updating.update0_1to1_0 (module), 180
examples.updating.update1_0to2_0 (module), 176
examples.updating.update2_0to3_0 (module), 175
execButNoTest() (in module fipy.tests.doctestPlus), 343
exp, 86, 87, 104, 119, 122, 150
exp() (fipy.variables.Variable method), 386
exp() (fipy.variables.variable.Variable method), 423
ExplicitDiffusionTerm, 56, 119, 121
ExplicitDiffusionTerm (class in fipy.terms), 321
ExplicitDiffusionTerm (class in fipy.terms.explicitDiffusionTerm), 331
ExplicitUpwindConvectionTerm (class in fipy.terms), 322
ExplicitUpwindConvectionTerm (class in fipy.terms.explicitUpwindConvectionTerm), 331
ExplicitVariableError, 319
ExponentialConvectionTerm, 181
ExponentialConvectionTerm (class in fipy.terms), 323
ExponentialConvectionTerm (class in fipy.terms.exponentialConvectionTerm), 332
ExponentialNoiseVariable (class in fipy.variables), 398
ExponentialNoiseVariable (class in fipy.variables.exponentialNoiseVariable), 411
extendVariable() (fipy.models.DistanceVariable method), 230
extendVariable() (fipy.models.levelSet.distanceFunction.DistanceVariable method), 262
extendVariable() (fipy.models.levelSet.distanceFunction.distanceVariable.D method), 265
extendVariable() (fipy.models.levelSet.DistanceVariable method), 245
exteriorFaces (fipy.meshes.abstractMesh.AbstractMesh attribute), 206
exteriorFaces (fipy.meshes.uniformGrid1D.UniformGrid1D attribute), 224
extrude() (fipy.meshes.mesh2D.Mesh2D method), 220

F

faceCellIDs (fipy.meshes.uniformGrid1D.UniformGrid1D attribute), 224
faceCellIDs (fipy.meshes.uniformGrid2D.UniformGrid2D attribute), 224
faceCellIDs (fipy.meshes.uniformGrid3D.UniformGrid3D attribute), 224
faceCenters (fipy.meshes.abstractMesh.AbstractMesh attribute), 206
faceCenters (fipy.meshes.cylindricalGrid2D.CylindricalGrid2D attribute), 210
faceGrad (fipy.variables.CellVariable attribute), 391
faceGrad (fipy.variables.cellVariable.CellVariable attribute), 407
faceGrad (fipy.variables.ModularVariable attribute), 396
faceGrad (fipy.variables.modularVariable.ModularVariable attribute), 418

faceGradAverage (fipy.variables.CellVariable attribute), 391
faceGradAverage (fipy.variables.cellVariable.CellVariable attribute), 407
faceGradNoMod (fipy.variables.ModularVariable attribute), 396
faceGradNoMod (fipy.variables.modularVariable.ModularVariable attribute), 418
facesBack (fipy.meshes.abstractMesh.AbstractMesh attribute), 206
facesBottom (fipy.meshes.abstractMesh.AbstractMesh attribute), 206
facesDown (fipy.meshes.abstractMesh.AbstractMesh attribute), 206
facesFront (fipy.meshes.abstractMesh.AbstractMesh attribute), 206
facesLeft (fipy.meshes.abstractMesh.AbstractMesh attribute), 207
facesRight (fipy.meshes.abstractMesh.AbstractMesh attribute), 207
facesTop (fipy.meshes.abstractMesh.AbstractMesh attribute), 207
facesUp (fipy.meshes.abstractMesh.AbstractMesh attribute), 207
FaceTerm (class in fipy.terms.faceTerm), 334
faceValue (fipy.variables.CellVariable attribute), 391
faceValue (fipy.variables.cellVariable.CellVariable attribute), 407
FaceVariable, 62
FaceVariable (class in fipy.variables), 395
FaceVariable (class in fipy.variables.faceVariable), 412
faceVertexIDs (fipy.meshes.uniformGrid2D.UniformGrid2D attribute), 224
faceVertexIDs (fipy.meshes.uniformGrid3D.UniformGrid3D attribute), 225
failFn() (fipy.steppers.stepper.Stepper static method), 318
figaspect() (fipy.viewers.matplotlibViewer.matplotlibViewer.AbstractMatplotlibViewer method), 455
finalize_options() (fipy.tools.copy_script.Copy_script method), 355
finalize_options() (fipy.tools.performance.efficiency_test.Efficiency_test method), 381
FiPy, **49**
fipy.boundaryConditions (module), 191
fipy.boundaryConditions.boundaryCondition (module), 192
fipy.boundaryConditions.constraint (module), 192
fipy.boundaryConditions.fixedFlux (module), 192
fipy.boundaryConditions.fixedValue (module), 192
fipy.boundaryConditions.nthOrderBoundaryCondition (module), 193
fipy.boundaryConditions.test (module), 193
fipy.matrices.offsetSparseMatrix (module), 195
fipy.matrices.pysparseMatrix (module), 195
fipy.matrices.scipyMatrix (module), 195
fipy.matrices.sparseMatrix (module), 195
fipy.matrices.test (module), 195
fipy.matrices.trilinosMatrix (module), 195
fipy.meshes (module), 197
fipy.meshes.abstractMesh (module), 205
fipy.meshes.builders (module), 225
fipy.meshes.builders.abstractGridBuilder (module), 225
fipy.meshes.builders.grid1DBuilder (module), 225
fipy.meshes.builders.grid2DBuilder (module), 225
fipy.meshes.builders.grid3DBuilder (module), 225
fipy.meshes.builders.periodicGrid1DBuilder (module), 225
fipy.meshes.builders.utilityClasses (module), 225
fipy.meshes.cylindricalGrid1D (module), 209
fipy.meshes.cylindricalGrid2D (module), 210
fipy.meshes.cylindricalUniformGrid1D (module), 210
fipy.meshes.cylindricalUniformGrid2D (module), 211
fipy.meshes.factoryMeshes (module), 211
fipy.meshes.gmshMesh (module), 213
fipy.meshes.grid1D (module), 218
fipy.meshes.grid2D (module), 218
fipy.meshes.grid3D (module), 219
fipy.meshes.mesh (module), 219
fipy.meshes.mesh1D (module), 219
fipy.meshes.mesh2D (module), 220
fipy.meshes.numMesh.cylindricalGrid1D (module), 225
fipy.meshes.numMesh.cylindricalGrid2D (module), 225
fipy.meshes.numMesh.cylindricalUniformGrid1D (module), 225
fipy.meshes.numMesh.cylindricalUniformGrid2D (module), 225
fipy.meshes.numMesh.deprecatedWarning (module), 225
fipy.meshes.numMesh.gmshImport (module), 226
fipy.meshes.numMesh.grid1D (module), 226
fipy.meshes.numMesh.grid2D (module), 226
fipy.meshes.numMesh.grid3D (module), 226
fipy.meshes.numMesh.periodicGrid1D (module), 226
fipy.meshes.numMesh.periodicGrid2D (module), 226
fipy.meshes.numMesh.skewedGrid2D (module), 226
fipy.meshes.numMesh.tri2D (module), 226
fipy.meshes.numMesh.uniformGrid1D (module), 226
fipy.meshes.numMesh.uniformGrid2D (module), 226
fipy.meshes.numMesh.uniformGrid3D (module), 226
fipy.meshes.periodicGrid1D (module), 220
fipy.meshes.periodicGrid2D (module), 221
fipy.meshes.representations.abstractRepresentation (module), 226
fipy.meshes.representations.gridRepresentation (module), 226
fipy.meshes.representations.meshRepresentation (module), 226
fipy.meshes.skewedGrid2D (module), 222
fipy.meshes.test (module), 222

