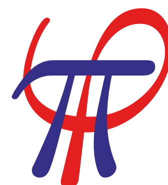


FiPy

Programmer's Reference

Daniel Wheeler
Jonathan E. Guyer
James A. Warren

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

June 11, 2009

Version 2.0.2

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

1 How To Read This Manual **11**

2 Package fipy.boundaryConditions **15**

2.1 Module fipy.boundaryConditions.boundaryCondition 15

2.2 Module fipy.boundaryConditions.fixedFlux 16

2.3 Module fipy.boundaryConditions.fixedValue 17

2.4 Module fipy.boundaryConditions.nthOrderBoundaryCondition 17

2.5 Module fipy.boundaryConditions.test 18

3 Package fipy.meshes **19**

3.1 Package fipy.meshes.common 19

3.2 Module fipy.meshes.common.mesh 19

3.3 Module fipy.meshes.cylindricalGrid1D 27

3.4 Module fipy.meshes.cylindricalGrid2D 28

3.5 Module fipy.meshes.gmshExport 29

3.6 Module fipy.meshes.gmshImport 29

3.7 Module fipy.meshes.grid1D 30

3.8 Module fipy.meshes.grid2D 31

3.9 Module fipy.meshes.grid3D 32

3.10 Package fipy.meshes.numMesh 33

3.11 Module fipy.meshes.numMesh.cell 33

3.12 Module fipy.meshes.numMesh.cylindricalGrid1D 34

3.13 Module fipy.meshes.numMesh.cylindricalGrid2D 35

3.14 Module fipy.meshes.numMesh.cylindricalUniformGrid1D 37

3.15 Module fipy.meshes.numMesh.cylindricalUniformGrid2D . . . 39
3.16 Module fipy.meshes.numMesh.face . . . 40
3.17 Module fipy.meshes.numMesh.gmshExport . . . 42
3.18 Module fipy.meshes.numMesh.gmshImport . . . 43
3.19 Module fipy.meshes.numMesh.grid1D . . . 50
3.20 Module fipy.meshes.numMesh.grid2D . . . 52
3.21 Module fipy.meshes.numMesh.grid3D . . . 54
3.22 Module fipy.meshes.numMesh.mesh . . . 56
3.23 Module fipy.meshes.numMesh.mesh1D . . . 63
3.24 Module fipy.meshes.numMesh.mesh2D . . . 65
3.25 Module fipy.meshes.numMesh.periodicGrid1D . . . 67
3.26 Module fipy.meshes.numMesh.periodicGrid2D . . . 69
3.27 Module fipy.meshes.numMesh.skewedGrid2D . . . 74
3.28 Module fipy.meshes.numMesh.test . . . 75
3.29 Module fipy.meshes.numMesh.tri2D . . . 76
3.30 Module fipy.meshes.numMesh.uniformGrid1D . . . 77
3.31 Module fipy.meshes.numMesh.uniformGrid2D . . . 80
3.32 Module fipy.meshes.numMesh.uniformGrid3D . . . 84
3.33 Module fipy.meshes.periodicGrid1D . . . 86
3.34 Module fipy.meshes.periodicGrid2D . . . 86
3.35 Package fipy.meshes.pyMesh . . . 86
3.36 Module fipy.meshes.pyMesh.cell . . . 87
3.37 Module fipy.meshes.pyMesh.face . . . 88
3.38 Module fipy.meshes.pyMesh.face2D . . . 90
3.39 Module fipy.meshes.pyMesh.grid2D . . . 90
3.40 Module fipy.meshes.pyMesh.mesh . . . 94
3.41 Module fipy.meshes.pyMesh.test . . . 95
3.42 Module fipy.meshes.pyMesh.vertex . . . 95
3.43 Module fipy.meshes.skewedGrid2D . . . 96
3.44 Module fipy.meshes.test . . . 96
3.45 Module fipy.meshes.tri2D . . . 96

4 Package fipy.models 97

4.1 Package fipy.models.levelSet . . . 97
4.2 Package fipy.models.levelSet.advection . . . 97
4.3 Module fipy.models.levelSet.advection.advectionEquation . . . 98
4.4 Module fipy.models.levelSet.advection.advectionTerm . . . 99
4.5 Module fipy.models.levelSet.advection.higherOrderAdvectionEquation . . . 100
4.6 Module fipy.models.levelSet.advection.higherOrderAdvectionTerm . . . 101
4.7 Package fipy.models.levelSet.distanceFunction . . . 101
4.8 Module fipy.models.levelSet.distanceFunction.distanceVariable . . . 101
4.9 Module fipy.models.levelSet.distanceFunction.levelSetDiffusionEquation . . . 106
4.10 Module fipy.models.levelSet.distanceFunction.levelSetDiffusionVariable . . . 106
4.11 Package fipy.models.levelSet.electroChem . . . 106
4.12 Module fipy.models.levelSet.electroChem.gapFillMesh . . . 106
4.13 Module fipy.models.levelSet.electroChem.metalIonDiffusionEquation . . . 113
4.14 Module fipy.models.levelSet.electroChem.metalIonSourceVariable . . . 115
4.15 Module fipy.models.levelSet.electroChem.test . . . 115
4.16 Package fipy.models.levelSet.surfactant . . . 115
4.17 Module fipy.models.levelSet.surfactant.adsorbingSurfactantEquation . . . 115
4.18 Module fipy.models.levelSet.surfactant.convectionCoeff . . . 120
4.19 Module fipy.models.levelSet.surfactant.lines . . . 120
4.20 Module fipy.models.levelSet.surfactant.matplotlibSurfactantViewer . . . 120
4.21 Module fipy.models.levelSet.surfactant.mayaviSurfactantViewer . . . 122
4.22 Module fipy.models.levelSet.surfactant.surfactantBulkDiffusionEquation . . . 125
4.23 Module fipy.models.levelSet.surfactant.surfactantEquation . . . 126
4.24 Module fipy.models.levelSet.surfactant.surfactantVariable . . . 127
4.25 Module fipy.models.levelSet.test . . . 130
4.26 Module fipy.models.test . . . 130

5 Package fipy.solvers 131

5.1 Package fipy.solvers.pysparse . . . 131
5.2 Module fipy.solvers.pysparse.linearCGSSolver . . . 131

5.3 Module fipy.solvers.pysparse.linearGMRESSolver 132
5.4 Module fipy.solvers.pysparse.linearJORSolver 133
5.5 Module fipy.solvers.pysparse.linearLUSolver 134
5.6 Module fipy.solvers.pysparse.linearPCGSolver 135
5.7 Module fipy.solvers.pysparse.pysparseSolver 135
5.8 Module fipy.solvers.solver 136
5.9 Module fipy.solvers.test 146
5.10 Package fipy.solvers.trilinos 146
5.11 Module fipy.solvers.trilinos.linearBicgstabSolver 146
5.12 Module fipy.solvers.trilinos.linearCGSSolver 147
5.13 Module fipy.solvers.trilinos.linearGMRESSolver 148
5.14 Module fipy.solvers.trilinos.linearLUSolver 149
5.15 Module fipy.solvers.trilinos.linearPCGSolver 150
5.16 Package fipy.solvers.trilinos.preconditioners 151
5.17 Module fipy.solvers.trilinos.preconditioners.domDecompPreconditioner 151
5.18 Module fipy.solvers.trilinos.preconditioners.icPreconditioner 151
5.19 Module fipy.solvers.trilinos.preconditioners.jacobiPreconditioner 152
5.20 Module fipy.solvers.trilinos.preconditioners.multilevelDDPreconditioner 152
5.21 Module fipy.solvers.trilinos.preconditioners.multilevelSGSPreconditioner 153
5.22 Module fipy.solvers.trilinos.preconditioners.multilevelSolverSmootherPreconditioner 153
5.23 Module fipy.solvers.trilinos.preconditioners.preconditioner 154
5.24 Module fipy.solvers.trilinos.trilinosAztecOOSolver 155
5.25 Module fipy.solvers.trilinos.trilinosMLTest 156
5.26 Module fipy.solvers.trilinos.trilinosSolver 157

6 Package fipy.steppers 159

6.1 Functions 159
6.2 Module fipy.steppers.pidStepper 162
6.3 Module fipy.steppers.pseudoRKQSStepper 162
6.4 Module fipy.steppers.stepper 163

7 Package fipy.terms 165

7.1 Module fipy.terms.cellTerm 165
7.2 Module fipy.terms.centralDiffConvectionTerm 166
7.3 Module fipy.terms.collectedDiffusionTerm 167
7.4 Module fipy.terms.convectionTerm 167
7.5 Module fipy.terms.diffusionTerm 169
7.6 Module fipy.terms.equation 171
7.7 Module fipy.terms.explicitDiffusionTerm 171
7.8 Module fipy.terms.explicitSourceTerm 172
7.9 Module fipy.terms.explicitUpwindConvectionTerm 172
7.10 Module fipy.terms.exponentialConvectionTerm 173
7.11 Module fipy.terms.faceTerm 174
7.12 Module fipy.terms.hybridConvectionTerm 175
7.13 Module fipy.terms.implicitDiffusionTerm 176
7.14 Module fipy.terms.implicitSourceTerm 176
7.15 Module fipy.terms.nthOrderDiffusionTerm 177
7.16 Module fipy.terms.powerLawConvectionTerm 179
7.17 Module fipy.terms.sourceTerm 180
7.18 Module fipy.terms.term 181
7.19 Module fipy.terms.test 187
7.20 Module fipy.terms.transientTerm 187
7.21 Module fipy.terms.upwindConvectionTerm 189
7.22 Module fipy.terms.vanLeerConvectionTerm 190
7.23 Module fipy.test 190
7.24 Package fipy.tests 190
7.25 Module fipy.tests.doctestPlus 191
7.26 Module fipy.tests.lateImportTest 192
7.27 Module fipy.tests.testBase 192
7.28 Module fipy.tests.testProgram 192

8 Package fipy.tools **193**

8.1 Variables 193

8.2 Package fipy.tools.dimensions . . . 197
8.3 Module fipy.tools.dimensions.DictWithDefault . . . 197
8.4 Module fipy.tools.dimensions.NumberDict . . . 197
8.5 Module fipy.tools.dimensions.physicalField . . . 198
8.6 Module fipy.tools.dump . . . 230
8.7 Module fipy.tools.inline . . . 231
8.8 Module fipy.tools.memoryLeak . . . 232
8.9 Module fipy.tools.memoryLogger . . . 232
8.10 Module fipy.tools.memoryUsage . . . 233
8.11 Module fipy.tools.numerix . . . 234
8.12 Module fipy.tools.parser . . . 251
8.13 Module fipy.tools.pysparseMatrix . . . 252
8.14 Module fipy.tools.sparseMatrix . . . 252
8.15 Module fipy.tools.test . . . 252
8.16 Module fipy.tools.trilinosMatrix . . . 252
8.17 Module fipy.tools.vector . . . 253

9 Package fipy.variables 255

9.1 Module fipy.variables.addOverFacesVariable . . . 255
9.2 Module fipy.variables.arithmeticCellToFaceVariable . . . 255
9.3 Module fipy.variables.betaNoiseVariable . . . 255
9.4 Module fipy.variables.binaryOperatorVariable . . . 258
9.5 Module fipy.variables.cellToFaceVariable . . . 258
9.6 Module fipy.variables.cellVariable . . . 258
9.7 Module fipy.variables.cellVolumeAverageVariable . . . 267
9.8 Module fipy.variables.constant . . . 267
9.9 Module fipy.variables.exponentialNoiseVariable . . . 267
9.10 Module fipy.variables.faceGradContributionsVariable . . . 270
9.11 Module fipy.variables.faceGradVariable . . . 270
9.12 Module fipy.variables.faceVariable . . . 270
9.13 Module fipy.variables.fixedBCFaceGradVariable . . . 272

9.14 Module fipy.variables.gammaNoiseVariable 272
9.15 Module fipy.variables.gaussCellGradVariable 275
9.16 Module fipy.variables.gaussianNoiseVariable 275
9.17 Module fipy.variables.harmonicCellToFaceVariable 279
9.18 Module fipy.variables.histogramVariable 279
9.19 Module fipy.variables.leastSquaresCellGradVariable 281
9.20 Module fipy.variables.meshVariable 281
9.21 Module fipy.variables.minmodCellToFaceVariable 281
9.22 Module fipy.variables.modCellGradVariable 281
9.23 Module fipy.variables.modCellToFaceVariable 281
9.24 Module fipy.variables.modFaceGradVariable 281
9.25 Module fipy.variables.modPhysicalField 281
9.26 Module fipy.variables.modularVariable 281
9.27 Module fipy.variables.noiseVariable 284
9.28 Module fipy.variables.operatorVariable 286
9.29 Module fipy.variables.scharfetterGummelFaceVariable 286
9.30 Module fipy.variables.test 288
9.31 Module fipy.variables.unaryOperatorVariable 288
9.32 Module fipy.variables.uniformNoiseVariable 288
9.33 Module fipy.variables.variable 291

10 Package fipy.viewers 309

10.1 Functions 309
10.2 Package fipy.viewers.gistViewer 311
10.3 Module fipy.viewers.gistViewer.colorbar 318
10.4 Module fipy.viewers.gistViewer.gist1DViewer 318
10.5 Module fipy.viewers.gistViewer.gist2DViewer 319
10.6 Module fipy.viewers.gistViewer.gistVectorViewer 322
10.7 Module fipy.viewers.gistViewer.gistViewer 324
10.8 Module fipy.viewers.gistViewer.test 324
10.9 Package fipy.viewers.gnuplotViewer 325

10.10Module fipy.viewers.gnuplotViewer.gnuplot1DViewer 329
10.11Module fipy.viewers.gnuplotViewer.gnuplot2DViewer 330
10.12Module fipy.viewers.gnuplotViewer.gnuplotViewer 331
10.13Module fipy.viewers.gnuplotViewer.test 331
10.14Package fipy.viewers.matplotlibViewer 332
10.15Module fipy.viewers.matplotlibViewer.matplotlib1DViewer 340
10.16Module fipy.viewers.matplotlibViewer.matplotlib2DGridContourViewer 341
10.17Module fipy.viewers.matplotlibViewer.matplotlib2DGridViewer 342
10.18Module fipy.viewers.matplotlibViewer.matplotlib2DViewer 344
10.19Module fipy.viewers.matplotlibViewer.matplotlibSparseMatrixViewer 345
10.20Module fipy.viewers.matplotlibViewer.matplotlibVectorViewer 348
10.21Module fipy.viewers.matplotlibViewer.matplotlibViewer 350
10.22Module fipy.viewers.matplotlibViewer.test 350
10.23Package fipy.viewers.mayaviViewer 350
10.24Module fipy.viewers.mayaviViewer.mayaviViewer 352
10.25Module fipy.viewers.mayaviViewer.test 355
10.26Module fipy.viewers.multiViewer 355
10.27Module fipy.viewers.test 356
10.28Module fipy.viewers.testinteractive 356
10.29Module fipy.viewers.tsvViewer 357
10.30Module fipy.viewers.viewer 359

Bibliography **361**

Index **363**

How To Read This Manual

This chapter will illustrate the conventions used throughout this manual.

Package fipy.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.

Module fipy.package.base

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

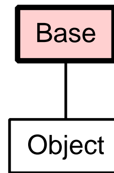
```
import fipy.package.base
```

in which case you refer to it by its full name of `fipy.package.base`, or:

```
from fipy.package import base
```

in which case you can refer simply to `base`.

Class Base



Known Subclasses: fipy.package.object.Object

With very few exceptions, the name of a class will be the capitalized form of the module it resides in. Depending on how you imported the module above, you will refer to either `fipy.package.object.Object` or `object.Object`. Alternatively, you can use:

```
from fipy.package.object import Object
```

and then refer simply to `Object`. There is a shorthand notation:

```
from fipy import Object
```

but it is still experimental and does not work for all of the objects in FiPy.

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.

Methods

method1(*self*)

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!

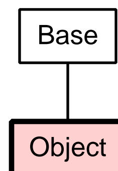
self is supplied automatically by the Python interpreter to all methods. You don't need to (and should not) specify it yourself.

method2(*self*)

This is another thing that you can instruct any object that derives from `Base` to do.

Module fipy.package.object

Class Object



Methods

__init__(*self*, *arg1*, *arg2*=None, *arg3*='string')

This method, like all those whose names begin and end with "__" are special. You won't ever need to call these methods directly, but Python will invoke them for you under certain circumstances, which are described in the Python Reference Manual: Special Method Names [1, §3.3].

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(*self*)

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

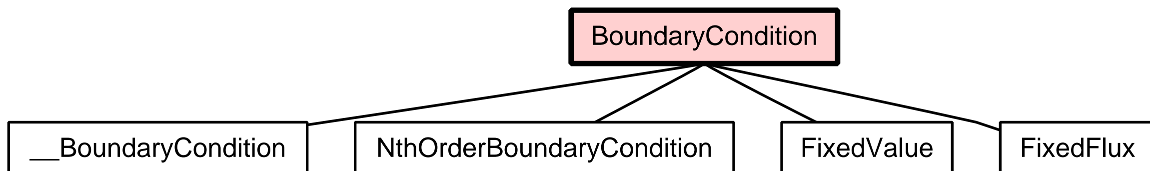
Overrides: fipy.package.base.Base.method2()

Inherited from fipy.package.base.Base: method1()

Package fipy.boundaryConditions

2.1 Module fipy.boundaryConditions.boundaryCondition

Class BoundaryCondition



Known Subclasses: fipy.boundaryConditions.boundaryCondition.__BoundaryCondition, fipy.boundaryConditions.nthOrderBoundaryCondition.NthOrderBoundaryCondition, fipy.boundaryConditions.fixedValue.FixedValue, fipy.boundaryConditions.fixedFlux.FixedFlux

Generic boundary condition base class.