fipy.meshes.topologies.abstractTopology (module), 226
fipy.meshes.topologies.gridTopology (module), 226
fipy.meshes.topologies.meshTopology (module), 226
fipy.meshes.tri2D (module), 223
fipy.meshes.uniformGrid (module), 223
fipy.meshes.uniformGrid1D (module), 223
fipy.meshes.uniformGrid2D (module), 224
fipy.meshes.uniformGrid3D (module), 224
fipy.models (module), 227
fipy.models.levelSet (module), 242
fipy.models.levelSet.advection (module), 257
fipy.models.levelSet.advection.advectionEquation (module), 258
fipy.models.levelSet.advection.advectionTerm (module), 258
fipy.models.levelSet.advection.higherOrderAdvectionEquation (module), 258
fipy.models.levelSet.advection.higherOrderAdvectionTerm (module), 258
fipy.models.levelSet.distanceFunction (module), 258
fipy.models.levelSet.distanceFunction.distanceVariable (module), 262
fipy.models.levelSet.distanceFunction.levelSetDiffusionEquation (module), 265
fipy.models.levelSet.distanceFunction.levelSetDiffusionVariable (module), 266
fipy.models.levelSet.electroChem (module), 266
fipy.models.levelSet.electroChem.gapFillMesh (module), 268
fipy.models.levelSet.electroChem.metalIonDiffusionEquation (module), 270
fipy.models.levelSet.electroChem.metalIonSourceVariable (module), 272
fipy.models.levelSet.electroChem.test (module), 272
fipy.models.levelSet.surfactant (module), 272
fipy.models.levelSet.surfactant.adsorbingSurfactantEquation (module), 281
fipy.models.levelSet.surfactant.convectionCoeff (module), 285
fipy.models.levelSet.surfactant.lines (module), 285
fipy.models.levelSet.surfactant.matplotlibSurfactantViewer (module), 286
fipy.models.levelSet.surfactant.mayaviSurfactantViewer (module), 287
fipy.models.levelSet.surfactant.surfactantBulkDiffusionEquation (module), 288
fipy.models.levelSet.surfactant.surfactantEquation (module), 290
fipy.models.levelSet.surfactant.surfactantVariable (module), 290
fipy.models.levelSet.test (module), 257
fipy.models.test (module), 242
fipy.solvers (module), 293
fipy.solvers.pyAMG (module), 296
fipy.solvers.pyAMG.linearCGSSolver (module), 298
fipy.solvers.pyAMG.linearGeneralSolver (module), 298
fipy.solvers.pyAMG.linearGMRESSolver (module), 298
fipy.solvers.pyAMG.linearLUSolver (module), 299
fipy.solvers.pyAMG.linearPCGSolver (module), 299
fipy.solvers.pyAMG.preconditioners (module), 299
fipy.solvers.pyAMG.preconditioners.smoothedAggregationPreconditioner (module), 299
fipy.solvers.pysparse (module), 299
fipy.solvers.pysparse.linearCGSSolver (module), 301
fipy.solvers.pysparse.linearGMRESSolver (module), 301
fipy.solvers.pysparse.linearJORSolver (module), 302
fipy.solvers.pysparse.linearLUSolver (module), 302
fipy.solvers.pysparse.linearPCGSolver (module), 302
fipy.solvers.pysparse.preconditioners (module), 303
fipy.solvers.pysparse.preconditioners.jacobiPreconditioner (module), 303
fipy.solvers.pysparse.preconditioners.preconditioner (module), 303
fipy.solvers.pysparse.preconditioners.ssorPreconditioner (module), 304
fipy.solvers.pysparse.pysparseSolver (module), 303
fipy.solvers.pysparseMatrixSolver (module), 295
fipy.solvers.scipy (module), 304
fipy.solvers.scipy.linearBicgstabSolver (module), 305
fipy.solvers.scipy.linearCGSSolver (module), 305
fipy.solvers.scipy.linearGMRESSolver (module), 306
fipy.solvers.scipy.linearLUSolver (module), 306
fipy.solvers.scipy.linearPCGSolver (module), 306
fipy.solvers.scipy.scipyKrylovSolver (module), 307
fipy.solvers.scipy.scipySolver (module), 307
fipy.solvers.solver (module), 295
fipy.solvers.test (module), 296
fipy.solvers.trilinos (module), 307
fipy.solvers.trilinos.linearBicgstabSolver (module), 310
fipy.solvers.trilinos.linearCGSSolver (module), 310
fipy.solvers.trilinos.linearGMRESSolver (module), 310
fipy.solvers.trilinos.linearLUSolver (module), 311
fipy.solvers.trilinos.linearPCGSolver (module), 311
fipy.solvers.trilinos.preconditioners (module), 313
fipy.solvers.trilinos.preconditioners.domDecompPreconditioner (module), 315
fipy.solvers.trilinos.preconditioners.icPreconditioner (module), 315
fipy.solvers.trilinos.preconditioners.jacobiPreconditioner (module), 315
fipy.solvers.trilinos.preconditioners.multilevelDDMLPreconditioner (module), 315
fipy.solvers.trilinos.preconditioners.multilevelDDPreconditioner (module), 315
fipy.solvers.trilinos.preconditioners.multilevelNSSAPreconditioner (module), 315
fipy.solvers.trilinos.preconditioners.multilevelSAPreconditioner (module), 315

fipy.solvers.trilinos.preconditioners.multilevelSGSPreconditioner (module), 316
fipy.solvers.trilinos.preconditioners.multilevelSolverSmootherPreconditioner (module), 316
fipy.solvers.trilinos.preconditioners.preconditioner (module), 316
fipy.solvers.trilinos.trilinosAztecOOSolver (module), 311
fipy.solvers.trilinos.trilinosMLTest (module), 312
fipy.solvers.trilinos.trilinosNonlinearSolver (module), 313
fipy.solvers.trilinos.trilinosSolver (module), 313
fipy.steppers (module), 317
fipy.steppers.pidStepper (module), 318
fipy.steppers.pseudoRKQSStepper (module), 318
fipy.steppers.stepper (module), 318
fipy.terms (module), 319
fipy.terms.abstractBinaryTerm (module), 329
fipy.terms.abstractConvectionTerm (module), 329
fipy.terms.abstractDiffusionTerm (module), 329
fipy.terms.abstractUpwindConvectionTerm (module), 329
fipy.terms.asymmetricConvectionTerm (module), 329
fipy.terms.binaryTerm (module), 329
fipy.terms.cellTerm (module), 329
fipy.terms.centralDiffConvectionTerm (module), 329
fipy.terms.coupledBinaryTerm (module), 330
fipy.terms.diffusionTerm (module), 330
fipy.terms.diffusionTermCorrection (module), 331
fipy.terms.diffusionTermNoCorrection (module), 331
fipy.terms.explicitDiffusionTerm (module), 331
fipy.terms.explicitSourceTerm (module), 331
fipy.terms.explicitUpwindConvectionTerm (module), 331
fipy.terms.exponentialConvectionTerm (module), 332
fipy.terms.faceTerm (module), 334
fipy.terms.hybridConvectionTerm (module), 334
fipy.terms.implicitDiffusionTerm (module), 335
fipy.terms.implicitSourceTerm (module), 335
fipy.terms.nonDiffusionTerm (module), 335
fipy.terms.powerLawConvectionTerm (module), 335
fipy.terms.residualTerm (module), 336
fipy.terms.sourceTerm (module), 337
fipy.terms.term (module), 337
fipy.terms.test (module), 339
fipy.terms.transientTerm (module), 339
fipy.terms.unaryTerm (module), 340
fipy.terms.upwindConvectionTerm (module), 340
fipy.terms.vanLeerConvectionTerm (module), 342
fipy.tests (module), 343
fipy.tests.doctestPlus (module), 343
fipy.tests.lateImportTest (module), 344
fipy.tests.testBase (module), 344
fipy.tests.testClass (module), 344
fipy.tests.testProgram (module), 344
fipy.tools (module), 345
fipy.tools.comms.commWrapper (module), 365
fipy.tools.comms.dummyComm (module), 366
fipy.tools.comms.mpi4pyCommWrapper (module), 366
fipy.tools.comms.serialCommWrapper (module), 366
fipy.tools.copy_script (module), 355
fipy.tools.debug (module), 355
fipy.tools.decorators (module), 355
fipy.tools.dimensions.DictWithDefault (module), 367
fipy.tools.dimensions.NumberDict (module), 367
fipy.tools.dimensions.physicalField (module), 367
fipy.tools.dump
    module, 123
fipy.tools.dump (module), 356
fipy.tools.inline (module), 357
fipy.tools.numerix (module), 357
fipy.tools.parser
    module, 120, 150
fipy.tools.parser (module), 364
fipy.tools.performance.efficiency_test (module), 381
fipy.tools.performance.efficiencyTestGenerator (module), 381
fipy.tools.performance.efficiencyTestHistory (module), 381
fipy.tools.performance.memoryLeak (module), 381
fipy.tools.performance.memoryLogger (module), 381
fipy.tools.performance.memoryUsage (module), 381
fipy.tools.test (module), 364
fipy.tools.vector (module), 364
fipy.tools.vitals (module), 365
fipy.variables (module), 383
fipy.variables.addOverFacesVariable (module), 403
fipy.variables.arithmeticCellToFaceVariable (module), 403
fipy.variables.betaNoiseVariable (module), 403
fipy.variables.binaryOperatorVariable (module), 405
fipy.variables.cellToFaceVariable (module), 405
fipy.variables.cellVariable (module), 405
fipy.variables.cellVariable.CellVariable
    object, 72
fipy.variables.cellVolumeAverageVariable (module), 411
fipy.variables.constant (module), 411
fipy.variables.constraintMask (module), 411
fipy.variables.coupledCellVariable (module), 411
fipy.variables.exponentialNoiseVariable (module), 411
fipy.variables.faceGradContributionsVariable (module), 412
fipy.variables.faceGradVariable (module), 412
fipy.variables.faceVariable (module), 412
fipy.variables.gammaNoiseVariable (module), 413
fipy.variables.gaussCellGradVariable (module), 415
fipy.variables.gaussianNoiseVariable (module), 415
fipy.variables.harmonicCellToFaceVariable (module), 417
fipy.variables.histogramVariable (module), 417

fipy.variables.leastSquaresCellGradVariable (module), 417
fipy.variables.meshVariable (module), 417
fipy.variables.minmodCellToFaceVariable (module), 417
fipy.variables.modCellGradVariable (module), 417
fipy.variables.modCellToFaceVariable (module), 417
fipy.variables.modFaceGradVariable (module), 417
fipy.variables.modPhysicalField (module), 417
fipy.variables.modularVariable (module), 417
fipy.variables.noiseVariable (module), 418
fipy.variables.operatorVariable (module), 419
fipy.variables.scharfetterGummelFaceVariable (module), 419
fipy.variables.test (module), 419
fipy.variables.unaryOperatorVariable (module), 419
fipy.variables.uniformNoiseVariable (module), 419
fipy.variables.variable (module), 420
fipy.viewers, 184
    module, 56, 70, 81, 86, 87, 92, 105, 119, 122, 165
fipy.viewers (module), 429
fipy.viewers.gistViewer (module), 442
fipy.viewers.gistViewer.gist1DViewer (module), 442
fipy.viewers.gistViewer.gist2DViewer (module), 442
fipy.viewers.gistViewer.gistVectorViewer (module), 442
fipy.viewers.gistViewer.gistViewer (module), 442
fipy.viewers.gistViewer.test (module), 442
fipy.viewers.gnuplotViewer (module), 442
fipy.viewers.gnuplotViewer.gnuplot1DViewer (module), 443
fipy.viewers.gnuplotViewer.gnuplot2DViewer (module), 443
fipy.viewers.gnuplotViewer.gnuplotViewer (module), 443
fipy.viewers.gnuplotViewer.test (module), 443
fipy.viewers.matplotlibViewer (module), 443
fipy.viewers.matplotlibViewer.matplotlib1DViewer (module), 449
fipy.viewers.matplotlibViewer.matplotlib2DGridContourViewer (module), 450
fipy.viewers.matplotlibViewer.matplotlib2DGridViewer (module), 451
fipy.viewers.matplotlibViewer.matplotlib2DViewer (module), 452
fipy.viewers.matplotlibViewer.matplotlibSparseMatrixViewer (module), 453
fipy.viewers.matplotlibViewer.matplotlibVectorViewer (module), 453
fipy.viewers.matplotlibViewer.matplotlibViewer (module), 454
fipy.viewers.matplotlibViewer.test (module), 455
fipy.viewers.mayaviViewer (module), 455
fipy.viewers.mayaviViewer.mayaviClient (module), 457
fipy.viewers.mayaviViewer.mayaviDaemon (module), 458
fipy.viewers.mayaviViewer.test (module), 459
fipy.viewers.multiViewer (module), 439
fipy.viewers.test (module), 439
fipy.viewers.testinteractive (module), 439
fipy.viewers.tsvViewer (module), 440
fipy.viewers.tsvViewer.TSVViewer object, 74
fipy.viewers.viewer (module), 441
fipy.viewers.vtkViewer (module), 459
fipy.viewers.vtkViewer.test (module), 460
fipy.viewers.vtkViewer.vtkCellViewer (module), 460
fipy.viewers.vtkViewer.vtkFaceViewer (module), 460
fipy.viewers.vtkViewer.vtkViewer (module), 461
FIPY_INCLUDE_NUMERIX_ALL, 175
FIPY_SOLVERS, 15, 22
FixedFlux, 181
FixedFlux (class in fipy.boundaryConditions), 191
FixedFlux (class in fipy.boundaryConditions.fixedFlux), 192
FixedValue, 181, 183
FixedValue (class in fipy.boundaryConditions), 191
FixedValue (class in fipy.boundaryConditions.fixedValue), 192
floor() (fipy.tools.dimensions.physicalField.PhysicalField method), 373
floor() (fipy.tools.PhysicalField method), 349
floor() (fipy.variables.Variable method), 386
floor() (fipy.variables.variable.Variable method), 423

G

GammaNoiseVariable (class in fipy.variables), 399
GammaNoiseVariable (class in fipy.variables.gammaNoiseVariable), 413
gaussGrad (fipy.variables.CellVariable attribute), 392
gaussGrad (fipy.variables.cellVariable.CellVariable attribute), 408
GaussianNoiseVariable (class in fipy.variables), 400
GaussianNoiseVariable (class in fipy.variables.gaussianNoiseVariable), 415
GeneralSolver (in module fipy.solvers), 294
GeneralSolver (in module fipy.solvers.pyAMG), 296
GeneralSolver (in module fipy.solvers.pysparse), 300
GeneralSolver (in module fipy.solvers.scipy), 304
GeneralSolver (in module fipy.solvers.trilinos), 307
getArithmeticFaceValue() (fipy.variables.CellVariable method), 392
getArithmeticFaceValue() (fipy.variables.cellVariable.CellVariable method), 408
getCellCenters() (fipy.meshes.abstractMesh.AbstractMesh method), 207
getCellInterfaceAreas() (fipy.models.DistanceVariable method), 230
getCellInterfaceAreas() (fipy.models.levelSet.distanceFunction.DistanceVar method), 262

getCellInterfaceAreas() (fipy.models.levelSet.distanceFunction.distanceVariable.DistanceVariable method), 265
getCellInterfaceAreas() (fipy.models.levelSet.DistanceVariable method), 245
getCellVolumeAverage() (fipy.variables.CellVariable method), 392
getCellVolumeAverage() (fipy.variables.cellVariable.CellVariable method), 408
getCellVolumes() (fipy.meshes.abstractMesh.AbstractMesh method), 208
getDefaultSolver() (fipy.terms.term.Term method), 337
getDim() (fipy.meshes.abstractMesh.AbstractMesh method), 208
getDivergence() (fipy.variables.FaceVariable method), 395
getDivergence() (fipy.variables.faceVariable.FaceVariable method), 413
getElectrolyteMask() (fipy.models.levelSet.electroChem.gapFillMesh.TrenchMesh method), 270
getElectrolyteMask() (fipy.models.levelSet.electroChem.TrenchMesh method), 268
getElectrolyteMask() (fipy.models.levelSet.TrenchMesh method), 257
getElectrolyteMask() (fipy.models.TrenchMesh method), 242
getExteriorFaces() (fipy.meshes.abstractMesh.AbstractMesh method), 208
getFaceCellIDs() (fipy.meshes.abstractMesh.AbstractMesh method), 208
getFaceCenters() (fipy.meshes.abstractMesh.AbstractMesh method), 208
getFaceGrad() (fipy.variables.CellVariable method), 392
getFaceGrad() (fipy.variables.cellVariable.CellVariable method), 408
getFaceGradAverage() (fipy.variables.CellVariable method), 392
getFaceGradAverage() (fipy.variables.cellVariable.CellVariable method), 408
getFaceGradNoMod() (fipy.variables.ModularVariable method), 396
getFaceGradNoMod() (fipy.variables.modularVariable.ModularVariable method), 418
getFacesBack() (fipy.meshes.abstractMesh.AbstractMesh method), 208
getFacesBottom() (fipy.meshes.abstractMesh.AbstractMesh method), 208
getFacesDown() (fipy.meshes.abstractMesh.AbstractMesh method), 208
getFacesFront() (fipy.meshes.abstractMesh.AbstractMesh method), 208
getFacesLeft() (fipy.meshes.abstractMesh.AbstractMesh method), 208
getFacesRight() (fipy.meshes.abstractMesh.AbstractMesh method), 208
getFacesTop() (fipy.meshes.abstractMesh.AbstractMesh method), 208
getFacesUp() (fipy.meshes.abstractMesh.AbstractMesh method), 208
getFaceValue() (fipy.variables.CellVariable method), 392
getFaceValue() (fipy.variables.cellVariable.CellVariable method), 408
getGaussGrad() (fipy.variables.CellVariable method), 392
getGaussGrad() (fipy.variables.cellVariable.CellVariable method), 408
getGrad() (fipy.variables.CellVariable method), 392
getGrad() (fipy.variables.cellVariable.CellVariable method), 408
getHarmonicFaceValue() (fipy.variables.CellVariable method), 392
getHarmonicFaceValue() (fipy.variables.cellVariable.CellVariable method), 408
getInterfaceVar() (fipy.models.levelSet.surfactant.SurfactantVariable method), 273
getInterfaceVar() (fipy.models.levelSet.surfactant.surfactantVariable.Surfact method), 291
getInterfaceVar() (fipy.models.levelSet.SurfactantVariable method), 246
getInterfaceVar() (fipy.models.SurfactantVariable method), 231
getInteriorFaceCellIDs() (fipy.meshes.abstractMesh.AbstractMesh method), 208
getInteriorFaceIDs() (fipy.meshes.abstractMesh.AbstractMesh method), 208
getInteriorFaces() (fipy.meshes.abstractMesh.AbstractMesh method), 208
getLeastSquaresGrad() (fipy.variables.CellVariable method), 392
getLeastSquaresGrad() (fipy.variables.cellVariable.CellVariable method), 408
getMag() (fipy.variables.Variable method), 386
getMag() (fipy.variables.variable.Variable method), 423
getMatrix, 165
getMatrix() (fipy.terms.term.Term method), 337
getMinmodFaceValue() (fipy.variables.CellVariable method), 392
getMinmodFaceValue() (fipy.variables.cellVariable.CellVariable method), 408
getName() (fipy.variables.Variable method), 386
getName() (fipy.variables.variable.Variable method), 424
getNearestCell() (fipy.meshes.abstractMesh.AbstractMesh method), 208
getNumberOfCells() (fipy.meshes.abstractMesh.AbstractMesh method), 208
getNumericValue() (fipy.tools.dimensions.physicalField.PhysicalField method), 373
getNumericValue() (fipy.tools.PhysicalField method), 349