Attention!

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

Methods

__init__(*self*, *faces*, *value*)

The `BoundaryCondition` class should raise an error when invoked with internal faces. Don't use the `BoundaryCondition` class in this manner. This is merely a test.

```
>>> from fipy.meshes.grid1D import Grid1D
>>> mesh = Grid1D(nx = 2)
>>> bc = __BoundaryCondition(mesh.getInteriorFaces(), 0)
Traceback (most recent call last):
 ...
IndexError: Face list has interior faces
```

Parameters

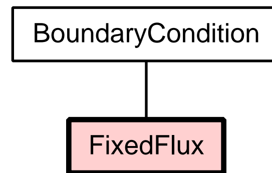
faces: A `list` or `tuple` of `Face` objects to which this condition applies.

value: The value to impose.

`__repr__`(*self*)

2.2 Module fipy.boundaryConditions.fixedFlux

Class FixedFlux



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.

Methods

`__init__`(*self*, *faces*, *value*)

Creates a `FixedFlux` object.

Parameters

faces: A `list` or `tuple` of `Face` objects to which this condition applies.

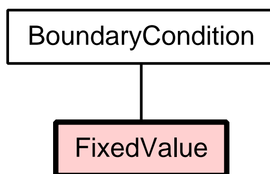
value: The value to impose.

Overrides: fipy.boundaryConditions.boundaryCondition.BoundaryCondition.__init__()

Inherited from fipy.boundaryConditions.boundaryCondition.BoundaryCondition: __repr__()

2.3 Module fipy.boundaryConditions.fixedValue

Class FixedValue



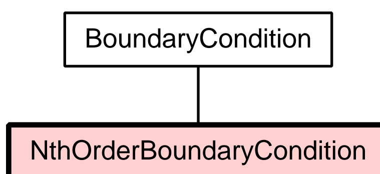
The `FixedValue` boundary condition adds a contribution, equivalent to a fixed value (Dirichlet condition), to the equation's RHS vector and coefficient matrix. The contributions are given by $-\texttt{value} * G_{\text{face}}$ for the RHS vector and G_{face} for the coefficient matrix. The parameter 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.

Methods

Inherited from fipy.boundaryConditions.boundaryCondition.BoundaryCondition: __init__(), __repr__()

2.4 Module fipy.boundaryConditions.nthOrderBoundaryCondition

Class NthOrderBoundaryCondition



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.

Methods

`__init__`(*self*, *faces*, *value*, *order*)

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.

Overrides: fipy.boundaryConditions.boundaryCondition.BoundaryCondition.__init__()

Inherited from fipy.boundaryConditions.boundaryCondition.BoundaryCondition: __repr__()

2.5 Module fipy.boundaryConditions.test

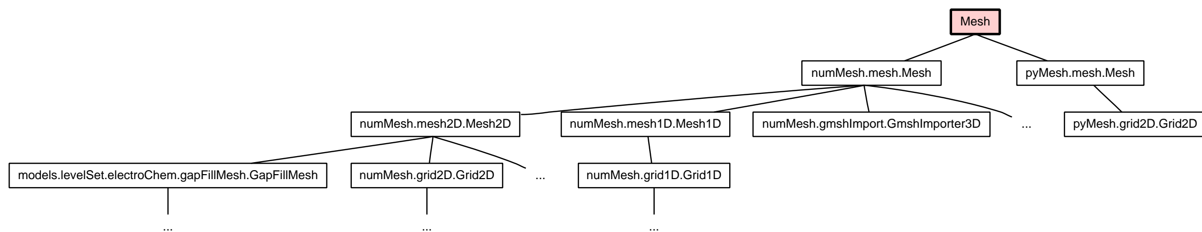
Test numeric implementation of the mesh

Package fipy.meshes

3.1 Package fipy.meshes.common

3.2 Module fipy.meshes.common.mesh

Class Mesh



Known Subclasses: fipy.meshes.numMesh.mesh.Mesh, fipy.meshes.pyMesh.mesh.Mesh

Generic mesh class defining implementation-agnostic behavior.

Make changes to mesh here first, then implement specific implementations in `pyMesh` and `numMesh`.

Meshes contain cells, faces, and vertices.

Methods

`__init__`(*self*)

`__add__`(*self*, *other*)

Either translate a `Mesh` or concatenate two `Mesh` objects.

```
>>> from fipy.meshes.grid2D import Grid2D
>>> baseMesh = Grid2D(dx = 1.0, dy = 1.0, nx = 2, ny = 2)
>>> print baseMesh.getCellCenters()
[[ 0.5 1.5 0.5 1.5]
 [ 0.5 0.5 1.5 1.5]]
```

If a vector is added to a Mesh, a translated Mesh is returned

```
>>> translatedMesh = baseMesh + ((5,), (10,))
>>> print translatedMesh.getCellCenters()
[[ 5.5 6.5 5.5 6.5]
 [ 10.5 10.5 11.5 11.5]]
```

If a Mesh is added to a Mesh, a concatenation of the two Mesh objects is returned

```
>>> addedMesh = baseMesh + (baseMesh + ((2,), (0,)))
>>> print addedMesh.getCellCenters()
[[ 0.5 1.5 0.5 1.5 2.5 3.5 2.5 3.5]
 [ 0.5 0.5 1.5 1.5 0.5 0.5 1.5 1.5]]
```

The two Mesh objects must be properly aligned in order to concatenate them

```
>>> addedMesh = baseMesh + (baseMesh + ((3,), (0,)))
Traceback (most recent call last):
...
MeshAdditionError: Vertices are not aligned

>>> addedMesh = baseMesh + (baseMesh + ((2,), (2,)))
Traceback (most recent call last):
...
MeshAdditionError: Faces are not aligned
```

No provision is made to avoid or consolidate overlapping Mesh objects

```
>>> addedMesh = baseMesh + (baseMesh + ((1,), (0,)))
>>> print addedMesh.getCellCenters()
[[ 0.5 1.5 0.5 1.5 1.5 2.5 1.5 2.5]
 [ 0.5 0.5 1.5 1.5 0.5 0.5 1.5 1.5]]
```

Different Mesh classes can be concatenated

```
>>> from fipy.meshes.tri2D import Tri2D
>>> triMesh = Tri2D(dx = 1.0, dy = 1.0, nx = 2, ny = 1)
>>> triMesh = triMesh + ((2,), (0,))
>>> triAddedMesh = baseMesh + triMesh
>>> print triAddedMesh.getCellCenters()
[[ 0.5 1.5 0.5 1.5 2.83333333 3.83333333
 2.5 3.5 2.16666667 3.16666667 2.5 3.5 ]
 [ 0.5 0.5 1.5 1.5 0.5 0.5
 0.83333333 0.83333333 0.5 0.5 0.16666667 0.16666667]]
```

but their faces must still align properly

```
>>> triMesh = Tri2D(dx = 1.0, dy = 2.0, nx = 2, ny = 1)
>>> triMesh = triMesh + ((2,), (0,))
>>> triAddedMesh = baseMesh + triMesh
Traceback (most recent call last):
...
MeshAdditionError: Faces are not aligned
```

`Mesh` concatenation is not limited to 2D meshes

```
>>> from fipy.meshes.grid3D import Grid3D
>>> threeDBaseMesh = Grid3D(dx = 1.0, dy = 1.0, dz = 1.0,
...  nx = 2, ny = 2, nz = 2)
>>> threeDSecondMesh = Grid3D(dx = 1.0, dy = 1.0, dz = 1.0,
...  nx = 1, ny = 1, nz = 1)
>>> threeDAddedMesh = threeDBaseMesh + (threeDSecondMesh + ((2,), (0,), (0,)))
>>> print threeDAddedMesh.getCellCenters()
[[ 0.5 1.5 0.5 1.5 0.5 1.5 0.5 1.5 2.5]
 [ 0.5 0.5 1.5 1.5 0.5 0.5 1.5 1.5 0.5]
 [ 0.5 0.5 0.5 0.5 1.5 1.5 1.5 1.5 0.5]]
```

but the different `Mesh` objects must, of course, have the same dimensionality.

```
>>> InvalidMesh = threeDBaseMesh + baseMesh
Traceback (most recent call last):
...
MeshAdditionError: Dimensions do not match
```

`__mul__`(*self*, *factor*)

Dilate a `Mesh` by `factor`.

```
>>> from fipy.meshes.grid2D import Grid2D
>>> baseMesh = Grid2D(dx = 1.0, dy = 1.0, nx = 2, ny = 2)
>>> print baseMesh.getCellCenters()
[[ 0.5 1.5 0.5 1.5]
 [ 0.5 0.5 1.5 1.5]]
```

The `factor` can be a scalar

```
>>> dilatedMesh = baseMesh * 3
>>> print dilatedMesh.getCellCenters()
[[ 1.5 4.5 1.5 4.5]
 [ 1.5 1.5 4.5 4.5]]
```

or a vector

```
>>> dilatedMesh = baseMesh * ((3,), (2,))
>>> print dilatedMesh.getCellCenters()
[[ 1.5 4.5 1.5 4.5]
 [ 1. 1. 3. 3. ]]
```

but the vector must have the same dimensionality as the `Mesh`

```
>>> dilatedMesh = baseMesh * ((3,), (2,), (1,))
Traceback (most recent call last):
...
ValueError: shape mismatch: objects cannot be broadcast to a single shape
```

`__repr__`(*self*)

`getExteriorFaces`(*self*)

`getInteriorFaces`(*self*)

`getNumberOfCells`(*self*)

`getDim`(*self*)

`getCells`(*self*, *ids*=None)

Return `Cell` objects of `Mesh`.

```
>>> from fipy import Grid2D
>>> m = Grid2D(nx=2, ny=2)
>>> x, y = m.getCellCenters()
>>> print m.getCells()[x < 1]
[Cell(mesh=UniformGrid2D(dx=1.0, dy=1.0, nx=2, ny=2), id=0)
 Cell(mesh=UniformGrid2D(dx=1.0, dy=1.0, nx=2, ny=2), id=2)]
>>> print m.getCells(ids=(0, 2))
[Cell(mesh=UniformGrid2D(dx=1.0, dy=1.0, nx=2, ny=2), id=0)
 Cell(mesh=UniformGrid2D(dx=1.0, dy=1.0, nx=2, ny=2), id=2)]
```

getFaces(*self*)

Return `Face` objects of `Mesh`.

```
>>> from fipy import Grid2D
>>> m = Grid2D(nx=2, ny=2)
>>> x, y = m.getFaceCenters()
>>> print m.getFaces()[x < 1]
[0 2 4 6 9]
```

getFacesLeft(*self*)

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.)
>>> numerix.allequal((21, 25),
...  numerix.nonzero(mesh.getFacesLeft())[0])
1
>>> mesh = Grid2D(nx = 3, ny = 2, dx = 0.5, dy = 2.)
>>> numerix.allequal((9, 13),
...  numerix.nonzero(mesh.getFacesLeft())[0])
1
```

getFacesRight(*self*)

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.)
>>> numerix.allequal((24, 28),
...  numerix.nonzero(mesh.getFacesRight())[0])
1
>>> mesh = Grid2D(nx = 3, ny = 2, dx = 0.5, dy = 2.)
```

```
>>> numerix.allequal((12, 16),
...  numerix.nonzero(mesh.getFacesRight())[0])
1
```

getFacesBottom(*self*)

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.)
>>> numerix.allequal((12, 13, 14),
...  numerix.nonzero(mesh.getFacesBottom())[0])
1
>>> x, y, z = mesh.getFaceCenters()
>>> numerix.allequal((12, 13),
...  numerix.nonzero(mesh.getFacesBottom() & (x < 1))[0])
1
```

getFacesDown(*self*)

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.)
>>> numerix.allequal((12, 13, 14),
...  numerix.nonzero(mesh.getFacesBottom())[0])
1
>>> x, y, z = mesh.getFaceCenters()
>>> numerix.allequal((12, 13),
...  numerix.nonzero(mesh.getFacesBottom() & (x < 1))[0])
1
```

getFacesTop(*self*)

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.)
>>> numerix.allequal((18, 19, 20),
...  numerix.nonzero(mesh.getFacesTop())[0])
1
>>> mesh = Grid2D(nx = 3, ny = 2, dx = 0.5, dy = 2.)
>>> numerix.allequal((6, 7, 8),
...  numerix.nonzero(mesh.getFacesTop())[0])
1
```

getFacesUp(*self*)

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.)
>>> numerix.allequal((18, 19, 20),
...  numerix.nonzero(mesh.getFacesTop())[0])
1
>>> mesh = Grid2D(nx = 3, ny = 2, dx = 0.5, dy = 2.)
>>> numerix.allequal((6, 7, 8),
...  numerix.nonzero(mesh.getFacesTop())[0])
1
```

getFacesBack(*self*)

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.)
>>> numerix.allequal((6, 7, 8, 9, 10, 11),
...  numerix.nonzero(mesh.getFacesBack())[0])
1
```

getFacesFront(*self*)

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.)
>>> numerix.allequal((0, 1, 2, 3, 4, 5),
...  numerix.nonzero(mesh.getFacesFront())[0])
1
```

getCellVolumes(*self*)

getCellCenters(*self*)

`setScale(self, value=1.0)`

`getNearestCell(self, point)`

Instance Variables

cellToCellIDsFilled	get topology methods
cellAreas	get geometry methods

3.3 Module fipy.meshes.cylindricalGrid1D

Functions

`CylindricalGrid1D(`*dr*`=None, `*nr*`=None, `*dx*`=1.0, `*nx*`=None)`

3.4 Module fipy.meshes.cylindricalGrid2D

Functions

`CylindricalGrid2D(`*dr*=`None`, *dz*=`None`, *nr*=`None`, *nz*=`None`, *dx*=`1.0`, *dy*=`1.0`, *nx*=`None`, *ny*=`None`)

3.5 Module fipy.meshes.gmshExport

3.6 Module fipy.meshes.gmshImport

3.7 Module fipy.meshes.grid1D

Functions

`Grid1D(`*dx*`=1.0,` *nx*`=None)`

3.8 Module fipy.meshes.grid2D

Functions

`Grid2D(`*dx*`=1.0,` *dy*`=1.0,` *nx*`=None,` *ny*`=None)`

3.9 Module fipy.meshes.grid3D

Functions

`Grid3D(`*dx*=`1.0`, *dy*=`1.0`, *dz*=`1.0`, *nx*=`None`, *ny*=`None`, *nz*=`None`)

3.10 Package fipy.meshes.numMesh

Description of mesh geometries

3.11 Module fipy.meshes.numMesh.cell

Class Cell

Methods

`__init__`(*self*, *mesh*, *id*)

`getID`(*self*)

`getCenter`(*self*)

`__cmp__`(*self*, *cell*)

`getMesh`(*self*)

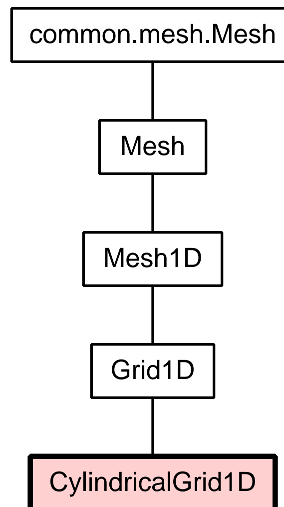
`getNormal`(*self*, *index*)

`__repr__`(*self*)

3.12 Module fipy.meshes.numMesh.cylindricalGrid1D

1D Mesh

Class CylindricalGrid1D



Creates a 1D cyliondrical grid mesh.

```
>>> mesh = CylindricalGrid1D(nx = 3)
>>> print mesh.getCellCenters()
[[ 0.5  1.5  2.5]]

>>> mesh = CylindricalGrid1D(dx = (1, 2, 3))
>>> print mesh.getCellCenters()
[[ 0.5  2.   4.5]]

>>> mesh = CylindricalGrid1D(nx = 2, dx = (1, 2, 3))
Traceback (most recent call last):
...
IndexError: nx != len(dx)
```

Methods

`__init__`(*self*, *dx*=1.0, *nx*=None)

faceVertexIds and cellFacesIds must be padded with minus ones.

Overrides: fipy.meshes.common.mesh.Mesh.__init__()

Inherited from fipy.meshes.numMesh.grid1D.Grid1D: __getstate__(), __repr__(), __setstate__(), getDim(), getPhysicalShape(), getScale(), getShape()

Inherited from fipy.meshes.numMesh.mesh1D.Mesh1D: __mul__()

Inherited from fipy.meshes.numMesh.mesh.Mesh: __add__(), __radd__(), __rmul__(), getExteriorFaces(), getFaceCellIDs(), getFaceCenters(), getInteriorFaces(), getVertexCoords()

Inherited from fipy.meshes.common.mesh.Mesh: getCellCenters(), getCellVolumes(), getCells(), getFaces(), getFacesBack(), getFacesBottom(), getFacesDown(), getFacesFront(), getFacesLeft(), getFacesRight(), getFacesTop(), getFacesUp(), getNearestCell(), getNumberOfCells(), setScale()

Instance Variables

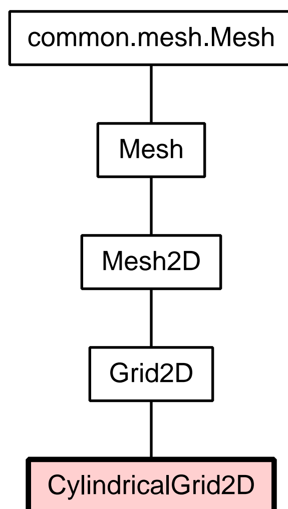
Inherited from fipy.meshes.numMesh.mesh.Mesh: cellNormals

Inherited from fipy.meshes.common.mesh.Mesh: cellAreas, cellToCellIDsFilled

3.13 Module fipy.meshes.numMesh.cylindricalGrid2D

2D rectangular Mesh

Class CylindricalGrid2D



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

Methods

`__init__`(*self*, *dx*=1.0, *dy*=1.0, *nx*=None, *ny*=None, *origin*=((0.0), (0.0)))

faceVertexIds and cellFacesIds must be padded with minus ones.