getNumericValue() (fipy.variables.Variable method), 386
getNumericValue() (fipy.variables.variable.Variable method), 424
getOld() (fipy.variables.CellVariable method), 392
getOld() (fipy.variables.cellVariable.CellVariable method), 408
getPhysicalShape() (fipy.meshes.abstractMesh.AbstractMesh method), 208
getRHSvector, 165
getRHSvector() (fipy.terms.term.Term method), 337
getScale() (fipy.meshes.abstractMesh.AbstractMesh method), 208
getsctype() (fipy.tools.dimensions.physicalField.PhysicalField method), 373
getsctype() (fipy.tools.PhysicalField method), 349
getsctype() (fipy.variables.Variable method), 386
getsctype() (fipy.variables.variable.Variable method), 424
getsetDeprecated() (in module fipy.tools.decorators), 355
getShape() (fipy.meshes.abstractMesh.AbstractMesh method), 208
getShape() (fipy.tools.dimensions.physicalField.PhysicalField method), 373
getShape() (fipy.tools.PhysicalField method), 349
getShape() (fipy.variables.Variable method), 386
getShape() (fipy.variables.variable.Variable method), 424
getShape() (in module fipy.tools.numerix), 361
getSubscribedVariables() (fipy.variables.Variable method), 386
getSubscribedVariables() (fipy.variables.variable.Variable method), 424
getUnit() (fipy.tools.dimensions.physicalField.PhysicalField method), 373
getUnit() (fipy.tools.PhysicalField method), 349
getUnit() (fipy.variables.Variable method), 386
getUnit() (fipy.variables.variable.Variable method), 424
getUnit() (in module fipy.tools.numerix), 361
getValue() (fipy.variables.Variable method), 386
getValue() (fipy.variables.variable.Variable method), 424
getVars() (fipy.viewers.viewer.AbstractViewer method), 441
getVertexCoords() (fipy.meshes.abstractMesh.AbstractMesh method), 208
getViewers() (fipy.viewers.MultiViewer method), 436
getViewers() (fipy.viewers.multiViewer.MultiViewer method), 439
Gist1DViewer, 184
GistViewer() (in module fipy.viewers), 429
GistViewer() (in module fipy.viewers.gistViewer), 442
globalValue (fipy.variables.CellVariable attribute), 392
globalValue (fipy.variables.cellVariable.CellVariable attribute), 408
globalValue (fipy.variables.FaceVariable attribute), 395
globalValue (fipy.variables.faceVariable.FaceVariable attribute), 413
Gmsh, **49**
gmsh, 142, 144, 146
Gmsh2D (class in fipy.meshes), 201
Gmsh2D (class in fipy.meshes.gmshMesh), 214
Gmsh2DIn3DSpace (class in fipy.meshes), 205
Gmsh2DIn3DSpace (class in fipy.meshes.gmshMesh), 217
Gmsh3D (class in fipy.meshes), 205
Gmsh3D (class in fipy.meshes.gmshMesh), 217
GmshGrid2D (class in fipy.meshes), 205
GmshGrid2D (class in fipy.meshes.gmshMesh), 217
GmshGrid3D (class in fipy.meshes), 205
GmshGrid3D (class in fipy.meshes.gmshMesh), 217
GmshImporter2D (class in fipy.meshes), 205
GmshImporter2D (class in fipy.meshes.gmshMesh), 218
GmshImporter2DIn3DSpace (class in fipy.meshes), 205
GmshImporter2DIn3DSpace (class in fipy.meshes.gmshMesh), 218
GmshImporter3D (class in fipy.meshes), 205
GmshImporter3D (class in fipy.meshes.gmshMesh), 218
GnuplotViewer() (in module fipy.viewers), 429
GnuplotViewer() (in module fipy.viewers.gnuplotViewer), 442
grad (fipy.variables.CellVariable attribute), 392
grad (fipy.variables.cellVariable.CellVariable attribute), 408
grad (fipy.variables.ModularVariable attribute), 396
grad (fipy.variables.modularVariable.ModularVariable attribute), 418
Grid1D, 80, 85, 86, 92, 99, 108, 117, 138, 182
Grid1D (class in fipy.meshes.grid1D), 218
Grid1D() (in module fipy.meshes), 198
Grid1D() (in module fipy.meshes.factoryMeshes), 212
Grid2D, 69, 120, 135, 136, 151, 164, 180
Grid2D (class in fipy.meshes.grid2D), 218
Grid2D() (in module fipy.meshes), 197
Grid2D() (in module fipy.meshes.factoryMeshes), 212
Grid2DGistViewer, 182
Grid3D (class in fipy.meshes.grid3D), 219
Grid3D() (in module fipy.meshes), 197
Grid3D() (in module fipy.meshes.factoryMeshes), 211

H

harmonicFaceValue (fipy.variables.CellVariable attribute), 392
harmonicFaceValue (fipy.variables.cellVariable.CellVariable attribute), 408
HistogramVariable (class in fipy.variables), 403
HistogramVariable (class in fipy.variables.histogramVariable), 417
HybridConvectionTerm (class in fipy.terms), 324
HybridConvectionTerm (class in fipy.terms.hybridConvectionTerm), 334

I

ICPreconditioner (class in fipy.solvers.trilinos), 309
ICPreconditioner (class in fipy.solvers.trilinos.preconditioners), 314
ICPreconditioner (class in fipy.solvers.trilinos.preconditioners.icPreconditioner), 315
IllConditionedPreconditionerWarning, 293, 295
ImplicitDiffusionTerm (in module fipy.terms), 321
ImplicitDiffusionTerm (in module fipy.terms.implicitDiffusionTerm), 335
ImplicitSourceTerm, 95, 119, 121
ImplicitSourceTerm (class in fipy.terms), 321
ImplicitSourceTerm (class in fipy.terms.implicitSourceTerm), 335
inBaseUnits() (fipy.tools.dimensions.physicalField.PhysicalField method), 373
inBaseUnits() (fipy.tools.PhysicalField method), 349
inBaseUnits() (fipy.variables.Variable method), 386
inBaseUnits() (fipy.variables.variable.Variable method), 424
IncorrectSolutionVariable, 319
indices() (in module fipy.tools.numerix), 358, 361
inDimensionless() (fipy.tools.dimensions.physicalField.PhysicalField method), 374
inDimensionless() (fipy.tools.PhysicalField method), 349
initialize_options() (fipy.tools.copy_script.Copy_script method), 355
initialize_options() (fipy.tools.performance.efficiency_test.Efficiency_test method), 381
inRadians() (fipy.tools.dimensions.physicalField.PhysicalField method), 374
inRadians() (fipy.tools.PhysicalField method), 350
inSIUnits() (fipy.tools.dimensions.physicalField.PhysicalField method), 374
inSIUnits() (fipy.tools.PhysicalField method), 350
interfaceVar (fipy.models.levelSet.surfactant.SurfactantVariable attribute), 273
interfaceVar (fipy.models.levelSet.surfactant.surfactantVariable.SurfactantVariable attribute), 291
interfaceVar (fipy.models.levelSet.SurfactantVariable attribute), 246
interfaceVar (fipy.models.SurfactantVariable attribute), 231
interiorFaceCellIDs (fipy.meshes.abstractMesh.AbstractMesh attribute), 209
interiorFaceIDs (fipy.meshes.abstractMesh.AbstractMesh attribute), 209
interiorFaces (fipy.meshes.abstractMesh.AbstractMesh attribute), 209
inUnitsOf() (fipy.tools.dimensions.physicalField.PhysicalField method), 374
inUnitsOf() (fipy.tools.PhysicalField method), 350
inUnitsOf() (fipy.variables.Variable method), 386
inUnitsOf() (fipy.variables.variable.Variable method), 424
IPython, **49**
isAngle() (fipy.tools.dimensions.physicalField.PhysicalUnit method), 379
isclose() (in module fipy.tools.numerix), 362
isCompatible() (fipy.tools.dimensions.physicalField.PhysicalField method), 375
isCompatible() (fipy.tools.dimensions.physicalField.PhysicalUnit method), 379
isCompatible() (fipy.tools.PhysicalField method), 350
isDimensionless() (fipy.tools.dimensions.physicalField.PhysicalUnit method), 380
isDimensionlessOrAngle() (fipy.tools.dimensions.physicalField.PhysicalUnit method), 380
isFloat() (in module fipy.tools.numerix), 362
isInt() (in module fipy.tools.numerix), 362
isInverseAngle() (fipy.tools.dimensions.physicalField.PhysicalUnit method), 380
itemset() (fipy.tools.dimensions.physicalField.PhysicalField method), 375
itemset() (fipy.tools.PhysicalField method), 351
itemset() (fipy.variables.Variable method), 387
itemset() (fipy.variables.variable.Variable method), 424
itemsize (fipy.tools.dimensions.physicalField.PhysicalField attribute), 375
itemsize (fipy.tools.PhysicalField attribute), 351
itemsize (fipy.variables.Variable attribute), 387
itemsize (fipy.variables.variable.Variable attribute), 424
Iterator, 181

J

JacobiPreconditioner (class in fipy.solvers), 295
JacobiPreconditioner (class in fipy.solvers.pysparse), 301
JacobiPreconditioner (class in fipy.solvers.pysparse.preconditioners), 303
JacobiPreconditioner (class in fipy.solvers.pysparse.preconditioners.jacobiPreconditioner), 303
JacobiPreconditioner (class in fipy.solvers.trilinos), 309
JacobiPreconditioner (class in fipy.solvers.trilinos.preconditioners), 314
JacobiPreconditioner (class in fipy.solvers.trilinos.preconditioners.jacobiPreconditioner), 315
justErrorVector() (fipy.terms.term.Term method), 337
justResidualVector() (fipy.terms.term.Term method), 338

L

L1error() (in module fipy.steppers), 317
L1norm() (in module fipy.tools.numerix), 362
L2error() (in module fipy.steppers), 317
L2norm() (in module fipy.tools.numerix), 362

LD_LIBRARY_PATH, 17
leastSquaresGrad (fipy.variables.CellVariable attribute), 393
leastSquaresGrad (fipy.variables.cellVariable.CellVariable attribute), 409
LinearBicgstabSolver (class in fipy.solvers.scipy), 304
LinearBicgstabSolver (class in fipy.solvers.scipy.linearBicgstabSolver), 305
LinearBicgstabSolver (class in fipy.solvers.trilinos), 308
LinearBicgstabSolver (class in fipy.solvers.trilinos.linearBicgstabSolver), 310
LinearCGSSolver (class in fipy.solvers), 294
LinearCGSSolver (class in fipy.solvers.pyAMG), 297
LinearCGSSolver (class in fipy.solvers.pyAMG.linearCGSSolver), 298
LinearCGSSolver (class in fipy.solvers.pysparse), 300
LinearCGSSolver (class in fipy.solvers.pysparse.linearCGSSolver), 301
LinearCGSSolver (class in fipy.solvers.scipy), 304
LinearCGSSolver (class in fipy.solvers.scipy.linearCGSSolver), 305
LinearCGSSolver (class in fipy.solvers.trilinos), 307
LinearCGSSolver (class in fipy.solvers.trilinos.linearCGSSolver), 310
LinearGeneralSolver (class in fipy.solvers.pyAMG), 297
LinearGeneralSolver (class in fipy.solvers.pyAMG.linearGeneralSolver), 298
LinearGMRESSolver (class in fipy.solvers), 294
LinearGMRESSolver (class in fipy.solvers.pyAMG), 296
LinearGMRESSolver (class in fipy.solvers.pyAMG.linearGMRESSolver), 298
LinearGMRESSolver (class in fipy.solvers.pysparse), 300
LinearGMRESSolver (class in fipy.solvers.pysparse.linearGMRESSolver), 301
LinearGMRESSolver (class in fipy.solvers.scipy), 304
LinearGMRESSolver (class in fipy.solvers.scipy.linearGMRESSolver), 306
LinearGMRESSolver (class in fipy.solvers.trilinos), 307
LinearGMRESSolver (class in fipy.solvers.trilinos.linearGMRESSolver), 310
LinearJORSolver (class in fipy.solvers), 295
LinearJORSolver (class in fipy.solvers.pysparse), 301
LinearJORSolver (class in fipy.solvers.pysparse.linearJORSolver), 302
LinearLUSolver, 105, 181
LinearLUSolver (class in fipy.solvers), 294
LinearLUSolver (class in fipy.solvers.pyAMG), 297
LinearLUSolver (class in fipy.solvers.pyAMG.linearLUSolver), 299
LinearLUSolver (class in fipy.solvers.pysparse), 300
LinearLUSolver (class in fipy.solvers.pysparse.linearLUSolver), 302
LinearLUSolver (class in fipy.solvers.scipy), 304
LinearLUSolver (class in fipy.solvers.scipy.linearLUSolver), 306
LinearLUSolver (class in fipy.solvers.trilinos), 308
LinearLUSolver (class in fipy.solvers.trilinos.linearLUSolver), 311
LinearPCGSolver (class in fipy.solvers), 294
LinearPCGSolver (class in fipy.solvers.pyAMG), 297
LinearPCGSolver (class in fipy.solvers.pyAMG.linearPCGSolver), 299
LinearPCGSolver (class in fipy.solvers.pysparse), 300
LinearPCGSolver (class in fipy.solvers.pysparse.linearPCGSolver), 302
LinearPCGSolver (class in fipy.solvers.scipy), 305
LinearPCGSolver (class in fipy.solvers.scipy.linearPCGSolver), 306
LinearPCGSolver (class in fipy.solvers.trilinos), 307
LinearPCGSolver (class in fipy.solvers.trilinos.linearPCGSolver), 311
LINFerror() (in module fipy.steppers), 317
LINFnorm() (in module fipy.tools.numerix), 362
loadtxt, 120, 122, 154
log, 104, 110
log (fipy.viewers.Matplotlib1DViewer attribute), 431
log (fipy.viewers.matplotlibViewer.Matplotlib1DViewer attribute), 445
log (fipy.viewers.matplotlibViewer.matplotlib1DViewer.Matplotlib1DViewe attribute), 449
log (fipy.viewers.matplotlibViewer.matplotlibViewer.AbstractMatplotlibVie attribute), 455
log() (fipy.tools.dimensions.physicalField.PhysicalField method), 375
log() (fipy.tools.PhysicalField method), 351
log() (fipy.variables.Variable method), 387
log() (fipy.variables.variable.Variable method), 424
log10() (fipy.tools.dimensions.physicalField.PhysicalField method), 375
log10() (fipy.tools.PhysicalField method), 351
log10() (fipy.variables.Variable method), 387
log10() (fipy.variables.variable.Variable method), 425

M

mag (fipy.variables.Variable attribute), 387
mag (fipy.variables.variable.Variable attribute), 425
mathMethodDeprecated() (in module fipy.tools.decorators), 355
Matplotlib, **49**
Matplotlib1DViewer (class in fipy.viewers), 431
Matplotlib1DViewer (class in fipy.viewers.matplotlibViewer), 444

Matplotlib1DViewer (class in fipy.viewers.matplotlibViewer.matplotlib1DViewer), 449
Matplotlib2DGridContourViewer (class in fipy.viewers), 432
Matplotlib2DGridContourViewer (class in fipy.viewers.matplotlibViewer), 446
Matplotlib2DGridContourViewer (class in fipy.viewers.matplotlibViewer.matplotlib2DGridContourViewer), 450
Matplotlib2DGridViewer (class in fipy.viewers), 431
Matplotlib2DGridViewer (class in fipy.viewers.matplotlibViewer), 445
Matplotlib2DGridViewer (class in fipy.viewers.matplotlibViewer.matplotlib2DGridViewer), 451
Matplotlib2DViewer (class in fipy.viewers), 433
Matplotlib2DViewer (class in fipy.viewers.matplotlibViewer), 446
Matplotlib2DViewer (class in fipy.viewers.matplotlibViewer.matplotlib2DViewer), 452
MatplotlibSparseMatrixViewer (class in fipy.viewers.matplotlibViewer.matplotlibSparseMatrixViewer), 453
MatplotlibSurfactantViewer (class in fipy.models), 237
MatplotlibSurfactantViewer (class in fipy.models.levelSet), 253
MatplotlibSurfactantViewer (class in fipy.models.levelSet.surfactant), 280
MatplotlibSurfactantViewer (class in fipy.models.levelSet.surfactant.matplotlibSurfactantViewer), 286
MatplotlibVectorViewer (class in fipy.viewers), 433
MatplotlibVectorViewer (class in fipy.viewers.matplotlibViewer), 447
MatplotlibVectorViewer (class in fipy.viewers.matplotlibViewer.matplotlibVectorViewer), 453
MatplotlibViewer() (in module fipy.viewers), 429
MatplotlibViewer() (in module fipy.viewers.matplotlibViewer), 443
matrix (fipy.terms.term.Term attribute), 338
MatrixIllConditionedWarning, 293, 295
max() (fipy.variables.Variable method), 387
max() (fipy.variables.variable.Variable method), 425
MaxAll() (fipy.tools.comms.commWrapper.CommWrapper method), 365
MaxAll() (fipy.tools.comms.dummyComm.DummyComm method), 366
MaximumIterationWarning, 293, 295
MayaVi, **49**
Mayavi, **49**
MayaviClient (class in fipy.viewers), 435
MayaviClient (class in fipy.viewers.mayaviViewer), 455
MayaviClient (class in fipy.viewers.mayaviViewer.mayaviClient), 457
MayaviDaemon (class in fipy.viewers.mayaviViewer.mayaviDaemon), 458
MayaviSurfactantViewer, 154
MayaviSurfactantViewer (class in fipy.models), 236
MayaviSurfactantViewer (class in fipy.models.levelSet), 252
MayaviSurfactantViewer (class in fipy.models.levelSet.surfactant), 279
MayaviSurfactantViewer (class in fipy.models.levelSet.surfactant.mayaviSurfactantViewer), 287
MemoryHighWaterThread (class in fipy.tools.performance.memoryLogger), 381
MemoryLogger (class in fipy.tools.performance.memoryLogger), 381
Mesh (class in fipy.meshes.mesh), 219
Mesh1D (class in fipy.meshes.mesh1D), 219
Mesh2D (class in fipy.meshes.mesh2D), 220
MeshAdditionError, 219
MeshDimensionError, 438
method1() (package.subpackage.base.Base method), 188
method2() (package.subpackage.base.Base method), 188
method2() (package.subpackage.object.Object method), 189
min() (fipy.variables.Variable method), 387
min() (fipy.variables.variable.Variable method), 425
MinAll() (fipy.tools.comms.commWrapper.CommWrapper method), 365
MinAll() (fipy.tools.comms.dummyComm.DummyComm method), 366
minmodFaceValue (fipy.variables.CellVariable attribute), 393
minmodFaceValue (fipy.variables.cellVariable.CellVariable attribute), 409
ModularVariable, 121
ModularVariable (class in fipy.variables), 395
ModularVariable (class in fipy.variables.modularVariable), 417
module
  fipy.tools.dump, 123
  fipy.tools.parser, 120, 150
  fipy.viewers, 56, 70, 81, 86, 87, 92, 105, 119, 122, 165
  scipy, 75
Mpi4pyCommWrapper (class in fipy.tools.comms.mpi4pyCommWrapper), 366
MultilevelDDMLPreconditioner (class in fipy.solvers.trilinos), 309