Overrides: fipy.meshes.common.mesh.Mesh.__init__()

`getCellVolumes`(*self*)

Overrides: fipy.meshes.common.mesh.Mesh.getCellVolumes()

`__mul__`(*self*, *factor*)

Dilate a `Mesh` by `factor`.

```
>>> from fipy.meshes.grid2D import Grid2D
>>> baseMesh = Grid2D(dx = 1.0, dy = 1.0, nx = 2, ny = 2)
>>> print baseMesh.getCellCenters()
[[ 0.5 1.5 0.5 1.5]
 [ 0.5 0.5 1.5 1.5]]
```

The `factor` can be a scalar

```
>>> dilatedMesh = baseMesh * 3
>>> print dilatedMesh.getCellCenters()
[[ 1.5 4.5 1.5 4.5]
 [ 1.5 1.5 4.5 4.5]]
```

or a vector

```
>>> dilatedMesh = baseMesh * ((3,), (2,))
>>> print dilatedMesh.getCellCenters()
[[ 1.5 4.5 1.5 4.5]
 [ 1. 1. 3. 3. ]]
```

but the vector must have the same dimensionality as the `Mesh`

```
>>> dilatedMesh = baseMesh * ((3,), (2,), (1,))
Traceback (most recent call last):
...
ValueError: shape mismatch: objects cannot be broadcast to a single shape
```

Overrides: fipy.meshes.common.mesh.Mesh.__mul__() *(inherited documentation)*

getVertexCoords(*self*)

Overrides: fipy.meshes.numMesh.mesh.Mesh.getVertexCoords()

getCellCenters(*self*)

Overrides: fipy.meshes.common.mesh.Mesh.getCellCenters()

getFaceCenters(*self*)

Overrides: fipy.meshes.numMesh.mesh.Mesh.getFaceCenters()

Inherited from fipy.meshes.numMesh.grid2D.Grid2D: __getstate__(), __repr__(), __setstate__(), getPhysicalShape(), getScale(), getShape()

Inherited from fipy.meshes.numMesh.mesh2D.Mesh2D: extrude()

Inherited from fipy.meshes.numMesh.mesh.Mesh: __add__(), __radd__(), __rmul__(), getExteriorFaces(), getFaceCellIDs(), getInteriorFaces()

Inherited from fipy.meshes.common.mesh.Mesh: getCells(), getDim(), getFaces(), getFacesBack(), getFacesBottom(), getFacesDown(), getFacesFront(), getFacesLeft(), getFacesRight(), getFacesTop(), getFacesUp(), getNearestCell(), getNumberOfCells(), setScale()

Instance Variables

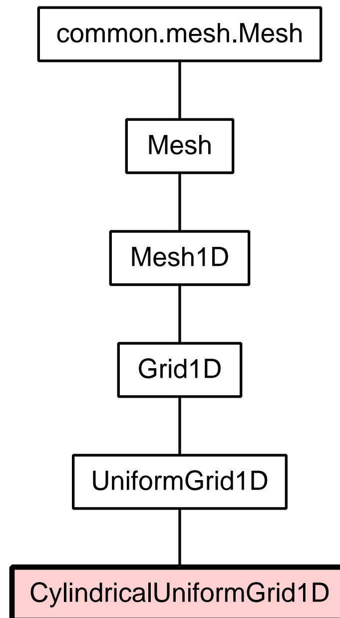
Inherited from fipy.meshes.numMesh.mesh.Mesh: cellNormals

Inherited from fipy.meshes.common.mesh.Mesh: cellAreas, cellToCellIDsFilled

3.14 Module fipy.meshes.numMesh.cylindricalUniformGrid1D

1D Mesh

Class CylindricalUniformGrid1D



Creates a 1D cylindrical grid mesh.

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

Methods

`__init__(self, dx=1.0, nx=1, origin=(0))`

faceVertexIds and cellFacesIds must be padded with minus ones.

Overrides: fipy.meshes.common.mesh.Mesh.__init__()

`getCellVolumes(self)`

Overrides: fipy.meshes.common.mesh.Mesh.getCellVolumes()

Inherited from fipy.meshes.numMesh.uniformGrid1D.UniformGrid1D: __mul__(), getCellCenters(), getFaceCellIDs(), getFaceCenters(), getInteriorFaces(), getVertexCoords()

Inherited from fipy.meshes.numMesh.grid1D.Grid1D: __getstate__(), __repr__(), __setstate__(), getDim(), getPhysicalShape(), getScale(), getShape()

Inherited from fipy.meshes.numMesh.mesh.Mesh: __add__(), __radd__(), __rmul__(), getExteriorFaces()

Inherited from fipy.meshes.common.mesh.Mesh: getCells(), getFaces(), getFacesBack(), getFacesBottom(), getFacesDown(), getFacesFront(), getFacesLeft(), getFacesRight(), getFacesTop(), getFacesUp(), getNearestCell(), getNumberOfCells(), setScale()

Instance Variables

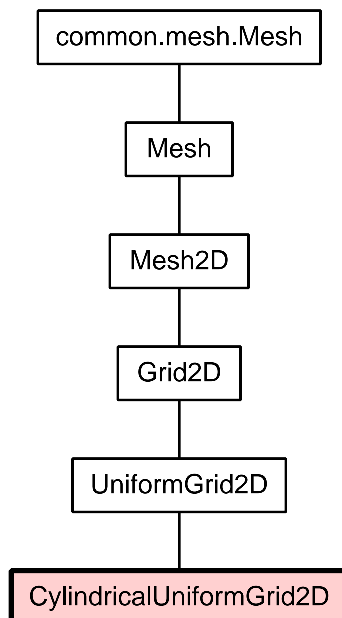
Inherited from fipy.meshes.numMesh.mesh.Mesh: cellNormals

Inherited from fipy.meshes.common.mesh.Mesh: cellAreas, cellToCellIDsFilled

3.15 Module fipy.meshes.numMesh.cylindricalUniformGrid2D

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

Class CylindricalUniformGrid2D



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

Methods

__init__(*self*, *dx*=1.0, *dy*=1.0, *nx*=1, *ny*=1, *origin*=((0), (0)))

faceVertexIds and cellFacesIds must be padded with minus ones.

Overrides: fipy.meshes.common.mesh.Mesh.__init__()

getCellVolumes(*self*)

Overrides: fipy.meshes.common.mesh.Mesh.getCellVolumes()

Inherited from fipy.meshes.numMesh.uniformGrid2D.UniformGrid2D: __mul__(), getCellCenters(), getExteriorFaces(), getFaceCellIDs(), getFaceCenters(), getInteriorFaces(), getVertexCoords()

Inherited from fipy.meshes.numMesh.grid2D.Grid2D: __getstate__(), __repr__(), __setstate__(), getPhysicalShape(), getScale(), getShape()

Inherited from fipy.meshes.numMesh.mesh2D.Mesh2D: extrude()

Inherited from fipy.meshes.numMesh.mesh.Mesh: __add__(), __radd__(), __rmul__()

Inherited from fipy.meshes.common.mesh.Mesh: getCells(), getDim(), getFaces(), getFacesBack(), getFacesBottom(), getFacesDown(), getFacesFront(), getFacesLeft(), getFacesRight(), getFacesTop(), getFacesUp(), getNearestCell(), getNumberOfCells(), setScale()

Instance Variables

Inherited from fipy.meshes.numMesh.mesh.Mesh: cellNormals

Inherited from fipy.meshes.common.mesh.Mesh: cellAreas, cellToCellIDsFilled

3.16 Module fipy.meshes.numMesh.face

Class Face

`Face` within a `Mesh`

`Face` objects are bounded by `Vertex` objects. `Face` objects separate `Cell` objects.

Methods

`__init__`(*self*, *mesh*, *id*)

`Face` is initialized by `Mesh`

Parameters

mesh: the `Mesh` that contains this `Face`

id: a unique identifier

`getMesh`(*self*)

`getID`(*self*)

`getCellID`(*self*, *index*=0)

Return the `id` of the specified `Cell` on one side of this `Face`.

`getCenter`(*self*)

Return the coordinates of the `Face` center.

`getArea`(*self*)

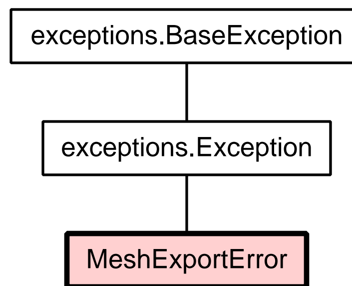
3.17 Module fipy.meshes.numMesh.gmshExport

This module takes a FiPy mesh and creates a mesh file that can be opened in Gmsh.

Functions

`exportAsMesh`(*mesh*, *filename*)

Class MeshExportError



Methods

Inherited from exceptions.Exception: __init__(), __new__()

Inherited from exceptions.BaseException: __delattr__(), __getattribute__(), __getitem__(), __reduce__(), __repr__(), __setattr__(), __setstate__(), __str__()

Properties

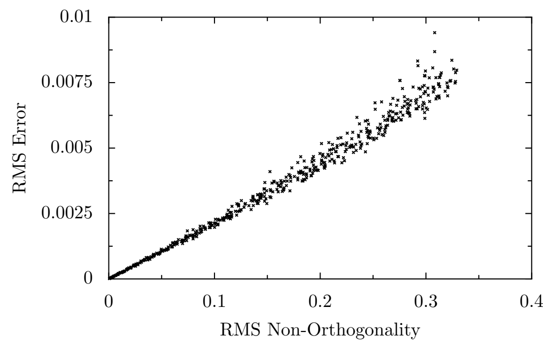
Inherited from exceptions.BaseException: args, message

3.18 Module fipy.meshes.numMesh.gmshImport

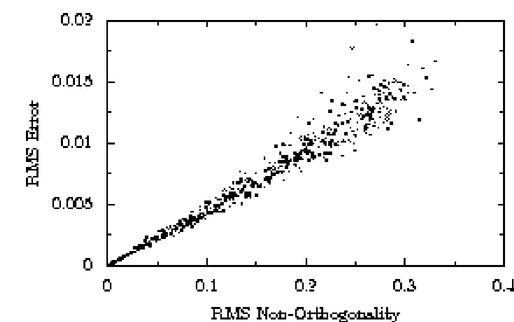
This module takes a Gmsh output file (`.msh`) and converts it into a FiPy mesh. This currently supports triangular and tetrahedral meshes only.

Gmsh generates unstructured meshes, which may contain a significant amount of non-orthogonality and it is very difficult to directly control the amount of non-orthogonality simply by manipulating Gmsh parameters. Therefore, it is necessary to take into account the possibility of errors arising due to the non-orthogonality of the mesh. To test the degree of error, an experiment was conducted using a simple 1D diffusion problem with constant diffusion coefficients and boundary conditions as follows: fixed value of 0 on the left side, fixed value of 1 on the right side, and a fixed flux of 0 on the top and bottom sides. The analytical solution is clearly a uniform gradient going from left to right. this problem was implemented using a Cartesian Grid2D mesh with each interior vertex displaced a short distance in a random direction to create non-orthogonality. Then the root-mean-square error was plotted against the root-mean-square non-orthogonality. The error in each cell was calculated by simply subtracting the analytical solution at each cell center from the calculated value for that cell. The non-orthogonality in each cell is the average, weighted by face area, of the sines of the angles between the face normals and the line segments joining the cells. Thus, the non-orthogonality of a cell can range from 0 (every face is orthogonal to its corresponding cell-to-cell line segment) to 1 (only possible in a degenerate case). This test was run using 500 separate 20x20 meshes and 500 separate 10x10 meshes, each with the interior vertices moved different amounts so as to created different levels of non-orthogonality. The results are shown below.

Results for 20x20 mesh:



Results for 10x10 mesh:



It is clear from the graphs that finer meshes decrease the error due to non-orthogonality, and that even with a reasonably coarse mesh the error is quite low. However, note that this test is only for a simple 1D diffusion

problem with a constant diffusion coefficient, and it is unknown whether the results will be significantly different with more complicated problems.

Test cases:

```
>>> newmesh = GmshImporter3D('fipy/meshes/numMesh/testgmsh.msh')
>>> print newmesh.getVertexCoords()
[[ 0.   0.5  1.   0.5  0.5]
 [ 0.   0.5  0.   1.   0.5]
 [ 0.   1.   0.   0.   0.5]]

>>> print newmesh._getFaceVertexIDs()
[[2 4 4 4 3 4 4 3 4 3]
 [1 1 2 2 1 3 3 2 3 2]
 [0 0 0 1 0 0 1 0 2 1]]

>>> print newmesh._getCellFaceIDs()
[[0 4 7 9]
 [1 1 2 3]
 [2 5 5 6]
 [3 6 8 8]]

>>> mesh = GmshImporter2DIn3DSpace('fipy/meshes/numMesh/GmshTest2D.msh')
>>> print mesh.getVertexCoords()
[[ 0.   1.   0.5  0.   1.   0.5  0.   1. ]
 [ 0.   0.   0.5  1.   1.   1.5  2.   2. ]
 [ 0.   0.   0.   0.   0.   0.   0.   0. ]]

>>> mesh = GmshImporter2D('fipy/meshes/numMesh/GmshTest2D.msh')
>>> print mesh.getVertexCoords()
[[ 0.   1.   0.5  0.   1.   0.5  0.   1. ]
 [ 0.   0.   0.5  1.   1.   1.5  2.   2. ]]

>>> print mesh._getFaceVertexIDs()
[[2 0 1 0 3 1 4 4 3 5 3 6 5 7 7]
 [0 1 2 3 2 4 2 3 5 4 6 5 7 4 6]]

>>> print (mesh._getCellFaceIDs() == [[0, 0, 2, 7, 7, 8, 12, 14],
...                                    [1, 3, 5, 4, 8, 10, 13, 11],
...                                    [2, 4, 6, 6, 9, 11, 9, 12]]).flatten().all()
True
```

The following test case is to test the handedness of the mesh to check it does not return negative volumes. Firstly we set up a list with tuples of strings to be read by gmsh. The list provide instuctions to gmsh to form a circular mesh.

```
>>> cellSize = 0.7
>>> radius = 1.
```

```
>>> lines = ['cellSize = ' + str(cellSize) + ';\n',
...          'radius = ' + str(radius) + ';\n',
...          'Point(1) = {0, 0, 0, cellSize};\n',
...          'Point(2) = {-radius, 0, 0, cellSize};\n',
...          'Point(3) = {0, radius, 0, cellSize};\n',
...          'Point(4) = {radius, 0, 0, cellSize};\n',
...          'Point(5) = {0, -radius, 0, cellSize};\n',
...          'Circle(6) = {2, 1, 3};\n',
...          'Circle(7) = {3, 1, 4};\n',
...          'Circle(8) = {4, 1, 5};\n',
...          'Circle(9) = {5, 1, 2};\n',
...          'Line Loop(10) = {6, 7, 8, 9} ;\n',
...          'Plane Surface(11) = {10};\n']
```

Check that the sign of the mesh volumes is correct

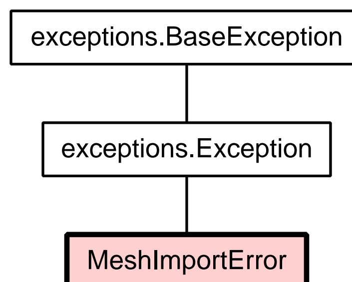
```
>>> mesh = GmshImporter2D(lines)
>>> print mesh.getCellVolumes()[0] > 0
1
```

Reverse the handedness of the mesh and check the sign

```
>>> lines[7:12] = ['Circle(6) = {3, 1, 2};\n',
...                'Circle(7) = {4, 1, 3};\n',
...                'Circle(8) = {5, 1, 4};\n',
...                'Circle(9) = {2, 1, 5};\n',
...                'Line Loop(10) = {9, 8, 7, 6};\n',]
```

```
>>> mesh = GmshImporter2D(lines)
>>> print mesh.getCellVolumes()[0] > 0
1
```

Class MeshImportError



Methods

Inherited from exceptions.Exception: __init__(), __new__()

Inherited from exceptions.BaseException: __delattr__(), __getattribute__(), __getitem__(), __reduce__(), __repr__(), __setattr__(), __setstate__(), __str__()

Properties

Inherited from exceptions.BaseException: args, message

Class MshFile

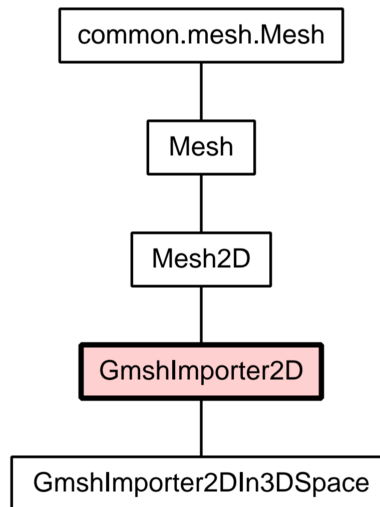
Methods

`__init__`(*self*, *arg*)

`getFilename`(*self*)

`remove`(*self*)

Class GmshImporter2D



Known Subclasses: fipy.meshes.numMesh.gmshImport.GmshImporter2DIn3DSpace

Methods

`__init__`(*self*, *arg*, *coordDimensions*=2)

---

faceVertexIds and cellFacesIds must be padded with minus ones.

Overrides: fipy.meshes.common.mesh.Mesh.__init__()

`getCellVolumes`(*self*)

Overrides: fipy.meshes.common.mesh.Mesh.getCellVolumes()

Inherited from fipy.meshes.numMesh.mesh2D.Mesh2D: __mul__(), extrude()

Inherited from fipy.meshes.numMesh.mesh.Mesh: __add__(), __getstate__(), __radd__(), __rmul__(), __setstate__(), getExteriorFaces(), getFaceCellIDs(), getFaceCenters(), getInteriorFaces(), getVertexCoords()

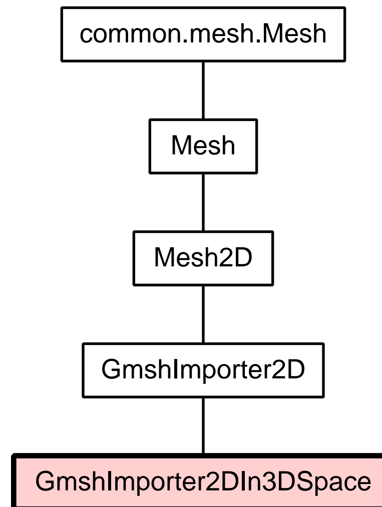
Inherited from fipy.meshes.common.mesh.Mesh: __repr__(), getCellCenters(), getCells(), getDim(), getFaces(), getFacesBack(), getFacesBottom(), getFacesDown(), getFacesFront(), getFacesLeft(), getFacesRight(), getFacesTop(), getFacesUp(), getNearestCell(), getNumberOfCells(), setScale()

Instance Variables

Inherited from fipy.meshes.numMesh.mesh.Mesh: cellNormals

Inherited from fipy.meshes.common.mesh.Mesh: cellAreas, cellToCellIDsFilled

Class GmshImporter2DIn3DSpace



Methods

`__init__`(*self*, *arg*)

faceVertexIds and cellFacesIds must be padded with minus ones.

Overrides: fipy.meshes.common.mesh.Mesh.__init__()

Inherited from fipy.meshes.numMesh.gmshImport.GmshImporter2D: getCellVolumes()

Inherited from fipy.meshes.numMesh.mesh2D.Mesh2D: __mul__(), extrude()

Inherited from fipy.meshes.numMesh.mesh.Mesh: __add__(), __getstate__(), __radd__(), __rmul__(), __setstate__(), getExteriorFaces(), getFaceCellIDs(), getFaceCenters(), getInteriorFaces(), getVertexCoords()

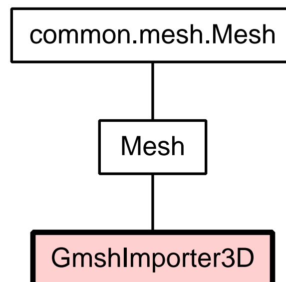
Inherited from fipy.meshes.common.mesh.Mesh: __repr__(), getCellCenters(), getCells(), getDim(), getFaces(), getFacesBack(), getFacesBottom(), getFacesDown(), getFacesFront(), getFacesLeft(), getFacesRight(), getFacesTop(), getFacesUp(), getNearestCell(), getNumberOfCells(), setScale()

Instance Variables

Inherited from fipy.meshes.numMesh.mesh.Mesh: cellNormals

Inherited from fipy.meshes.common.mesh.Mesh: cellAreas, cellToCellIDsFilled

Class GmshImporter3D




```
>>> mesh = GmshImporter3D('fipy/meshes/numMesh/testgmsh.msh')
```

Methods

`__init__`(*self*, *arg*)

faceVertexIds and cellFacesIds must be padded with minus ones.

Overrides: fipy.meshes.common.mesh.Mesh.__init__()

`getCellVolumes`(*self*)

Overrides: fipy.meshes.common.mesh.Mesh.getCellVolumes()

Inherited from fipy.meshes.numMesh.mesh.Mesh: __add__(), __getstate__(), __mul__(), __radd__(), __rmul__(), __setstate__(), getExteriorFaces(), getFaceCellIDs(), getFaceCenters(), getInteriorFaces(), getVertexCoords()

Inherited from fipy.meshes.common.mesh.Mesh: __repr__(), getCellCenters(), getCells(), getDim(), getFaces(), getFacesBack(), getFacesBottom(), getFacesDown(), getFacesFront(), getFacesLeft(), getFacesRight(), getFacesTop(), getFacesUp(), getNearestCell(), getNumberOfCells(), setScale()

Instance Variables

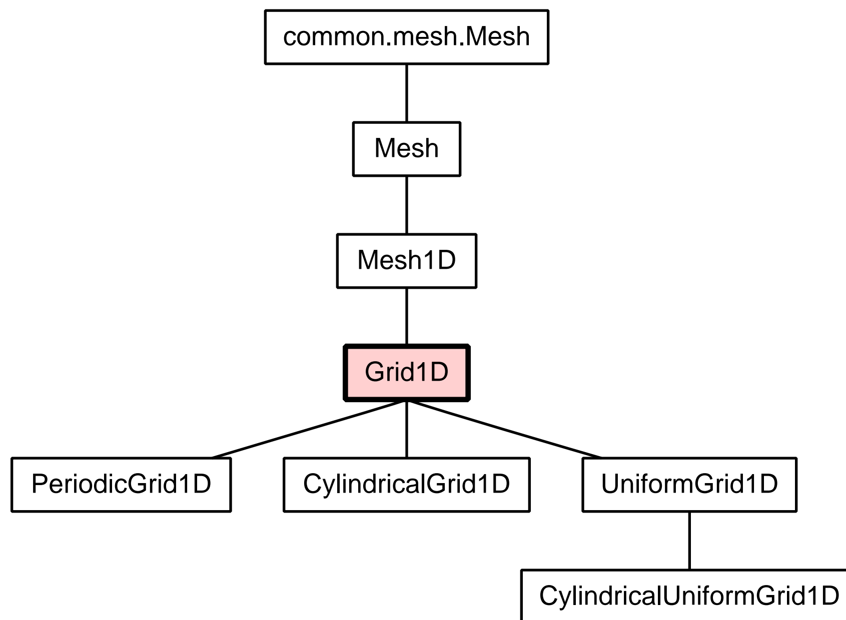
Inherited from fipy.meshes.numMesh.mesh.Mesh: cellNormals

Inherited from fipy.meshes.common.mesh.Mesh: cellAreas, cellToCellIDsFilled

3.19 Module fipy.meshes.numMesh.grid1D

1D Mesh

Class Grid1D



Known Subclasses: fipy.meshes.numMesh.periodicGrid1D.PeriodicGrid1D, fipy.meshes.numMesh.cylindricalGrid1D.CylindricalGrid1D, fipy.meshes.numMesh.uniformGrid1D.UniformGrid1D

Creates a 1D grid mesh.

```
>>> mesh = Grid1D(nx = 3)
>>> print mesh.getCellCenters()
[[ 0.5  1.5  2.5]]

>>> mesh = Grid1D(dx = (1, 2, 3))
>>> print mesh.getCellCenters()
[[ 0.5  2.   4.5]]
```

```
>>> mesh = Grid1D(nx = 2, dx = (1, 2, 3))
Traceback (most recent call last):
...
IndexError: nx != len(dx)
```

Methods

`__init__`(*self*, *dx*=1.0, *nx*=None)

faceVertexIds and cellFacesIds must be padded with minus ones.

Overrides: fipy.meshes.common.mesh.Mesh.__init__()

`__repr__`(*self*)

Overrides: fipy.meshes.common.mesh.Mesh.__repr__()

`getDim`(*self*)

Overrides: fipy.meshes.common.mesh.Mesh.getDim()

`getScale`(*self*)

`getPhysicalShape`(*self*)

Return physical dimensions of Grid1D.

`getShape`(*self*)

`__getstate__`(*self*)

Overrides: fipy.meshes.numMesh.mesh.Mesh.__getstate__()

__setstate__(*self*, *dict*)

Overrides: fipy.meshes.numMesh.mesh.Mesh.__setstate__()

Inherited from fipy.meshes.numMesh.mesh1D.Mesh1D: __mul__()

Inherited from fipy.meshes.numMesh.mesh.Mesh: __add__(), __radd__(), __rmul__(), getExteriorFaces(), getFaceCellIDs(), getFaceCenters(), getInteriorFaces(), getVertexCoords()

Inherited from fipy.meshes.common.mesh.Mesh: getCellCenters(), getCellVolumes(), getCells(), getFaces(), getFacesBack(), getFacesBottom(), getFacesDown(), getFacesFront(), getFacesLeft(), getFacesRight(), getFacesTop(), getFacesUp(), getNearestCell(), getNumberOfCells(), setScale()

Instance Variables

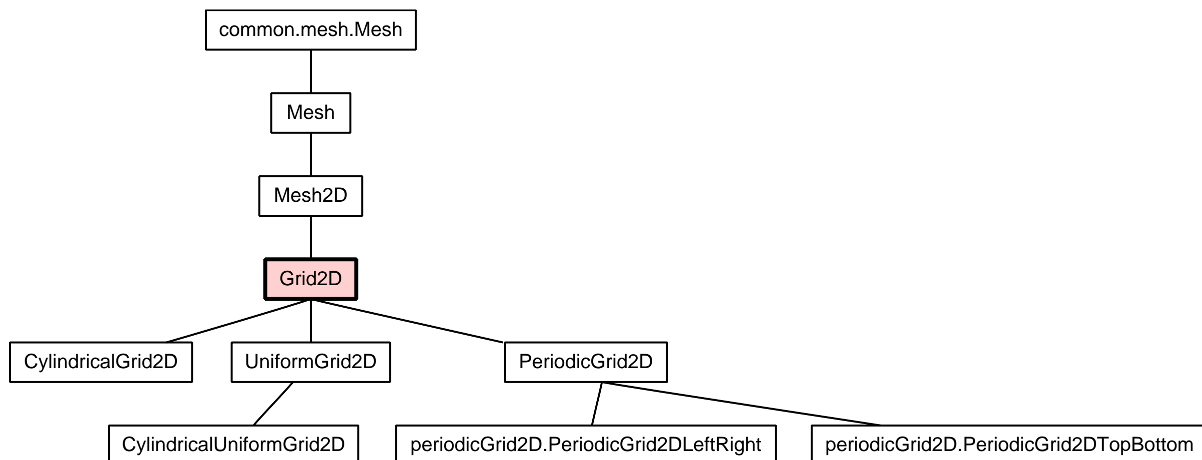
Inherited from fipy.meshes.numMesh.mesh.Mesh: cellNormals

Inherited from fipy.meshes.common.mesh.Mesh: cellAreas, cellToCellIDsFilled

3.20 Module fipy.meshes.numMesh.grid2D

2D rectangular Mesh

Class Grid2D



Known Subclasses: fipy.meshes.numMesh.cylindricalGrid2D.CylindricalGrid2D, fipy.meshes.numMesh.uniformGrid2D.UniformGrid2D, fipy.meshes.numMesh.periodicGrid2D.PeriodicGrid2D

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

Methods

`__init__`(*self*, *dx*=`1.0`, *dy*=`1.0`, *nx*=`None`, *ny*=`None`)

faceVertexIds and cellFacesIds must be padded with minus ones.

Overrides: fipy.meshes.common.mesh.Mesh.__init__()

`__repr__`(*self*)

Overrides: fipy.meshes.common.mesh.Mesh.__repr__()

`getScale`(*self*)

`getPhysicalShape`(*self*)

Return physical dimensions of Grid2D.

`getShape`(*self*)

`__getstate__`(*self*)

Used internally to collect the necessary information to `pickle` the `Grid2D` to persistent storage.

Overrides: fipy.meshes.numMesh.mesh.Mesh.__getstate__()

`__setstate__`(*self*, *dict*)

Used internally to create a new `Grid2D` from `pickled` persistent storage.

Overrides: fipy.meshes.numMesh.mesh.Mesh.__setstate__()

Inherited from fipy.meshes.numMesh.mesh2D.Mesh2D: __mul__(), extrude()

Inherited from fipy.meshes.numMesh.mesh.Mesh: __add__(), __radd__(), __rmul__(), getExteriorFaces(), getFaceCellIDs(), getFaceCenters(), getInteriorFaces(), getVertexCoords()

Inherited from fipy.meshes.common.mesh.Mesh: getCellCenters(), getCellVolumes(), getCells(), getDim(), getFaces(), getFacesBack(), getFacesBottom(), getFacesDown(), getFacesFront(), getFacesLeft(), getFacesRight(), getFacesTop(), getFacesUp(), getNearestCell(), getNumberOfCells(), setScale()

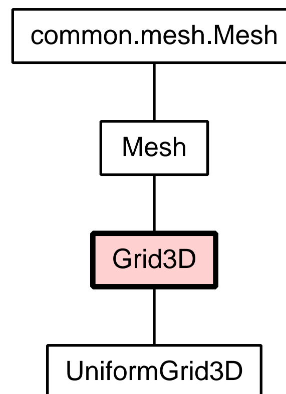
Instance Variables

Inherited from fipy.meshes.numMesh.mesh.Mesh: cellNormals

Inherited from fipy.meshes.common.mesh.Mesh: cellAreas, cellToCellIDsFilled

3.21 Module fipy.meshes.numMesh.grid3D

Class Grid3D



Known Subclasses: fipy.meshes.numMesh.uniformGrid3D.UniformGrid3D

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.

Methods

`__init__`(*self*, *dx*=`1.0`, *dy*=`1.0`, *dz*=`1.0`, *nx*=`None`, *ny*=`None`, *nz*=`None`)

faceVertexIds and cellFacesIds must be padded with minus ones.

Overrides: fipy.meshes.common.mesh.Mesh.__init__()

`__repr__`(*self*)

Overrides: fipy.meshes.common.mesh.Mesh.__repr__()

`getScale`(*self*)

`getPhysicalShape`(*self*)

Return physical dimensions of Grid3D.

`getShape`(*self*)

`__getstate__`(*self*)

Overrides: fipy.meshes.numMesh.mesh.Mesh.__getstate__()

`__setstate__`(*self*, *dict*)

Overrides: fipy.meshes.numMesh.mesh.Mesh.__setstate__()

Inherited from fipy.meshes.numMesh.mesh.Mesh: __add__(), __mul__(), __radd__(), __rmul__(), getExteriorFaces(), getFaceCellIDs(), getFaceCenters(), getInteriorFaces(), getVertexCoords()

Inherited from fipy.meshes.common.mesh.Mesh: getCellCenters(), getCellVolumes(), getCells(), getDim(), getFaces(), getFacesBack(), getFacesBottom(), getFacesDown(), getFacesFront(), getFacesLeft(), getFacesRight(), getFacesTop(), getFacesUp(), getNearestCell(), getNumberOfCells(), setScale()

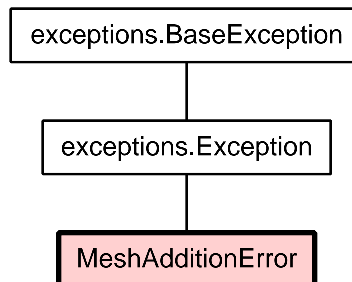
Instance Variables

Inherited from fipy.meshes.numMesh.mesh.Mesh: cellNormals

Inherited from fipy.meshes.common.mesh.Mesh: cellAreas, cellToCellIDsFilled

3.22 Module fipy.meshes.numMesh.mesh

Class MeshAdditionError



Methods

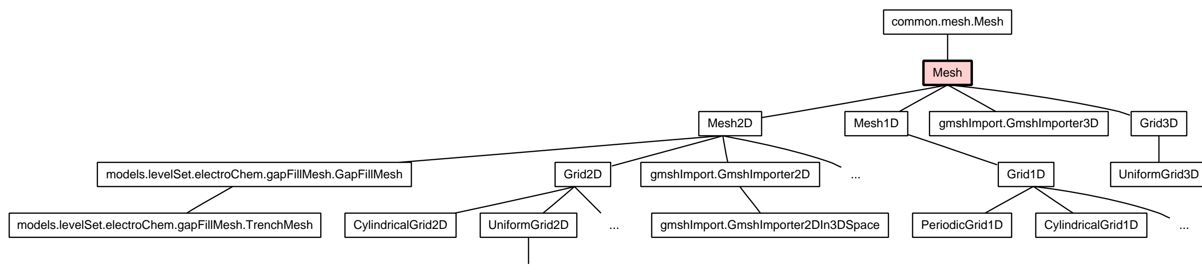
Inherited from exceptions.Exception: __init__(), __new__()

Inherited from exceptions.BaseException: __delattr__(), __getattribute__(), __getitem__(), __reduce__(), __repr__(), __setattr__(), __setstate__(), __str__()

Properties

Inherited from exceptions.BaseException: args, message

Class Mesh



Known Subclasses: fipy.meshes.numMesh.mesh2D.Mesh2D, fipy.meshes.numMesh.mesh1D.Mesh1D, fipy.meshes.numMesh.gmshImport.GmshImporter3D, fipy.meshes.numMesh.grid3D.Grid3D

Generic mesh class using numerix to do the calculations

Meshes contain cells, faces, and vertices.

This is built for a non-mixed element mesh.

Methods

`__init__`(*self*, *vertexCoords*, *faceVertexIDs*, *cellFaceIDs*)

faceVertexIds and cellFacesIds must be padded with minus ones.