MultilevelDDMLPreconditioner (class in fipy.solvers.trilinos.preconditioners), 314
MultilevelDDMLPreconditioner (class in fipy.solvers.trilinos.preconditioners.multilevelDDMLPreconditioner), 315
MultilevelDDPreconditioner (class in fipy.solvers.trilinos), 308
MultilevelDDPreconditioner (class in fipy.solvers.trilinos.preconditioners), 313
MultilevelDDPreconditioner (class in fipy.solvers.trilinos.preconditioners.multilevelDDPreconditioner), 315
MultilevelNSSAPreconditioner (class in fipy.solvers.trilinos), 309
MultilevelNSSAPreconditioner (class in fipy.solvers.trilinos.preconditioners), 314
MultilevelNSSAPreconditioner (class in fipy.solvers.trilinos.preconditioners.multilevelNSSAPreconditioner), 315
MultilevelSAPreconditioner (class in fipy.solvers.trilinos), 308
MultilevelSAPreconditioner (class in fipy.solvers.trilinos.preconditioners), 314
MultilevelSAPreconditioner (class in fipy.solvers.trilinos.preconditioners.multilevelSAPreconditioner), 315
MultilevelSGSPreconditioner (class in fipy.solvers.trilinos), 309
MultilevelSGSPreconditioner (class in fipy.solvers.trilinos.preconditioners), 314
MultilevelSGSPreconditioner (class in fipy.solvers.trilinos.preconditioners.multilevelSGSPreconditioner), 316
MultilevelSolverSmootherPreconditioner (class in fipy.solvers.trilinos), 309
MultilevelSolverSmootherPreconditioner (class in fipy.solvers.trilinos.preconditioners), 314
MultilevelSolverSmootherPreconditioner (class in fipy.solvers.trilinos.preconditioners.multilevelSolverSmootherPreconditioner), 316
multiply() (fipy.tools.dimensions.physicalField.PhysicalField method), 375
multiply() (fipy.tools.PhysicalField method), 351
MultiViewer (class in fipy.viewers), 436
MultiViewer (class in fipy.viewers.multiViewer), 439

N

name (fipy.variables.Variable attribute), 387
name (fipy.variables.variable.Variable attribute), 425
name() (fipy.tools.dimensions.physicalField.PhysicalUnit method), 380
nearest() (in module fipy.tools.numerix), 362
NoiseVariable (class in fipy.variables.noiseVariable), 418
Norm2() (fipy.tools.comms.commWrapper.CommWrapper method), 365
Norm2() (fipy.tools.comms.serialCommWrapper.SerialCommWrapper method), 366
Nproc (fipy.tools.comms.commWrapper.CommWrapper attribute), 365
Nproc (fipy.tools.comms.serialCommWrapper.SerialCommWrapper attribute), 366
NthOrderBoundaryCondition, 81
NthOrderBoundaryCondition (class in fipy.boundaryConditions), 191
NthOrderBoundaryCondition (class in fipy.boundaryConditions.nthOrderBoundaryCondition), 193
numarray, **49**
Numeric, **49**
numericValue (fipy.tools.dimensions.physicalField.PhysicalField attribute), 376
numericValue (fipy.tools.PhysicalField attribute), 351
numericValue (fipy.variables.Variable attribute), 387
numericValue (fipy.variables.variable.Variable attribute), 425
numerix, 182
numMeshDeprecated() (in module fipy.meshes.numMesh.deprecatedWarning), 225
NumPy, **49**

O

object
fipy.variables.cellVariable.CellVariable, 72
fipy.viewers.tsvViewer.TSVViewer, 74
Object (class in package.subpackage.object), 188
OffsetSparseMatrix() (in module fipy.matrices.offsetSparseMatrix), 195
old (fipy.variables.CellVariable attribute), 394
old (fipy.variables.cellVariable.CellVariable attribute), 410
openMSHFile() (in module fipy.meshes), 201
openMSHFile() (in module fipy.meshes.gmshMesh), 213
openPOSFile() (in module fipy.meshes), 201
openPOSFile() (in module fipy.meshes.gmshMesh), 214

P

package.subpackage (module), 187
package.subpackage.base (module), 187
package.subpackage.object (module), 188
ParallelCommWrapper (class in fipy.tools.comms.commWrapper), 366
parallelRandom() (fipy.variables.GaussianNoiseVariable method), 402
parallelRandom() (fipy.variables.gaussianNoiseVariable.GaussianNoiseVari method), 416

parallelRandom() (fipy.variables.noiseVariable.NoiseVariable method), 419
parse() (in module fipy.tools.parser), 364
parse_command_line() (fipy.viewers.mayaviViewer.mayaviDaemon.MayaviDaemon method), 458
PeriodicGrid1D (class in fipy.meshes), 199
PeriodicGrid1D (class in fipy.meshes.periodicGrid1D), 220
PeriodicGrid2D (class in fipy.meshes), 199
PeriodicGrid2D (class in fipy.meshes.periodicGrid2D), 221
PeriodicGrid2DLeftRight (class in fipy.meshes), 200
PeriodicGrid2DLeftRight (class in fipy.meshes.periodicGrid2D), 222
PeriodicGrid2DTopBottom (class in fipy.meshes), 200
PeriodicGrid2DTopBottom (class in fipy.meshes.periodicGrid2D), 222
PhysicalField (class in fipy.tools), 345
PhysicalField (class in fipy.tools.dimensions.physicalField), 369
physicalShape (fipy.meshes.SkewedGrid2D attribute), 200
physicalShape (fipy.meshes.skewedGrid2D.SkewedGrid2D attribute), 222
physicalShape (fipy.meshes.Tri2D attribute), 201
physicalShape (fipy.meshes.tri2D.Tri2D attribute), 223
PhysicalUnit (class in fipy.tools.dimensions.physicalField), 378
pi, 114, 119, 121
PIDStepper (class in fipy.steppers.pidStepper), 318
pip, **49**
plot() (fipy.models.levelSet.MayaviSurfactantViewer method), 253
plot() (fipy.models.levelSet.surfactant.MayaviSurfactantViewer method), 280
plot() (fipy.models.levelSet.surfactant.mayaviSurfactantViewer.MayaviSurfactantViewer method), 288
plot() (fipy.models.MayaviSurfactantViewer method), 237
plot() (fipy.viewers.DummyViewer method), 438
plot() (fipy.viewers.matplotlibViewer.matplotlibSparseMatrixViewer.MatplotlibSparseMatrixViewer method), 453
plot() (fipy.viewers.matplotlibViewer.matplotlibViewer.AbstractMatplotlibViewer method), 455
plot() (fipy.viewers.MayaviClient method), 436
plot() (fipy.viewers.mayaviViewer.MayaviClient method), 457
plot() (fipy.viewers.mayaviViewer.mayaviClient.MayaviClient method), 458
plot() (fipy.viewers.MultiViewer method), 436
plot() (fipy.viewers.multiViewer.MultiViewer method), 439
plot() (fipy.viewers.TSVViewer method), 437
plot() (fipy.viewers.tsvViewer.TSVViewer method), 440
plot() (fipy.viewers.viewer.AbstractViewer method), 441
plot() (fipy.viewers.vtkViewer.vtkViewer.VTKViewer method), 461
plotMesh() (fipy.viewers.viewer.AbstractViewer method), 441
poll_file() (fipy.viewers.mayaviViewer.mayaviDaemon.MayaviDaemon method), 459
PowerLawConvectionTerm, 110
PowerLawConvectionTerm (class in fipy.terms), 325
PowerLawConvectionTerm (class in fipy.terms.powerLawConvectionTerm), 335
Preconditioner (class in fipy.solvers.pysparse.preconditioners.preconditioner), 303
Preconditioner (class in fipy.solvers.trilinos.preconditioners.preconditioner), 316
PreconditionerNotPositiveDefiniteWarning, 293, 295
PreconditionerWarning, 293, 295
PRINT() (in module fipy.tools.debug), 355
procID (fipy.tools.comms.commWrapper.CommWrapper attribute), 366
procID (fipy.tools.comms.serialCommWrapper.SerialCommWrapper attribute), 366
prune() (in module fipy.tools.vector), 364
PseudoRKQSStepper (class in fipy.steppers.pseudoRKQSStepper), 318
put() (fipy.tools.dimensions.physicalField.PhysicalField method), 376
put() (fipy.tools.PhysicalField method), 352
put() (fipy.variables.Variable method), 387
put() (fipy.variables.variable.Variable method), 425
put() (in module fipy.tools.numerix), 358, 359, 362
putAdd() (in module fipy.tools.vector), 364
PyAMG, **49**
PyPI, **49**
Pyrex, **49**
PySparse, **49**
PysparseSolver (class in fipy.solvers.pysparse.pysparseSolver), 303
Python, **49**
Python 3, **49**
Python Enhancement Proposals
PEP 3000, 49
PYTHONPATH, 13, 17
PyTrilinos, **50**
PyxViewer, **50**