Overrides: fipy.meshes.common.mesh.Mesh.__init__()

`__add__`(*self*, *other*)

Either translate a `Mesh` or concatenate two `Mesh` objects.

```
>>> from fipy.meshes.grid2D import Grid2D
>>> baseMesh = Grid2D(dx = 1.0, dy = 1.0, nx = 2, ny = 2)
>>> print baseMesh.getCellCenters()
[[ 0.5 1.5 0.5 1.5]
 [ 0.5 0.5 1.5 1.5]]
```

If a vector is added to a `Mesh`, a translated `Mesh` is returned

```
>>> translatedMesh = baseMesh + ((5,), (10,))
>>> print translatedMesh.getCellCenters()
[[ 5.5 6.5 5.5 6.5]
 [ 10.5 10.5 11.5 11.5]]
```

If a `Mesh` is added to a `Mesh`, a concatenation of the two `Mesh` objects is returned

```
>>> addedMesh = baseMesh + (baseMesh + ((2,), (0,)))
>>> print addedMesh.getCellCenters()
[[ 0.5 1.5 0.5 1.5 2.5 3.5 2.5 3.5]
 [ 0.5 0.5 1.5 1.5 0.5 0.5 1.5 1.5]]
```

The two `Mesh` objects must be properly aligned in order to concatenate them

```
>>> addedMesh = baseMesh + (baseMesh + ((3,), (0,)))
Traceback (most recent call last):
...
MeshAdditionError: Vertices are not aligned
```

```
>>> addedMesh = baseMesh + (baseMesh + ((2,), (2,)))
Traceback (most recent call last):
...
MeshAdditionError: Faces are not aligned
```

No provision is made to avoid or consolidate overlapping `Mesh` objects

```
>>> addedMesh = baseMesh + (baseMesh + ((1,), (0,)))
>>> print addedMesh.getCellCenters()
[[ 0.5 1.5 0.5 1.5 1.5 2.5 1.5 2.5]
 [ 0.5 0.5 1.5 1.5 0.5 0.5 1.5 1.5]]
```

Different `Mesh` classes can be concatenated

```
>>> from fipy.meshes.tri2D import Tri2D
>>> triMesh = Tri2D(dx = 1.0, dy = 1.0, nx = 2, ny = 1)
>>> triMesh = triMesh + ((2,), (0,))
>>> triAddedMesh = baseMesh + triMesh
>>> print triAddedMesh.getCellCenters()
[[ 0.5 1.5 0.5 1.5 2.83333333 3.83333333
 2.5 3.5 2.16666667 3.16666667 2.5 3.5 ]
 [ 0.5 0.5 1.5 1.5 0.5 0.5
 0.83333333 0.83333333 0.5 0.5 0.16666667 0.16666667]]
```

but their faces must still align properly

```
>>> triMesh = Tri2D(dx = 1.0, dy = 2.0, nx = 2, ny = 1)
>>> triMesh = triMesh + ((2,), (0,))
>>> triAddedMesh = baseMesh + triMesh
Traceback (most recent call last):
...
MeshAdditionError: Faces are not aligned
```

`Mesh` concatenation is not limited to 2D meshes

```
>>> from fipy.meshes.grid3D import Grid3D
>>> threeDBaseMesh = Grid3D(dx = 1.0, dy = 1.0, dz = 1.0,
```

```
...  nx = 2, ny = 2, nz = 2)
>>> threeDSecondMesh = Grid3D(dx = 1.0, dy = 1.0, dz = 1.0,
...  nx = 1, ny = 1, nz = 1)
>>> threeDAddedMesh = threeDBaseMesh + (threeDSecondMesh + ((2,), (0,), (0,)))
>>> print threeDAddedMesh.getCellCenters()
[[ 0.5 1.5 0.5 1.5 0.5 1.5 0.5 1.5 2.5]
 [ 0.5 0.5 1.5 1.5 0.5 0.5 1.5 1.5 0.5]
 [ 0.5 0.5 0.5 0.5 1.5 1.5 1.5 1.5 0.5]]
```

but the different `Mesh` objects must, of course, have the same dimensionality.

```
>>> InvalidMesh = threeDBaseMesh + baseMesh
Traceback (most recent call last):
...
MeshAdditionError: Dimensions do not match
```

Overrides: fipy.meshes.common.mesh.Mesh.__add__() *(inherited documentation)*

__radd__(*self*, *other*)

Either translate a `Mesh` or concatenate two `Mesh` objects.

```
>>> from fipy.meshes.grid2D import Grid2D
>>> baseMesh = Grid2D(dx = 1.0, dy = 1.0, nx = 2, ny = 2)
>>> print baseMesh.getCellCenters()
[[ 0.5 1.5 0.5 1.5]
 [ 0.5 0.5 1.5 1.5]]
```

If a vector is added to a `Mesh`, a translated `Mesh` is returned

```
>>> translatedMesh = baseMesh + ((5,), (10,))
>>> print translatedMesh.getCellCenters()
[[ 5.5 6.5 5.5 6.5]
 [ 10.5 10.5 11.5 11.5]]
```

If a `Mesh` is added to a `Mesh`, a concatenation of the two `Mesh` objects is returned

```
>>> addedMesh = baseMesh + (baseMesh + ((2,), (0,)))
>>> print addedMesh.getCellCenters()
[[ 0.5 1.5 0.5 1.5 2.5 3.5 2.5 3.5]
 [ 0.5 0.5 1.5 1.5 0.5 0.5 1.5 1.5]]
```

The two `Mesh` objects must be properly aligned in order to concatenate them

```
>>> addedMesh = baseMesh + (baseMesh + ((3,), (0,)))
Traceback (most recent call last):
...
MeshAdditionError: Vertices are not aligned
```

```
>>> addedMesh = baseMesh + (baseMesh + ((2,), (2,)))
Traceback (most recent call last):
...
MeshAdditionError: Faces are not aligned
```

No provision is made to avoid or consolidate overlapping `Mesh` objects

```
>>> addedMesh = baseMesh + (baseMesh + ((1,), (0,)))
>>> print addedMesh.getCellCenters()
[[ 0.5 1.5 0.5 1.5 1.5 2.5 1.5 2.5]
 [ 0.5 0.5 1.5 1.5 0.5 0.5 1.5 1.5]]
```

Different `Mesh` classes can be concatenated

```
>>> from fipy.meshes.tri2D import Tri2D
>>> triMesh = Tri2D(dx = 1.0, dy = 1.0, nx = 2, ny = 1)
>>> triMesh = triMesh + ((2,), (0,))
>>> triAddedMesh = baseMesh + triMesh
>>> print triAddedMesh.getCellCenters()
[[ 0.5 1.5 0.5 1.5 2.83333333 3.83333333
 2.5 3.5 2.16666667 3.16666667 2.5 3.5 ]
 [ 0.5 0.5 1.5 1.5 0.5 0.5
 0.83333333 0.83333333 0.5 0.5 0.16666667 0.16666667]]
```

but their faces must still align properly

```
>>> triMesh = Tri2D(dx = 1.0, dy = 2.0, nx = 2, ny = 1)
>>> triMesh = triMesh + ((2,), (0,))
>>> triAddedMesh = baseMesh + triMesh
Traceback (most recent call last):
...
MeshAdditionError: Faces are not aligned
```

`Mesh` concatenation is not limited to 2D meshes

```
>>> from fipy.meshes.grid3D import Grid3D
>>> threeDBaseMesh = Grid3D(dx = 1.0, dy = 1.0, dz = 1.0,
...  nx = 2, ny = 2, nz = 2)
>>> threeDSecondMesh = Grid3D(dx = 1.0, dy = 1.0, dz = 1.0,
...  nx = 1, ny = 1, nz = 1)
>>> threeDAddedMesh = threeDBaseMesh + (threeDSecondMesh + ((2,), (0,), (0,)))
>>> print threeDAddedMesh.getCellCenters()
[[ 0.5 1.5 0.5 1.5 0.5 1.5 0.5 1.5 2.5]
 [ 0.5 0.5 1.5 1.5 0.5 0.5 1.5 1.5 0.5]
 [ 0.5 0.5 0.5 0.5 1.5 1.5 1.5 1.5 0.5]]
```

but the different `Mesh` objects must, of course, have the same dimensionality.

```
>>> InvalidMesh = threeDBaseMesh + baseMesh
Traceback (most recent call last):
```

```
...
MeshAdditionError: Dimensions do not match
```

__mul__(*self*, *factor*)

Dilate a `Mesh` by `factor`.

```
>>> from fipy.meshes.grid2D import Grid2D
>>> baseMesh = Grid2D(dx = 1.0, dy = 1.0, nx = 2, ny = 2)
>>> print baseMesh.getCellCenters()
[[ 0.5 1.5 0.5 1.5]
 [ 0.5 0.5 1.5 1.5]]
```

The `factor` can be a scalar

```
>>> dilatedMesh = baseMesh * 3
>>> print dilatedMesh.getCellCenters()
[[ 1.5 4.5 1.5 4.5]
 [ 1.5 1.5 4.5 4.5]]
```

or a vector

```
>>> dilatedMesh = baseMesh * ((3,), (2,))
>>> print dilatedMesh.getCellCenters()
[[ 1.5 4.5 1.5 4.5]
 [ 1.  1.  3.  3. ]]
```

but the vector must have the same dimensionality as the `Mesh`

```
>>> dilatedMesh = baseMesh * ((3,), (2,), (1,))
Traceback (most recent call last):
...
ValueError: shape mismatch: objects cannot be broadcast to a single shape
```

Overrides: fipy.meshes.common.mesh.Mesh.__mul__() *(inherited documentation)*

__rmul__(*self*, *factor*)

Dilate a `Mesh` by `factor`.

```
>>> from fipy.meshes.grid2D import Grid2D
>>> baseMesh = Grid2D(dx = 1.0, dy = 1.0, nx = 2, ny = 2)
>>> print baseMesh.getCellCenters()
[[ 0.5 1.5 0.5 1.5]
 [ 0.5 0.5 1.5 1.5]]
```

The `factor` can be a scalar

```
>>> dilatedMesh = baseMesh * 3
>>> print dilatedMesh.getCellCenters()
[[ 1.5 4.5 1.5 4.5]
 [ 1.5 1.5 4.5 4.5]]
```

or a vector

```
>>> dilatedMesh = baseMesh * ((3,), (2,))
>>> print dilatedMesh.getCellCenters()
[[ 1.5 4.5 1.5 4.5]
 [ 1. 1. 3. 3. ]]
```

but the vector must have the same dimensionality as the `Mesh`

```
>>> dilatedMesh = baseMesh * ((3,), (2,), (1,))
Traceback (most recent call last):
...
ValueError: shape mismatch: objects cannot be broadcast to a single shape
```

`getVertexCoords`(*self*)

`getExteriorFaces`(*self*)

Return only the faces that have one neighboring cell.

Overrides: fipy.meshes.common.mesh.Mesh.getExteriorFaces()

`getInteriorFaces`(*self*)

Return only the faces that have two neighboring cells.

Overrides: fipy.meshes.common.mesh.Mesh.getInteriorFaces()

`getFaceCellIDs`(*self*)

`getFaceCenters(self)`

`__getstate__(self)`

`__setstate__(self, dict)`

Inherited from fipy.meshes.common.mesh.Mesh: __repr__(), getCellCenters(), getCellVolumes(), getCells(), getDim(), getFaces(), getFacesBack(), getFacesBottom(), getFacesDown(), getFacesFront(), getFacesLeft(), getFacesRight(), getFacesTop(), getFacesUp(), getNearestCell(), getNumberOfCells(), setScale()

Instance Variables

cellNormals	get geometry methods

Inherited from fipy.meshes.common.mesh.Mesh: cellAreas, cellToCellIDsFilled

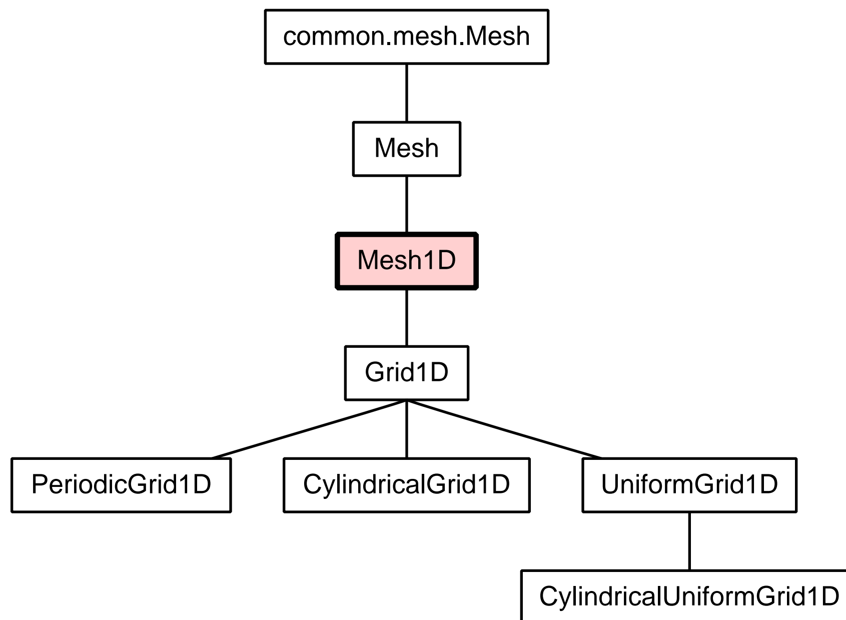
3.23 Module fipy.meshes.numMesh.mesh1D

Generic mesh class using numerix to do the calculations

Meshes contain cells, faces, and vertices.

This is built for a non-mixed element mesh.

Class Mesh1D



Known Subclasses: fipy.meshes.numMesh.grid1D.Grid1D

Methods

`__mul__`(*self*, *factor*)

Dilate a `Mesh` by `factor`.

```
>>> from fipy.meshes.grid2D import Grid2D
>>> baseMesh = Grid2D(dx = 1.0, dy = 1.0, nx = 2, ny = 2)
>>> print baseMesh.getCellCenters()
[[ 0.5 1.5 0.5 1.5]
 [ 0.5 0.5 1.5 1.5]]
```

The `factor` can be a scalar

```
>>> dilatedMesh = baseMesh * 3
>>> print dilatedMesh.getCellCenters()
[[ 1.5 4.5 1.5 4.5]
 [ 1.5 1.5 4.5 4.5]]
```

or a vector

```
>>> dilatedMesh = baseMesh * ((3,), (2,))
>>> print dilatedMesh.getCellCenters()
```

```
[[ 1.5 4.5 1.5 4.5]
 [ 1. 1. 3. 3. ]]
```

but the vector must have the same dimensionality as the Mesh

```
>>> dilatedMesh = baseMesh * ((3,), (2,), (1,))
Traceback (most recent call last):
...
ValueError: shape mismatch: objects cannot be broadcast to a single shape
```

Overrides: fipy.meshes.common.mesh.Mesh.__mul__() *(inherited documentation)*

Inherited from fipy.meshes.numMesh.mesh.Mesh: __add__(), __getstate__(), __init__(), __radd__(), __rmul__(), __setstate__(), getExteriorFaces(), getFaceCellIDs(), getFaceCenters(), getInteriorFaces(), getVertexCoords()

Inherited from fipy.meshes.common.mesh.Mesh: __repr__(), getCellCenters(), getCellVolumes(), getCells(), getDim(), getFaces(), getFacesBack(), getFacesBottom(), getFacesDown(), getFacesFront(), getFacesLeft(), getFacesRight(), getFacesTop(), getFacesUp(), getNearestCell(), getNumberOfCells(), setScale()

Instance Variables

Inherited from fipy.meshes.numMesh.mesh.Mesh: cellNormals

Inherited from fipy.meshes.common.mesh.Mesh: cellAreas, cellToCellIDsFilled

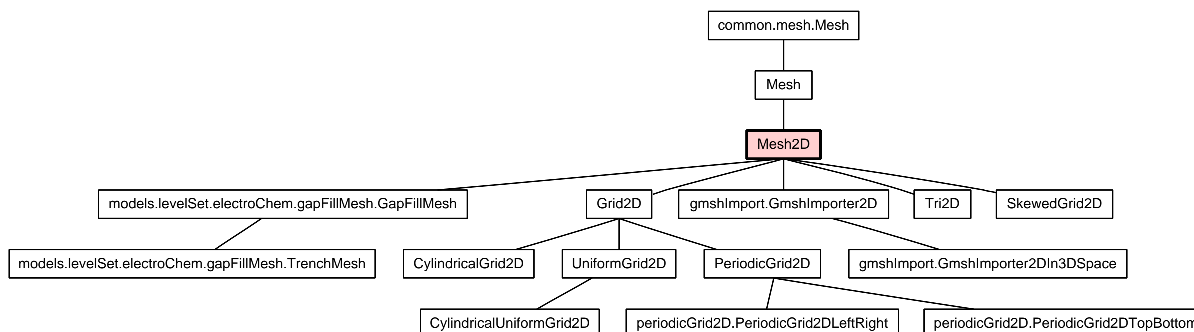
3.24 Module fipy.meshes.numMesh.mesh2D

Generic mesh class using numerix to do the calculations

Meshes contain cells, faces, and vertices.

This is built for a non-mixed element mesh.

Class Mesh2D



Known Subclasses: fipy.models.levelSet.electroChem.gapFillMesh.GapFillMesh, fipy.meshes.numMesh.grid2D.Grid2D, fipy.meshes.numMesh.gmshImport.GmshImporter2D, fipy.meshes.numMesh.tri2D.Tri2D, fipy.meshes.numMesh.skewedGrid2D.SkewedGrid2D

Methods

__mul__(*self*, *factor*)

Dilate a `Mesh` by `factor`.

```
>>> from fipy.meshes.grid2D import Grid2D
>>> baseMesh = Grid2D(dx = 1.0, dy = 1.0, nx = 2, ny = 2)
>>> print baseMesh.getCellCenters()
[[ 0.5 1.5 0.5 1.5]
 [ 0.5 0.5 1.5 1.5]]
```

The `factor` can be a scalar

```
>>> dilatedMesh = baseMesh * 3
>>> print dilatedMesh.getCellCenters()
[[ 1.5 4.5 1.5 4.5]
 [ 1.5 1.5 4.5 4.5]]
```

or a vector

```
>>> dilatedMesh = baseMesh * ((3,), (2,))
>>> print dilatedMesh.getCellCenters()
[[ 1.5 4.5 1.5 4.5]
 [ 1. 1. 3. 3. ]]
```

but the vector must have the same dimensionality as the `Mesh`

```
>>> dilatedMesh = baseMesh * ((3,), (2,), (1,))
Traceback (most recent call last):
...
ValueError: shape mismatch: objects cannot be broadcast to a single shape
```

Overrides: fipy.meshes.common.mesh.Mesh.__mul__() *(inherited documentation)*

extrude(*self*, *extrudeFunc*=<function <lambda> at 0x1a65770>, *layers*=1)

This function returns a new 3D mesh. The 2D mesh is extruded using the extrudeFunc and the number of layers.

Parameters:

- `extrudeFunc`: function that takes the vertex coordinates and returns the displaced values
- `layers`: the number of layers in the extruded mesh (number of times extrudeFunc will be called)

```
>>> from fipy.meshes.grid2D import Grid2D
>>> print Grid2D(nx=2,ny=2).extrude(layers=2).getCellCenters()
[[ 0.5 1.5 0.5 1.5 0.5 1.5 0.5 1.5]
 [ 0.5 0.5 1.5 1.5 0.5 0.5 1.5 1.5]
 [ 0.5 0.5 0.5 0.5 1.5 1.5 1.5 1.5]]
```

```
>>> from fipy.meshes.tri2D import Tri2D
>>> print Tri2D().extrude(layers=2).getCellCenters()
[[ 0.83333333 0.5 0.16666667 0.5 0.83333333 0.5
 0.16666667 0.5 ]
 [ 0.5 0.83333333 0.5 0.16666667 0.5 0.83333333
 0.5 0.16666667]
 [ 0.5 0.5 0.5 0.5 1.5 1.5 1.5
 1.5 ]]
```

Inherited from fipy.meshes.numMesh.mesh.Mesh: __add__(), __getstate__(), __init__(), __radd__(), __rmul__(), __setstate__(), getExteriorFaces(), getFaceCellIDs(), getFaceCenters(), getInteriorFaces(), getVertexCoords()

Inherited from fipy.meshes.common.mesh.Mesh: __repr__(), getCellCenters(), getCellVolumes(), getCells(), getDim(), getFaces(), getFacesBack(), getFacesBottom(), getFacesDown(), getFacesFront(), getFacesLeft(), getFacesRight(), getFacesTop(), getFacesUp(), getNearestCell(), getNumberOfCells(), setScale()

Instance Variables

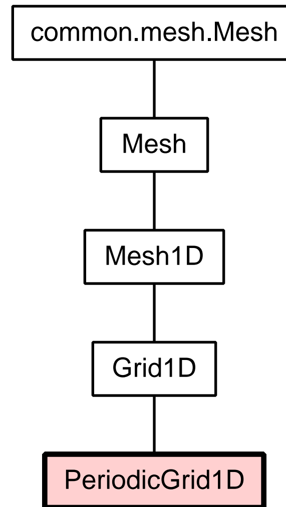
Inherited from fipy.meshes.numMesh.mesh.Mesh: cellNormals

Inherited from fipy.meshes.common.mesh.Mesh: cellAreas, cellToCellIDsFilled

3.25 Module fipy.meshes.numMesh.periodicGrid1D

Peridoic 1D Mesh

Class PeriodicGrid1D




```
>>> from fipy import numerix
```

Creates a Periodic grid mesh.

```
>>> mesh = PeriodicGrid1D(dx = (1, 2, 3))
```

```
>>> print numerix.nonzero(mesh.getExteriorFaces())[0]
[3]
```

```
>>> print mesh.getFaceCellIDs()
[[2 0 1 2]
 [0 1 2 --]]
```

```
>>> print mesh._getCellDistances()
[ 2.   1.5  2.5  1.5]
```

```
>>> print mesh._getCellToCellDistances()
[[ 2.   1.5  2.5]
 [ 1.5  2.5  2. ]]
```

```
>>> print mesh._getFaceNormals()
[[ 1.  1.  1.  1.]]
```

```
>>> print mesh._getCellVertexIDs()
[[1 2 2]
 [0 1 0]]
```

Methods

`__init__`(*self*, *dx*=1.0, *nx*=None)

---

faceVertexIds and cellFacesIds must be padded with minus ones.

Overrides: fipy.meshes.common.mesh.Mesh.__init__()

Inherited from fipy.meshes.numMesh.grid1D.Grid1D: __getstate__(), __repr__(), __setstate__(), getDim(), getPhysicalShape(), getScale(), getShape()

Inherited from fipy.meshes.numMesh.mesh1D.Mesh1D: __mul__()

Inherited from fipy.meshes.numMesh.mesh.Mesh: __add__(), __radd__(), __rmul__(), getExteriorFaces(), getFaceCellIDs(), getFaceCenters(), getInteriorFaces(), getVertexCoords()

Inherited from fipy.meshes.common.mesh.Mesh: getCellCenters(), getCellVolumes(), getCells(), getFaces(), getFacesBack(), getFacesBottom(), getFacesDown(), getFacesFront(), getFacesLeft(), getFacesRight(), getFacesTop(), getFacesUp(), getNearestCell(), getNumberOfCells(), setScale()

Instance Variables

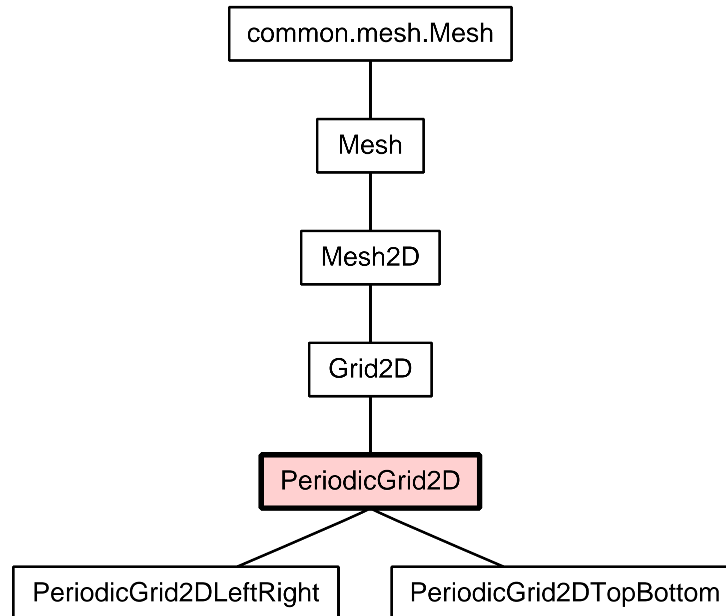
Inherited from fipy.meshes.numMesh.mesh.Mesh: cellNormals

Inherited from fipy.meshes.common.mesh.Mesh: cellAreas, cellToCellIDsFilled

3.26 Module fipy.meshes.numMesh.periodicGrid2D

2D periodic rectangular Mesh

Class PeriodicGrid2D



Known Subclasses: fipy.meshes.numMesh.periodicGrid2D.PeriodicGrid2DLeftRight, fipy.meshes.numMesh.periodicGrid2D.PeriodicGrid2DTopBottom

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.nonzero(mesh.getExteriorFaces())[0]
[ 4  5  8 11]

>>> print mesh.getFaceCellIDs()
[[2 3 0 1 2 3 1 0 1 3 2 3]
 [0 1 2 3 -- -- 0 1 -- 2 3 --]]

>>> print mesh._getCellDistances()
[ 0.5   0.5   0.5   0.5   0.25  0.25  1.    1.    0.5   1.    1.    0.5 ]

>>> print (mesh._getCellFaceIDs() == [[0, 1, 2, 3],
...                                    [7, 6, 10, 9],
...                                    [2, 3, 0, 1],
...                                    [6, 7, 9, 10]]).flatten().all()
True
```

```
>>> print mesh._getCellToCellDistances()
[[ 0.5  0.5  0.5  0.5]
 [ 1.   1.   1.   1. ]
 [ 0.5  0.5  0.5  0.5]
 [ 1.   1.   1.   1. ]]

>>> 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]]

>>> from fipy.tools import numerix
>>> numerix.allclose(mesh._getFaceNormals(), normals)
1

>>> print mesh._getCellVertexIDs()
[[4 5 7 8]
 [3 4 6 7]
 [1 2 4 5]
 [0 1 3 4]]
```

Methods

__init__(*self*, *dx*=1.0, *dy*=1.0, *nx*=None, *ny*=None)

---

faceVertexIds and cellFacesIds must be padded with minus ones.

Overrides: fipy.meshes.common.mesh.Mesh.__init__()

Inherited from fipy.meshes.numMesh.grid2D.Grid2D: __getstate__(), __repr__(), __setstate__(), getPhysicalShape(), getScale(), getShape()

Inherited from fipy.meshes.numMesh.mesh2D.Mesh2D: __mul__(), extrude()

Inherited from fipy.meshes.numMesh.mesh.Mesh: __add__(), __radd__(), __rmul__(), getExteriorFaces(), getFaceCellIDs(), getFaceCenters(), getInteriorFaces(), getVertexCoords()

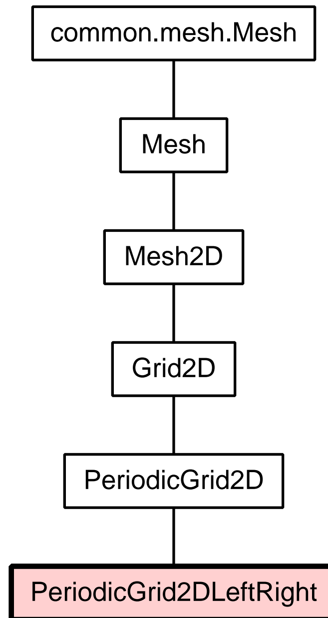
Inherited from fipy.meshes.common.mesh.Mesh: getCellCenters(), getCellVolumes(), getCells(), getDim(), getFaces(), getFacesBack(), getFacesBottom(), getFacesDown(), getFacesFront(), getFacesLeft(), getFacesRight(), getFacesTop(), getFacesUp(), getNearestCell(), getNumberOfCells(), setScale()

Instance Variables

Inherited from fipy.meshes.numMesh.mesh.Mesh: cellNormals

Inherited from fipy.meshes.common.mesh.Mesh: cellAreas, cellToCellIDsFilled

Class PeriodicGrid2DLeftRight



Methods

`__init__`(*self*, *dx*=1.0, *dy*=1.0, *nx*=None, *ny*=None)

---

faceVertexIds and cellFacesIds must be padded with minus ones.

Overrides: fipy.meshes.common.mesh.Mesh.__init__()

Inherited from fipy.meshes.numMesh.grid2D.Grid2D: __getstate__(), __repr__(), __setstate__(), getPhysicalShape(), getScale(), getShape()

Inherited from fipy.meshes.numMesh.mesh2D.Mesh2D: __mul__(), extrude()

Inherited from fipy.meshes.numMesh.mesh.Mesh: __add__(), __radd__(), __rmul__(), getExteriorFaces(), getFaceCellIDs(), getFaceCenters(), getInteriorFaces(), getVertexCoords()

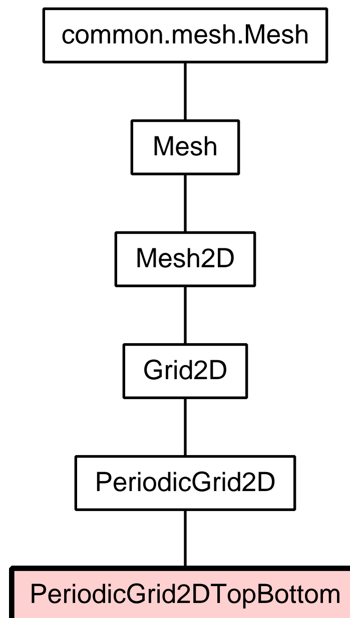
Inherited from fipy.meshes.common.mesh.Mesh: getCellCenters(), getCellVolumes(), getCells(), getDim(), getFaces(), getFacesBack(), getFacesBottom(), getFacesDown(), getFacesFront(), getFacesLeft(), getFacesRight(), getFacesTop(), getFacesUp(), getNearestCell(), getNumberOfCells(), setScale()

Instance Variables

Inherited from fipy.meshes.numMesh.mesh.Mesh: cellNormals

Inherited from fipy.meshes.common.mesh.Mesh: cellAreas, cellToCellIDsFilled

Class PeriodicGrid2DTopBottom



Methods

`__init__(self, dx=1.0, dy=1.0, nx=None, ny=None)`

faceVertexIds and cellFacesIds must be padded with minus ones.

Overrides: fipy.meshes.common.mesh.Mesh.__init__()

Inherited from fipy.meshes.numMesh.grid2D.Grid2D: __getstate__(), __repr__(), __setstate__(), getPhysicalShape(), getScale(), getShape()

Inherited from fipy.meshes.numMesh.mesh2D.Mesh2D: __mul__(), extrude()

Inherited from fipy.meshes.numMesh.mesh.Mesh: __add__(), __radd__(), __rmul__(), getExteriorFaces(), getFaceCellIDs(), getFaceCenters(), getInteriorFaces(), getVertexCoords()

Inherited from fipy.meshes.common.mesh.Mesh: getCellCenters(), getCellVolumes(), getCells(), getDim(), getFaces(), getFacesBack(), getFacesBottom(), getFacesDown(), getFacesFront(), getFacesLeft(), getFacesRight(), getFacesTop(), getFacesUp(), getNearestCell(), getNumberOfCells(), setScale()

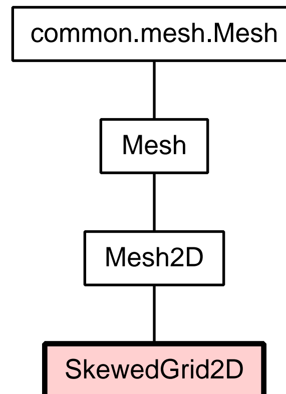
Instance Variables

Inherited from fipy.meshes.numMesh.mesh.Mesh: cellNormals

Inherited from fipy.meshes.common.mesh.Mesh: cellAreas, cellToCellIDsFilled

3.27 Module fipy.meshes.numMesh.skewedGrid2D

Class SkewedGrid2D



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.

Methods

`__init__`(*self*, *dx*=`1.0`, *dy*=`1.0`, *nx*=`None`, *ny*=`1`, *rand*=`0`)

---

faceVertexIds and cellFacesIds must be padded with minus ones.

Overrides: fipy.meshes.common.mesh.Mesh.__init__()

`getScale`(*self*)

`getPhysicalShape(self)`

Return physical dimensions of Grid2D.

`getShape(self)`

`__getstate__(self)`

Overrides: fipy.meshes.numMesh.mesh.Mesh.__getstate__()

`__setstate__(self, dict)`

Overrides: fipy.meshes.numMesh.mesh.Mesh.__setstate__()

Inherited from fipy.meshes.numMesh.mesh2D.Mesh2D: __mul__(), extrude()

Inherited from fipy.meshes.numMesh.mesh.Mesh: __add__(), __radd__(), __rmul__(), getExteriorFaces(), getFaceCellIDs(), getFaceCenters(), getInteriorFaces(), getVertexCoords()

Inherited from fipy.meshes.common.mesh.Mesh: __repr__(), getCellCenters(), getCellVolumes(), getCells(), getDim(), getFaces(), getFacesBack(), getFacesBottom(), getFacesDown(), getFacesFront(), getFacesLeft(), getFacesRight(), getFacesTop(), getFacesUp(), getNearestCell(), getNumberOfCells(), setScale()

Instance Variables

Inherited from fipy.meshes.numMesh.mesh.Mesh: cellNormals

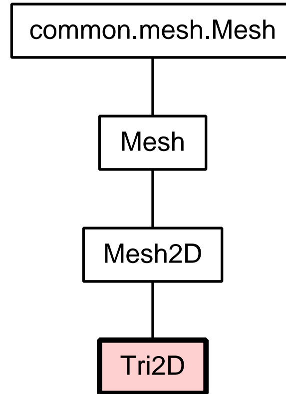
Inherited from fipy.meshes.common.mesh.Mesh: cellAreas, cellToCellIDsFilled

3.28 Module fipy.meshes.numMesh.test

Test numeric implementation of the mesh

3.29 Module fipy.meshes.numMesh.tri2D

Class Tri2D



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.

Methods

`__init__`(*self*, *dx*=`1.0`, *dy*=`1.0`, *nx*=`1`, *ny*=`1`)

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`.

Overrides: fipy.meshes.common.mesh.Mesh.`__init__`()

`getScale`(*self*)

---

getPhysicalShape(*self*)

Return physical dimensions of Grid2D.

getShape(*self*)

__getstate__(*self*)

Overrides: fipy.meshes.numMesh.mesh.Mesh.__getstate__()

__setstate__(*self*, *dict*)

Overrides: fipy.meshes.numMesh.mesh.Mesh.__setstate__()

Inherited from fipy.meshes.numMesh.mesh2D.Mesh2D: __mul__(), extrude()

Inherited from fipy.meshes.numMesh.mesh.Mesh: __add__(), __radd__(), __rmul__(), getExteriorFaces(), getFaceCellIDs(), getFaceCenters(), getInteriorFaces(), getVertexCoords()

Inherited from fipy.meshes.common.mesh.Mesh: __repr__(), getCellCenters(), getCellVolumes(), getCells(), getDim(), getFaces(), getFacesBack(), getFacesBottom(), getFacesDown(), getFacesFront(), getFacesLeft(), getFacesRight(), getFacesTop(), getFacesUp(), getNearestCell(), getNumberOfCells(), setScale()

Instance Variables

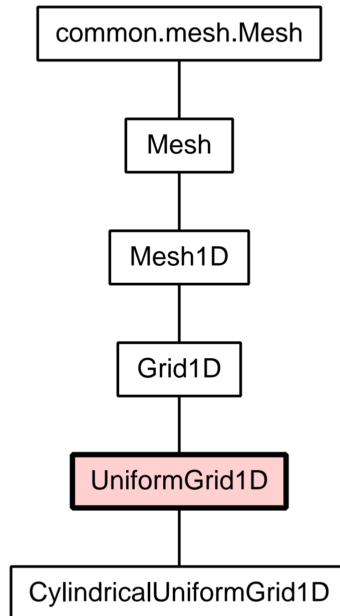
Inherited from fipy.meshes.numMesh.mesh.Mesh: cellNormals

Inherited from fipy.meshes.common.mesh.Mesh: cellAreas, cellToCellIDsFilled

3.30 Module fipy.meshes.numMesh.uniformGrid1D

1D Mesh

Class UniformGrid1D



Known Subclasses: fipy.meshes.numMesh.cylindricalUniformGrid1D.CylindricalUniformGrid1D

Creates a 1D grid mesh.

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

Methods

`__init__`(*self*, *dx*=1.0, *nx*=1, *origin*=(0))

faceVertexIds and cellFacesIds must be padded with minus ones.

Overrides: fipy.meshes.common.mesh.Mesh.__init__()

`__mul__`(*self*, *factor*)

Dilate a `Mesh` by `factor`.