Q

quiver() (fipy.viewers.MatplotlibVectorViewer method), 435
quiver() (fipy.viewers.matplotlibViewer.MatplotlibVectorViewer method), 448

quiver() (fipy.viewers.matplotlibViewer.matplotlibVectorViewer.MatplotlibVectorViewer method), 454

R

random() (fipy.variables.BetaNoiseVariable method), 398
random() (fipy.variables.betaNoiseVariable.BetaNoiseVariable method), 405
random() (fipy.variables.ExponentialNoiseVariable method), 399
random() (fipy.variables.exponentialNoiseVariable.ExponentialNoiseVariable method), 412
random() (fipy.variables.GammaNoiseVariable method), 400
random() (fipy.variables.gammaNoiseVariable.GammaNoiseVariable method), 414
random() (fipy.variables.noiseVariable.NoiseVariable method), 419
random() (fipy.variables.UniformNoiseVariable method), 403
random() (fipy.variables.uniformNoiseVariable.UniformNoiseVariable method), 420
rank() (in module fipy.tools.numerix), 359, 360, 363
ravel() (fipy.tools.dimensions.physicalField.PhysicalField method), 376
ravel() (fipy.tools.PhysicalField method), 352
ravel() (fipy.variables.Variable method), 387
ravel() (fipy.variables.variable.Variable method), 425
read() (in module fipy.tools.dump), 356
register_skipper() (in module fipy.tests.doctestPlus), 343
release() (fipy.variables.CellVariable method), 394
release() (fipy.variables.cellVariable.CellVariable method), 410
release() (fipy.variables.Variable method), 387
release() (fipy.variables.variable.Variable method), 425
report_skips() (in module fipy.tests.doctestPlus), 343
reshape() (fipy.tools.dimensions.physicalField.PhysicalField method), 376
reshape() (fipy.tools.PhysicalField method), 352
reshape() (fipy.variables.Variable method), 387
reshape() (fipy.variables.variable.Variable method), 425
reshape() (in module fipy.tools.numerix), 358, 359, 363
ResidualTerm (class in fipy.terms), 321
ResidualTerm (class in fipy.terms.residualTerm), 336
residualVectorAndNorm() (fipy.terms.term.Term method), 338
RHSvector (fipy.terms.term.Term attribute), 337
run() (fipy.tools.copy_script.Copy_script method), 355
run() (fipy.tools.performance.efficiency_test.Efficiency_test method), 381
run() (fipy.tools.performance.memoryLogger.MemoryHighWaterThread method), 381
run() (fipy.viewers.mayaviViewer.mayaviDaemon.MayaviDaemon method), 459
runGold, 144
runLeveler, 146
runSimpleTrenchSystem, 141

S

save() (fipy.tools.Vitals method), 354
save() (fipy.tools.vitals.Vitals method), 365
ScalarQuantityOutOfRangeWarning, 293, 296
scale (fipy.meshes.abstractMesh.AbstractMesh attribute), 209
scaledCellDistances (fipy.meshes.abstractMesh.AbstractMesh attribute), 209
scaledCellToCellDistances (fipy.meshes.abstractMesh.AbstractMesh attribute), 209
scaledCellVolumes (fipy.meshes.abstractMesh.AbstractMesh attribute), 209
scaledFaceAreas (fipy.meshes.abstractMesh.AbstractMesh attribute), 209
scaledFaceToCellDistances (fipy.meshes.abstractMesh.AbstractMesh attribute), 209
ScharfetterGummelFaceVariable (class in fipy.variables), 395
ScharfetterGummelFaceVariable (class in fipy.variables.scharfetterGummelFaceVariable), 419
ScientificPython, **50**
SciPy, **50**, 98, 104
scipy
  module, 75
scramble() (fipy.variables.noiseVariable.NoiseVariable method), 419
SerialCommWrapper (class in fipy.tools.comms.serialCommWrapper), 366
setLimits() (fipy.viewers.MultiViewer method), 436
setLimits() (fipy.viewers.multiViewer.MultiViewer method), 439
setLimits() (fipy.viewers.viewer.AbstractViewer method), 441
setName() (fipy.tools.dimensions.physicalField.PhysicalUnit method), 380
setName() (fipy.variables.Variable method), 387
setName() (fipy.variables.variable.Variable method), 425
setScale() (fipy.meshes.abstractMesh.AbstractMesh method), 209
setUnit() (fipy.tools.dimensions.physicalField.PhysicalField method), 376
setUnit() (fipy.tools.PhysicalField method), 352
setUnit() (fipy.variables.Variable method), 388
setUnit() (fipy.variables.variable.Variable method), 425
setup_source() (fipy.viewers.mayaviViewer.mayaviDaemon.MayaviDaemon method), 459
setValue() (fipy.variables.CellVariable method), 394

setValue() (fipy.variables.cellVariable.CellVariable method), 410
setValue() (fipy.variables.FaceVariable method), 395
setValue() (fipy.variables.faceVariable.FaceVariable method), 413
setValue() (fipy.variables.Variable method), 388
setValue() (fipy.variables.variable.Variable method), 425
shape (fipy.meshes.SkewedGrid2D attribute), 200
shape (fipy.meshes.skewedGrid2D.SkewedGrid2D attribute), 222
shape (fipy.meshes.Tri2D attribute), 201
shape (fipy.meshes.tri2D.Tri2D attribute), 223
shape (fipy.tools.dimensions.physicalField.PhysicalField attribute), 376
shape (fipy.tools.PhysicalField attribute), 352
shape (fipy.variables.Variable attribute), 388
shape (fipy.variables.variable.Variable attribute), 426
sign() (fipy.tools.dimensions.physicalField.PhysicalField method), 376
sign() (fipy.tools.PhysicalField method), 352
sign() (fipy.variables.Variable method), 388
sign() (fipy.variables.variable.Variable method), 426
sin() (fipy.tools.dimensions.physicalField.PhysicalField method), 376
sin() (fipy.tools.PhysicalField method), 352
sin() (fipy.variables.Variable method), 388
sin() (fipy.variables.variable.Variable method), 426
sinh() (fipy.tools.dimensions.physicalField.PhysicalField method), 377
sinh() (fipy.tools.PhysicalField method), 353
sinh() (fipy.variables.Variable method), 388
sinh() (fipy.variables.variable.Variable method), 426
SkewedGrid2D (class in fipy.meshes), 200
SkewedGrid2D (class in fipy.meshes.skewedGrid2D), 222
SmoothedAggregationPreconditioner (class in fipy.solvers.pyAMG.preconditioners.smoothedAggregationPreconditioner), 299
SolutionVariableNumberError, 319
SolutionVariableRequiredError, 319
solve, 105
solve() (fipy.models.AdsorbingSurfactantEquation method), 235
solve() (fipy.models.levelSet.AdsorbingSurfactantEquation method), 250
solve() (fipy.models.levelSet.surfactant.AdsorbingSurfactantEquation method), 277
solve() (fipy.models.levelSet.surfactant.adsorbingSurfactantEquation.AdsorbingSurfactantEquation method), 285
solve() (fipy.models.levelSet.surfactant.SurfactantEquation method), 273
solve() (fipy.models.levelSet.surfactant.surfactantEquation.SurfactantEquation method), 290
solve() (fipy.models.levelSet.SurfactantEquation method), 246
solve() (fipy.models.SurfactantEquation method), 231
solve() (fipy.solvers.trilinos.trilinosNonlinearSolver.TrilinosNonlinearSolver method), 313
solve() (fipy.terms.term.Term method), 339
Solver (class in fipy.solvers), 293
Solver (class in fipy.solvers.solver), 296
SolverConvergenceWarning, 293, 295
SourceTerm (class in fipy.terms.sourceTerm), 337
Sphinx, **50**
sqrt, 93, 120, 150
arcsin
cos, 75
sqrt() (fipy.tools.dimensions.physicalField.PhysicalField method), 377
sqrt() (fipy.tools.PhysicalField method), 353
sqrt() (fipy.variables.Variable method), 388
sqrt() (fipy.variables.variable.Variable method), 426
sqrtDot() (in module fipy.tools.numerix), 363
SsorPreconditioner (class in fipy.solvers), 295
SsorPreconditioner (class in fipy.solvers.pysparse), 301
SsorPreconditioner (class in fipy.solvers.pysparse.preconditioners), 303
SsorPreconditioner (class in fipy.solvers.pysparse.preconditioners.ssorPreconditioner), 304
StagnatedSolverWarning, 293, 296
start() (fipy.tools.performance.memoryLogger.MemoryLogger method), 381
SteadyConvectionDiffusionScEquation, 181
step() (fipy.steppers.stepper.Stepper method), 318
Stepper (class in fipy.steppers.stepper), 318
stop() (fipy.tools.performance.memoryLogger.MemoryHighWaterThread method), 381
stop() (fipy.tools.performance.memoryLogger.MemoryLogger method), 381
subscribedVariables (fipy.variables.Variable attribute), 389
subscribedVariables (fipy.variables.variable.Variable attribute), 426
subtract() (fipy.tools.dimensions.physicalField.PhysicalField method), 377
subtract() (fipy.tools.PhysicalField method), 353
successFn() (fipy.steppers.stepper.Stepper static method), 318
sum() (fipy.tools.comms.commWrapper.CommWrapper method), 366
sum() (fipy.tools.comms.dummyComm.DummyComm method), 366
sum() (fipy.tools.dimensions.physicalField.PhysicalField method), 377
sum() (fipy.tools.PhysicalField method), 353
sum() (fipy.variables.Variable method), 389

sum() (fipy.variables.variable.Variable method), 426
sum() (in module fipy.tools.numerix), 359, 360, 363
SurfactantEquation (class in fipy.models), 231
SurfactantEquation (class in fipy.models.levelSet), 246
SurfactantEquation (class in fipy.models.levelSet.surfactant), 273
SurfactantEquation (class in fipy.models.levelSet.surfactant.surfactantEquation), 290
SurfactantVariable, 151
SurfactantVariable (class in fipy.models), 230
SurfactantVariable (class in fipy.models.levelSet), 245
SurfactantVariable (class in fipy.models.levelSet.surfactant), 272
SurfactantVariable (class in fipy.models.levelSet.surfactant.surfactantVariable), 290
svn() (fipy.tools.Vitals method), 354
svn() (fipy.tools.vitals.Vitals method), 365
svncmd() (fipy.tools.Vitals method), 354
svncmd() (fipy.tools.vitals.Vitals method), 365
sweep, 98, 105, 165
sweep() (fipy.models.AdsorbingSurfactantEquation method), 236
sweep() (fipy.models.levelSet.AdsorbingSurfactantEquation method), 251
sweep() (fipy.models.levelSet.surfactant.AdsorbingSurfactantEquation method), 277
sweep() (fipy.models.levelSet.surfactant.adsorbingSurfactantEquation.AdsorbingSurfactantEquation method), 285
sweep() (fipy.models.levelSet.surfactant.SurfactantEquation method), 273
sweep() (fipy.models.levelSet.surfactant.surfactantEquation.SurfactantEquation method), 290
sweep() (fipy.models.levelSet.SurfactantEquation method), 247
sweep() (fipy.models.SurfactantEquation method), 232
sweep() (fipy.terms.term.Term method), 339
sweepFn() (fipy.steppers.stepper.Stepper static method), 318
sweepMonotonic() (in module fipy.steppers), 318

T

take() (fipy.tools.dimensions.physicalField.PhysicalField method), 377
take() (fipy.tools.PhysicalField method), 353
take() (fipy.variables.Variable method), 389
take() (fipy.variables.variable.Variable method), 426
take() (in module fipy.tools.numerix), 358, 359, 363
tan, 114
tan() (fipy.tools.dimensions.physicalField.PhysicalField method), 378
tan() (fipy.tools.PhysicalField method), 354
tan() (fipy.variables.Variable method), 389
tan() (fipy.variables.variable.Variable method), 426
tanh, 93
tanh() (fipy.tools.dimensions.physicalField.PhysicalField method), 378
tanh() (fipy.tools.PhysicalField method), 354
tanh() (fipy.variables.Variable method), 389
tanh() (fipy.variables.variable.Variable method), 426
Term (class in fipy.terms.term), 337
TermMultiplyError, 319
testmod() (in module fipy.tests.doctestPlus), 343
tostring() (fipy.tools.dimensions.physicalField.PhysicalField method), 378
tostring() (fipy.tools.PhysicalField method), 354
tostring() (fipy.variables.Variable method), 389
tostring() (fipy.variables.variable.Variable method), 426
tostring() (in module fipy.tools.numerix), 363
TransientTerm, 56, 94, 119, 121
TransientTerm (class in fipy.terms), 319
TransientTerm (class in fipy.terms.transientTerm), 339
TrenchMesh (class in fipy.models), 240
TrenchMesh (class in fipy.models.levelSet), 256
TrenchMesh (class in fipy.models.levelSet.electroChem), 267
TrenchMesh (class in fipy.models.levelSet.electroChem.gapFillMesh), 268
Tri2D (class in fipy.meshes), 200
Tri2D (class in fipy.meshes.tri2D), 223
Trilinos, **50**
TrilinosAztecOOSolver (class in fipy.solvers.trilinos.trilinosAztecOOSolver), 311
TrilinosMLTest (class in fipy.solvers.trilinos), 308
TrilinosMLTest (class in fipy.solvers.trilinos.trilinosMLTest), 312
TrilinosNonlinearSolver (class in fipy.solvers.trilinos.trilinosNonlinearSolver), 313
TrilinosSolver (class in fipy.solvers.trilinos.trilinosSolver), 313
TSVViewer (class in fipy.viewers), 436
TSVViewer (class in fipy.viewers.tsvViewer), 440
tupleToXML() (fipy.tools.Vitals method), 354
tupleToXML() (fipy.tools.vitals.Vitals method), 365

U

UniformGrid (class in fipy.meshes.uniformGrid), 223
UniformGrid1D (class in fipy.meshes.uniformGrid1D), 223
UniformGrid2D (class in fipy.meshes.uniformGrid2D), 224
UniformGrid3D (class in fipy.meshes.uniformGrid3D), 224
UniformNoiseVariable (class in fipy.variables), 402

UniformNoiseVariable (class in fipy.variables.uniformNoiseVariable), 419
unit (fipy.tools.dimensions.physicalField.PhysicalField attribute), 378
unit (fipy.tools.PhysicalField attribute), 354
unit (fipy.variables.Variable attribute), 389
unit (fipy.variables.variable.Variable attribute), 426
update_pipeline() (fipy.viewers.mayaviViewer.mayaviDaemon.MayaviDaemon method), 459
updateOld() (fipy.variables.CellVariable method), 394
updateOld() (fipy.variables.cellVariable.CellVariable method), 410
updateOld() (fipy.variables.ModularVariable method), 396
updateOld() (fipy.variables.modularVariable.ModularVariable method), 418
UpwindConvectionTerm (class in fipy.terms), 326
UpwindConvectionTerm (class in fipy.terms.upwindConvectionTerm), 340
user_options (fipy.tools.copy_script.Copy_script attribute), 355
user_options (fipy.tools.performance.efficiency_test.Efficiency_test attribute), 381

V

value (fipy.variables.Variable attribute), 389
value (fipy.variables.variable.Variable attribute), 427
VanLeerConvectionTerm (class in fipy.terms), 327
VanLeerConvectionTerm (class in fipy.terms.vanLeerConvectionTerm), 342
Variable, 97, 100
Variable (class in fipy.variables), 383
Variable (class in fipy.variables.variable), 420
VectorCoeffError, 319
vertexCoords (fipy.meshes.uniformGrid1D.UniformGrid1D attribute), 224
vertexCoords (fipy.meshes.uniformGrid2D.UniformGrid2D attribute), 224
vertexCoords (fipy.meshes.uniformGrid3D.UniformGrid3D attribute), 225
view_data() (fipy.viewers.mayaviViewer.mayaviDaemon.MayaviDaemon method), 459
Viewer() (in module fipy.viewers), 438
Vitals (class in fipy.tools), 354
Vitals (class in fipy.tools.vitals), 365
VTKCellDataSet (fipy.meshes.abstractMesh.AbstractMesh attribute), 205
VTKCellViewer (class in fipy.viewers), 438
VTKCellViewer (class in fipy.viewers.vtkViewer), 459
VTKCellViewer (class in fipy.viewers.vtkViewer.vtkCellViewer), 460
VTKFaceDataSet (fipy.meshes.abstractMesh.AbstractMesh attribute), 206
VTKFaceViewer (class in fipy.viewers), 438
VTKFaceViewer (class in fipy.viewers.vtkViewer), 460
VTKFaceViewer (class in fipy.viewers.vtkViewer.vtkFaceViewer), 460
VTKViewer (class in fipy.viewers.vtkViewer.vtkViewer), 461
VTKViewer() (in module fipy.viewers), 437
VTKViewer() (in module fipy.viewers.vtkViewer), 459

W

write() (in module fipy.tools.dump), 356

X

x (fipy.meshes.abstractMesh.AbstractMesh attribute), 209

Y

y (fipy.meshes.abstractMesh.AbstractMesh attribute), 209

Z

z (fipy.meshes.abstractMesh.AbstractMesh attribute), 209