```
>>> from fipy.meshes.grid2D import Grid2D
>>> baseMesh = Grid2D(dx = 1.0, dy = 1.0, nx = 2, ny = 2)
>>> print baseMesh.getCellCenters()
[[ 0.5 1.5 0.5 1.5]
 [ 0.5 0.5 1.5 1.5]]
```

The `factor` can be a scalar

```
>>> dilatedMesh = baseMesh * 3
>>> print dilatedMesh.getCellCenters()
[[ 1.5 4.5 1.5 4.5]
 [ 1.5 1.5 4.5 4.5]]
```

or a vector

```
>>> dilatedMesh = baseMesh * ((3,), (2,))
>>> print dilatedMesh.getCellCenters()
[[ 1.5 4.5 1.5 4.5]
 [ 1. 1. 3. 3. ]]
```

but the vector must have the same dimensionality as the `Mesh`

```
>>> dilatedMesh = baseMesh * ((3,), (2,), (1,))
Traceback (most recent call last):
...
ValueError: shape mismatch: objects cannot be broadcast to a single shape
```

Overrides: fipy.meshes.common.mesh.Mesh.__mul__() *(inherited documentation)*

`getInteriorFaces(`*self*`)`

Return only the faces that have two neighboring cells.

Overrides: fipy.meshes.common.mesh.Mesh.getInteriorFaces()

`getVertexCoords(`*self*`)`

Overrides: fipy.meshes.numMesh.mesh.Mesh.getVertexCoords()

`getFaceCellIDs(`*self*`)`

Overrides: fipy.meshes.numMesh.mesh.Mesh.getFaceCellIDs()

getCellVolumes(*self*)

Overrides: fipy.meshes.common.mesh.Mesh.getCellVolumes()

getCellCenters(*self*)

Overrides: fipy.meshes.common.mesh.Mesh.getCellCenters()

getFaceCenters(*self*)

Overrides: fipy.meshes.numMesh.mesh.Mesh.getFaceCenters()

Inherited from fipy.meshes.numMesh.grid1D.Grid1D: __getstate__(), __repr__(), __setstate__(), getDim(), getPhysicalShape(), getScale(), getShape()

Inherited from fipy.meshes.numMesh.mesh.Mesh: __add__(), __radd__(), __rmul__(), getExteriorFaces()

Inherited from fipy.meshes.common.mesh.Mesh: getCells(), getFaces(), getFacesBack(), getFacesBottom(), getFacesDown(), getFacesFront(), getFacesLeft(), getFacesRight(), getFacesTop(), getFacesUp(), getNearestCell(), getNumberOfCells(), setScale()

Instance Variables

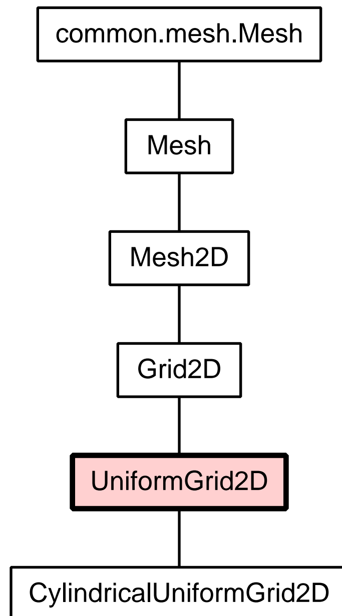
Inherited from fipy.meshes.numMesh.mesh.Mesh: cellNormals

Inherited from fipy.meshes.common.mesh.Mesh: cellAreas, cellToCellIDsFilled

3.31 Module fipy.meshes.numMesh.uniformGrid2D

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

Class UniformGrid2D



Known Subclasses: fipy.meshes.numMesh.cylindricalUniformGrid2D.CylindricalUniformGrid2D

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

Methods

`__init__`(*self*, *dx*=`1.0`, *dy*=`1.0`, *nx*=`1`, *ny*=`1`, *origin*=`((0), (0))`)

faceVertexIds and cellFacesIds must be padded with minus ones.

Overrides: fipy.meshes.common.mesh.Mesh.__init__()

`__mul__`(*self*, *factor*)

Dilate a `Mesh` by `factor`.

```
>>> from fipy.meshes.grid2D import Grid2D
>>> baseMesh = Grid2D(dx = 1.0, dy = 1.0, nx = 2, ny = 2)
>>> print baseMesh.getCellCenters()
[[ 0.5 1.5 0.5 1.5]
 [ 0.5 0.5 1.5 1.5]]
```

The `factor` can be a scalar

```
>>> dilatedMesh = baseMesh * 3
>>> print dilatedMesh.getCellCenters()
[[ 1.5 4.5 1.5 4.5]
 [ 1.5 1.5 4.5 4.5]]
```

or a vector

```
>>> dilatedMesh = baseMesh * ((3,), (2,))
>>> print dilatedMesh.getCellCenters()
[[ 1.5 4.5 1.5 4.5]
 [ 1. 1. 3. 3. ]]
```

but the vector must have the same dimensionality as the `Mesh`

```
>>> dilatedMesh = baseMesh * ((3,), (2,), (1,))
Traceback (most recent call last):
...
ValueError: shape mismatch: objects cannot be broadcast to a single shape
```

Overrides: fipy.meshes.common.mesh.Mesh.__mul__() *(inherited documentation)*

`getExteriorFaces`(*self*)

Return only the faces that have one neighboring cell.

Overrides: fipy.meshes.common.mesh.Mesh.getExteriorFaces()

`getInteriorFaces`(*self*)

Return only the faces that have two neighboring cells.

Overrides: fipy.meshes.common.mesh.Mesh.getInteriorFaces()

`getVertexCoords`(*self*)

Overrides: fipy.meshes.numMesh.mesh.Mesh.getVertexCoords()

`getFaceCellIDs`(*self*)

Overrides: fipy.meshes.numMesh.mesh.Mesh.getFaceCellIDs()

`getCellVolumes`(*self*)

Overrides: fipy.meshes.common.mesh.Mesh.getCellVolumes()

`getCellCenters`(*self*)

Overrides: fipy.meshes.common.mesh.Mesh.getCellCenters()

`getFaceCenters`(*self*)

Overrides: fipy.meshes.numMesh.mesh.Mesh.getFaceCenters()

Inherited from fipy.meshes.numMesh.grid2D.Grid2D: __getstate__(), __repr__(), __setstate__(), getPhysicalShape(), getScale(), getShape()

Inherited from fipy.meshes.numMesh.mesh2D.Mesh2D: extrude()

Inherited from fipy.meshes.numMesh.mesh.Mesh: __add__(), __radd__(), __rmul__()

Inherited from fipy.meshes.common.mesh.Mesh: getCells(), getDim(), getFaces(), getFacesBack(), getFacesBottom(), getFacesDown(), getFacesFront(), getFacesLeft(), getFacesRight(), getFacesTop(), getFacesUp(), getNearestCell(), getNumberOfCells(), setScale()

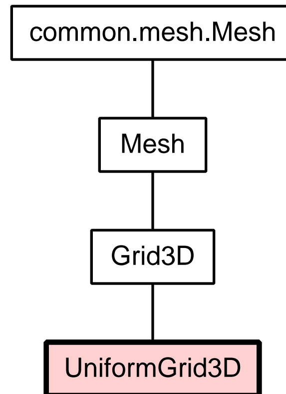
Instance Variables

Inherited from fipy.meshes.numMesh.mesh.Mesh: cellNormals

Inherited from fipy.meshes.common.mesh.Mesh: cellAreas, cellToCellIDsFilled

3.32 Module fipy.meshes.numMesh.uniformGrid3D

Class UniformGrid3D



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.

Methods

`__init__`(*self*, *dx*=`1.0`, *dy*=`1.0`, *dz*=`1.0`, *nx*=`1`, *ny*=`1`, *nz*=`1`, *origin*=`[[0], [0], [0]]`)

---

faceVertexIds and cellFacesIds must be padded with minus ones.

Overrides: fipy.meshes.common.mesh.Mesh.`__init__`()

`__mul__`(*self*, *factor*)

Dilate a `Mesh` by `factor`.

```
>>> from fipy.meshes.grid2D import Grid2D
>>> baseMesh = Grid2D(dx = 1.0, dy = 1.0, nx = 2, ny = 2)
>>> print baseMesh.getCellCenters()
[[ 0.5 1.5 0.5 1.5]
 [ 0.5 0.5 1.5 1.5]]
```

The `factor` can be a scalar

```
>>> dilatedMesh = baseMesh * 3
>>> print dilatedMesh.getCellCenters()
[[ 1.5 4.5 1.5 4.5]
 [ 1.5 1.5 4.5 4.5]]
```

or a vector

```
>>> dilatedMesh = baseMesh * ((3,), (2,))
>>> print dilatedMesh.getCellCenters()
[[ 1.5 4.5 1.5 4.5]
 [ 1. 1. 3. 3. ]]
```

but the vector must have the same dimensionality as the `Mesh`

```
>>> dilatedMesh = baseMesh * ((3,), (2,), (1,))
Traceback (most recent call last):
...
ValueError: shape mismatch: objects cannot be broadcast to a single shape
```

Overrides: fipy.meshes.common.mesh.Mesh.__mul__() *(inherited documentation)*

getExteriorFaces(*self*)

Return only the faces that have one neighboring cell.

Overrides: fipy.meshes.common.mesh.Mesh.getExteriorFaces()

getInteriorFaces(*self*)

Return only the faces that have two neighboring cells

Overrides: fipy.meshes.common.mesh.Mesh.getInteriorFaces()

getVertexCoords(*self*)

Overrides: fipy.meshes.numMesh.mesh.Mesh.getVertexCoords()

`getFaceCellIDs(self)`

Overrides: fipy.meshes.numMesh.mesh.Mesh.getFaceCellIDs()

`getCellVolumes(self)`

Overrides: fipy.meshes.common.mesh.Mesh.getCellVolumes()

`getCellCenters(self)`

Overrides: fipy.meshes.common.mesh.Mesh.getCellCenters()

`getFaceCenters(self)`

Overrides: fipy.meshes.numMesh.mesh.Mesh.getFaceCenters()

Inherited from fipy.meshes.numMesh.grid3D.Grid3D: __getstate__(), __repr__(), __setstate__(), getPhysicalShape(), getScale(), getShape()

Inherited from fipy.meshes.numMesh.mesh.Mesh: __add__(), __radd__(), __rmul__()

Inherited from fipy.meshes.common.mesh.Mesh: getCells(), getDim(), getFaces(), getFacesBack(), getFacesBottom(), getFacesDown(), getFacesFront(), getFacesLeft(), getFacesRight(), getFacesTop(), getFacesUp(), getNearestCell(), getNumberOfCells(), setScale()

Instance Variables

Inherited from fipy.meshes.numMesh.mesh.Mesh: cellNormals

Inherited from fipy.meshes.common.mesh.Mesh: cellAreas, cellToCellIDsFilled

3.33 Module fipy.meshes.periodicGrid1D

3.34 Module fipy.meshes.periodicGrid2D

3.35 Package fipy.meshes.pyMesh

Description of mesh geometries

3.36 Module fipy.meshes.pyMesh.cell

Cell within a mesh

Class Cell

Cell within a mesh

`Cell` objects are bounded by `Face` objects.

Methods

`__init__`(*self*, *faces*, *faceOrientations*, *id*)

`Cell` is initialized by `Mesh`

Parameters

faces: `list` or `tuple` of bounding faces that define the cell

faceOrientations: `list`, `tuple`, or `numerix.array` of orientations (+/-1) to indicate whether a face points into this face or out of it. Can be calculated, but the mesh typically knows this information already.

id: unique identifier

`getID`(*self*)

Return the id of this Cell.

`getFaceOrientations`(*self*)

`getVolume`(*self*)

Return the volume of the `Cell`.

`getCenter`(*self*)

Return the coordinates of the Cell center.

__repr__(*self*)

Textual representation of Cell.

getFaces(*self*)

Return the faces bounding the Cell.

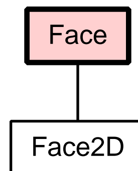
getFaceIDs(*self*)

getBoundingCells(*self*)

3.37 Module fipy.meshes.pyMesh.face

Face within a Mesh

Class Face



Known Subclasses: fipy.meshes.pyMesh.face2D.Face2D

Face within a Mesh

Face objects are bounded by Vertex objects. Face objects separate Cell objects.

Methods

`__init__`(*self*, *vertices*, *id*)

`Face` is initialized by `Mesh`

> **Parameters**
>
> *vertices:* the `Vertex` points that bound the `Face`
>
> *id:* a unique identifier

`addBoundingCell`(*self*, *cell*, *orientation*)

Add `cell` to the list of `Cell` objects which lie on either side of this Face.

`getCells`(*self*)

Return the `Cell` objects which lie on either side of this `Face`.

`getID`(*self*)

`getCellID`(*self*, *index*=0)

Return the `id` of the specified `Cell` on one side of this `Face`.

`getCenter`(*self*)

Return the coordinates of the `Face` center.

`getArea`(*self*)

Return the area of the `Face`.

`getNormal(self)`

Return the unit normal vector

`getCellDistance(self)`

Return the distance between adjacent `Cell` centers.

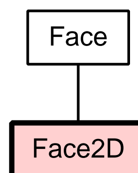
`__repr__(self)`

Textual representation of `Face`.

3.38 Module fipy.meshes.pyMesh.face2D

1D (edge) Face in a 2D Mesh

Class Face2D



1D (edge) Face in a 2D Mesh

Face2D is bounded by two Vertices.

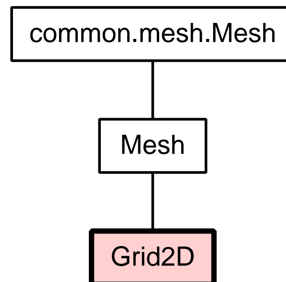
Methods

Inherited from fipy.meshes.pyMesh.face.Face: __init__(), __repr__(), addBoundingCell(), getArea(), getCellDistance(), getCellID(), getCells(), getCenter(), getID(), getNormal()

3.39 Module fipy.meshes.pyMesh.grid2D

2D rectangular Mesh

Class Grid2D



2D rectangular Mesh

Numbering system

nx=5

ny=3

Cells:

```
*************************************
*      *      *       *      *      *
* 10   * 11   * 12    * 13   * 14   *
*************************************
*      *      *       *      *      *
* 5    * 6    * 7     * 8    * 9    *
*************************************
*      *      *       *      *      *
* 0    * 1    * 2     * 3    * 4    *
*************************************
```

Faces (before reordering):

```
***15******16*****17******18****19***
*      *      *       *      *      *
32    33      34      35     36     37
***10******11*****12******13*****14**
*      *      *       *      *      *
26     27     28      29     30     31
***5*******6******7*******8******9***
*      *      *       *      *      *
20     21     22      23     24     25
***0*******1******2*******3******4***
```

Faces (after reordering):

```
***27******28*****29******30****31***
*      *      *       *      *      *
34     18     19      20     21     37
***5*******6******7*******8******9***
```

```
*      *      *       *      *      *
33     14     15      16     17     36
***0*******1******2*******3******4***
*      *      *       *      *      *
32     10     11      12     13     35
***22******23*****24******25*****26**
```

Vertices:

```
18*****19*****20******21*****22****23
*      *      *       *      *      *
*      *      *       *      *      *
12*****13*****14******15*****16****17
*      *      *       *      *      *
*      *      *       *      *      *
6******7******8*******9******10****11
*      *      *       *      *      *
*      *      *       *      *      *
0******1******2*******3******4******5
```

Methods

`__init__`(*self*, *dx*, *dy*, *nx*, *ny*)

Grid2D is initialized by caller

Parameters

dx: dimension of each cell in **x** direction

dy: dimension of each cell in **y** direction

nx: number of cells in **x** direction

ny: number of cells in **y** direction

Overrides: fipy.meshes.common.mesh.Mesh.__init__()

`getFacesLeft`(*self*)

Return list of faces on left boundary of Grid2D with the x-axis running from left to right.

Overrides: fipy.meshes.common.mesh.Mesh.getFacesLeft()

`getFacesRight`(*self*)

Return list of faces on right boundary of Grid2D with the x-axis running from left to right.

Overrides: fipy.meshes.common.mesh.Mesh.getFacesRight()

getFacesTop(*self*)

Return list of faces on top boundary of Grid2D with the y-axis running from bottom to top.

Overrides: fipy.meshes.common.mesh.Mesh.getFacesTop()

getFacesBottom(*self*)

Return list of faces on bottom boundary of Grid2D with the y-axis running from bottom to top.

Overrides: fipy.meshes.common.mesh.Mesh.getFacesBottom()

getShape(*self*)

Return cell dimensions `Grid2D`.

getPhysicalShape(*self*)

Return physical dimensions of Grid2D.

Overrides: fipy.meshes.pyMesh.mesh.Mesh.getPhysicalShape()

getCellVolumes(*self*)

Overrides: fipy.meshes.common.mesh.Mesh.getCellVolumes()

getCellCenters(*self*)

Overrides: fipy.meshes.common.mesh.Mesh.getCellCenters()

Inherited from fipy.meshes.pyMesh.mesh.Mesh: getExteriorFaces(), getFaceOrientations(), getScale(), setScale()

Inherited from fipy.meshes.common.mesh.Mesh: __add__(), __mul__(), __repr__(), getCells(), getDim(), getFaces(), getFacesBack(), getFacesDown(), getFacesFront(), getFacesUp(), getInteriorFaces(), getNearestCell(), getNumberOfCells()

Instance Variables

Inherited from fipy.meshes.pyMesh.mesh.Mesh: faceOrientations, scale

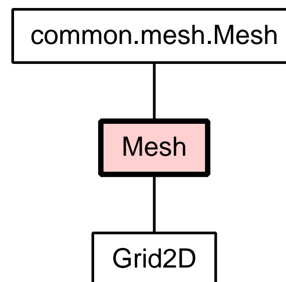
Inherited from fipy.meshes.common.mesh.Mesh: cellAreas, cellToCellIDsFilled

3.40 Module fipy.meshes.pyMesh.mesh

Generic mesh class

Meshes contain cells, faces, and vertices.

Class Mesh



Known Subclasses: fipy.meshes.pyMesh.grid2D.Grid2D

Methods

`__init__`(*self*, *cells*, *faces*, *interiorFaces*, *vertices*)

Overrides: fipy.meshes.common.mesh.Mesh.`__init__`()

`getExteriorFaces`(*self*)

Return only the faces that have one neighboring cell.

Overrides: fipy.meshes.common.mesh.Mesh.getExteriorFaces()

`getFaceOrientations`(*self*)

---

getPhysicalShape(*self*)

Return physical dimensions of Mesh.

getScale(*self*)

setScale(*self*, *scale*)

Overrides: fipy.meshes.common.mesh.Mesh.setScale()

Inherited from fipy.meshes.common.mesh.Mesh: __add__(), __mul__(), __repr__(), getCellCenters(), getCellVolumes(), getCells(), getDim(), getFaces(), getFacesBack(), getFacesBottom(), getFacesDown(), getFacesFront(), getFacesLeft(), getFacesRight(), getFacesTop(), getFacesUp(), getInteriorFaces(), getNearestCell(), getNumberOfCells()

Instance Variables

faceOrientations get topology methods

scale calc geometry methods

Inherited from fipy.meshes.common.mesh.Mesh: cellAreas, cellToCellIDsFilled

3.41 Module fipy.meshes.pyMesh.test

Test numeric implementation of the mesh

3.42 Module fipy.meshes.pyMesh.vertex

Vertex within a Mesh

Vertices bound Faces.

Class Vertex

Methods

`__init__(self, coordinates)`

Vertex is initialized by Mesh with its coordinates.

`getCoordinates(self)`

Return coordinates of Vertex.

`__repr__(self)`

Textual representation of Vertex.

3.43 Module fipy.meshes.skewedGrid2D

3.44 Module fipy.meshes.test

Test implementation of the mesh

3.45 Module fipy.meshes.tri2D

Package fipy.models

4.1 Package fipy.models.levelSet

4.2 Package fipy.models.levelSet.advection

4.3 Module fipy.models.levelSet.advection.advectionEquation

Functions

`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.

4.4 Module fipy.models.levelSet.advection.advectionTerm

4.5 Module fipy.models.levelSet.advection.higherOrderAdvectionEquation

Functions

`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

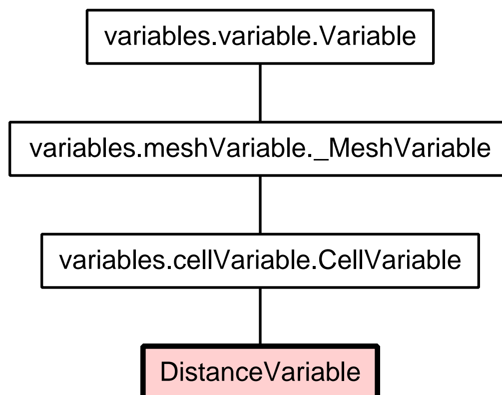
advectionCoeff: The `coeff` to pass to the `_HigherOrderAdvectionTerm`

4.6 Module fipy.models.levelSet.advection.higherOrderAdvectionTerm

4.7 Package fipy.models.levelSet.distanceFunction

4.8 Module fipy.models.levelSet.distanceFunction.distanceVariable

Class DistanceVariable



A `DistanceVariable` object calculates ϕ so it satisfies,

$$|\nabla \phi| = 1$$

using the fast marching method with an initial condition defined by the zero level set. Currently the solution is first order, This suffices for initial conditions with straight edges (e.g. trenches in electrodeposition). The method should work for unstructured 2D grids but testing on unstructured grids is untested thus far. This is a 2D implementation as it stands. Extending to 3D should be relatively simple.

Here we will define a few test cases. Firstly a 1D test case

```
>>> from fipy.meshes.grid1D import Grid1D
>>> mesh = Grid1D(dx = .5, nx = 8)
>>> 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)
>>> var = DistanceVariable(mesh = mesh, value = (-1, -1, -1, -1, 1, 1, 1, 1))
```

```
>>> var.calcDistanceFunction()
>>> answer = numerix.arange(8) * dx - 3.5 * dx
>>> print var.allclose(answer)
1
```

A 2D test case to test `_calcTrialValue` for a pathological case.

```
>>> dx = 1.
>>> dy = 2.
>>> from fipy.meshes.grid2D import Grid2D
>>> mesh = Grid2D(dx = dx, dy = dy, nx = 2, ny = 3)
>>> var = DistanceVariable(mesh = mesh, value = (-1, 1, 1, 1, -1, 1))

>>> var.calcDistanceFunction()
>>> vbl = -dx * dy / numerix.sqrt(dx**2 + dy**2) / 2.
>>> vbr = dx / 2
>>> vml = dy / 2.
>>> crossProd = dx * dy
>>> dsq = dx**2 + dy**2
>>> top = vbr * dx**2 + vml * dy**2
>>> sqrt = crossProd**2 *(dsq - (vbr - vml)**2)
>>> sqrt = numerix.sqrt(max(sqrt, 0))
>>> vmr = (top + sqrt) / dsq
>>> answer = (vbl, vbr, vml, vmr, vbl, vbr)
>>> print var.allclose(answer)
1
```

The `extendVariable` method solves the following equation for a given extensionVariable.

$$\nabla u \cdot \nabla \phi = 0$$

using the fast marching method with an initial condition defined at the zero level set. Essentially the equation solves a fake distance function to march out the velocity from the interface.

```
>>> from fipy.variables.cellVariable import CellVariable
>>> mesh = Grid2D(dx = 1., dy = 1., nx = 2, ny = 2)
>>> var = DistanceVariable(mesh = mesh, value = (-1, 1, 1, 1))
>>> var.calcDistanceFunction()
>>> extensionVar = CellVariable(mesh = mesh, value = (-1, .5, 2, -1))
>>> tmp = 1 / numerix.sqrt(2)
>>> print var.allclose((-tmp / 2, 0.5, 0.5, 0.5 + tmp))
1
>>> var.extendVariable(extensionVar)
>>> print extensionVar.allclose((1.25, .5, 2, 1.25))
1
>>> mesh = Grid2D(dx = 1., dy = 1., nx = 3, ny = 3)
>>> var = DistanceVariable(mesh = mesh, value = (-1, 1, 1,
...                                               1, 1, 1,
...                                               1, 1, 1))
```

```
>>> var.calcDistanceFunction()
>>> extensionVar = CellVariable(mesh = mesh, value = (-1, .5, -1,
...                                                    2, -1, -1,
...                                                   -1, -1, -1))

>>> v1 = 0.5 + tmp
>>> v2 = 1.5
>>> tmp1 = (v1 + v2) / 2 + numerix.sqrt(2. - (v1 - v2)**2) / 2
>>> tmp2 = tmp1 + 1 / numerix.sqrt(2)
>>> print var.allclose((-tmp / 2, 0.5, 1.5, 0.5, 0.5 + tmp,
...                     tmp1, 1.5, tmp1, tmp2))
1
>>> answer = (1.25, .5, .5, 2, 1.25, 0.9544, 2, 1.5456, 1.25)
>>> var.extendVariable(extensionVar)
>>> print extensionVar.allclose(answer, rtol = 1e-4)
1
```

Test case for a bug that occurs when initializing the distance variable at the interface. Currently it is assumed that adjacent cells that are opposite sign neighbors have perpendicular normal vectors. In fact the two closest cells could have opposite normals.

```
>>> mesh = Grid1D(dx = 1., nx = 3)
>>> var = DistanceVariable(mesh = mesh, value = (-1, 1, -1))
>>> var.calcDistanceFunction()
>>> print var.allclose((-0.5, 0.5, -0.5))
1
```

For future reference, the minimum distance for the interface cells can be calculated with the following functions. The trial cell values will also be calculated with these functions. In essence it is not difficult to calculate the level set distance function on an unstructured 3D grid. However a lot of testing will be required. The minimum distance functions will take the following form.

$$X_{\text{min}} = \frac{\left|\vec{s} \times \vec{t}\right|}{\left|\vec{s} - \vec{t}\right|}$$

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.

Methods

__init__(*self*, *mesh*, *name*='', *value*=0.0, *unit*=None, *hasOld*=0, *narrowBandWidth*=10000000000.0)

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.

Overrides: fipy.variables.variable.Variable.__init__()

`extendVariable`(*self*, *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.

`calcDistanceFunction`(*self*, *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.

`getCellInterfaceAreas`(*self*)

Returns the length of the interface that crosses the cell

A simple 1D test:

```
>>> from fipy.meshes.grid1D import Grid1D
>>> mesh = Grid1D(dx = 1., nx = 4)
>>> distanceVariable = DistanceVariable(mesh = mesh,
...  value = (-1.5, -0.5, 0.5, 1.5))
>>> numerix.allclose(distanceVariable.getCellInterfaceAreas(),
...  (0, 0., 1., 0))
1
```

A 2D test case:

```
>>> from fipy.meshes.grid2D import Grid2D
>>> mesh = Grid2D(dx = 1., dy = 1., nx = 3, ny = 3)
>>> distanceVariable = DistanceVariable(mesh = mesh,
...  value = (1.5, 0.5, 1.5,
...  0.5,-0.5, 0.5,
...  1.5, 0.5, 1.5))
>>> numerix.allclose(distanceVariable.getCellInterfaceAreas(),
...  (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))
>>> numerix.allclose(distanceVariable.getCellInterfaceAreas(),
...  (0, numerix.sqrt(2) / 4, numerix.sqrt(2) / 4, 0))
1
```

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.getCellCenters()
>>> rad = numerix.sqrt((x - .5)**2 + (y - .5)**2) - r
>>> distanceVariable = DistanceVariable(mesh = mesh, value = rad)
>>> print numerix.sum(distanceVariable.getCellInterfaceAreas())
1.57984690073
```

Inherited from fipy.variables.cellVariable.CellVariable: __call__(), __getstate__(), __setstate__(), copy(), getArithmeticFaceValue(), getCellVolumeAverage(), getFaceGrad(), getFaceGradAverage(), getFaceValue(), getGaussGrad(), getGrad(), getHarmonicFaceValue(), getLeastSquaresGrad(), getMinmodFaceValue(), getOld(), updateOld()

Inherited from fipy.variables.meshVariable._MeshVariable: __repr__(), dot(), getMesh(), getRank(), getShape(), rdot(), setValue()

Inherited from fipy.variables.variable.Variable: __abs__(), __add__(), __and__(), __array__(), __array_wrap__(), __div__(), __eq__(), __float__(), __ge__(), __getitem__(), __gt__(), __int__(), __iter__(), __le__(),

__len__(), __lt__(), __mod__(), __mul__(), __ne__(), __neg__(), __new__(), __nonzero__(), __or__(), __pos__(), __pow__(), __radd__(), __rdiv__(), __rmul__(), __rpow__(), __rsub__(), __setitem__(), __str__(), __sub__(), all(), allclose(), allequal(), any(), arccos(), arccosh(), arcsin(), arcsinh(), arctan(), arctan2(), arctanh(), cacheMe(), ceil(), conjugate(), cos(), cosh(), dontCacheMe(), exp(), floor(), getMag(), getName(), getNumericValue(), getSubscribedVariables(), getUnit(), getValue(), getsctype(), inBaseUnits(), inUnitsOf(), itemset(), log(), log10(), max(), min(), put(), reshape(), setName(), setUnit(), sign(), sin(), sinh(), sqrt(), sum(), take(), tan(), tanh(), tostring(), transpose()

Properties

Inherited from fipy.variables.variable.Variable: shape

Class Variables

Inherited from fipy.variables.variable.Variable: __array_priority__

4.9 Module fipy.models.levelSet.distanceFunction.levelSetDiffusionEquation

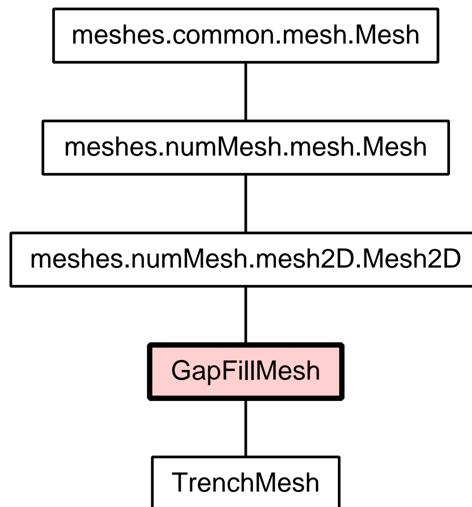
4.10 Module fipy.models.levelSet.distanceFunction.levelSetDiffusionVariable

4.11 Package fipy.models.levelSet.electroChem

4.12 Module fipy.models.levelSet.electroChem.gapFillMesh

The `GapFillMesh` object glues 3 meshes together to form a composite mesh. The first mesh is a `Grid2D` object that is fine and deals with the area around the trench or via. The second mesh is a `GmshImporter2D` object that forms a transition mesh from a fine to a course region. The third mesh is another `Grid2D` object that forms the boundary layer. This region consists of very large elements and is only used for the diffusion in the boundary layer.

Class GapFillMesh



Known Subclasses: fipy.models.levelSet.electroChem.gapFillMesh.TrenchMesh

The following test case tests for diffusion across the domain.

```
>>> domainHeight = 5.
>>> mesh = GapFillMesh(transitionRegionHeight = 2.,
...                    cellSize = 0.1,
...                    desiredFineRegionHeight = 1.,
...                    desiredDomainHeight = domainHeight,
...                    desiredDomainWidth = 1.)

>>> import fipy.tools.dump as dump
>>> (f, filename) = dump.write(mesh)
>>> mesh = dump.read(filename, f)
>>> mesh.getNumberOfCells() - len(mesh.getCellIDsAboveFineRegion())
90

>>> from fipy.variables.cellVariable import CellVariable
>>> var = CellVariable(mesh = mesh)

>>> from fipy.terms.diffusionTerm import DiffusionTerm
>>> eq = DiffusionTerm()

>>> from fipy.boundaryConditions.fixedValue import FixedValue
>>> eq.solve(var, boundaryConditions = (FixedValue(mesh.getFacesBottom(), 0.),
...                                     FixedValue(mesh.getFacesTop(), domainHeight)))
```

Evaluate the result:

```
>>> centers = mesh.getCellCenters()[1].copy() ## the copy makes the array contigous for inlining
>>> localErrors = (centers - var)**2 / centers**2
>>> globalError = numerix.sqrt(numerix.sum(localErrors) / mesh.getNumberOfCells())
>>> argmax = numerix.argmax(localErrors)
>>> print numerix.sqrt(localErrors[argmax]) < 0.1
1
>>> print globalError < 0.05
1
```

Methods

`__init__`(*self*, *cellSize*=None, *desiredDomainWidth*=None, *desiredDomainHeight*=None, *desiredFineRegionHeight*=None, *transitionRegionHeight*=None)

Arguments:

`cellSize` - The cell size in the fine grid around the trench.

`desiredDomainWidth` - The desired domain width.

`desiredDomainHeight` - The total desired height of the domain.

`desiredFineRegionHeight` - The desired height of the in the fine region around the trench.

`transitionRegionHeight` - The height of the transition region.

Overrides: fipy.meshes.common.mesh.Mesh.__init__()

`__getstate__`(*self*)

Overrides: fipy.meshes.numMesh.mesh.Mesh.__getstate__()

`__setstate__`(*self*, *dict*)

Overrides: fipy.meshes.numMesh.mesh.Mesh.__setstate__()

`buildTransitionMesh`(*self*, *nx*, *height*, *cellSize*)

`getCellIDsAboveFineRegion`(*self*)

getFineMesh(*self*)

Inherited from fipy.meshes.numMesh.mesh2D.Mesh2D: __mul__(), extrude()

Inherited from fipy.meshes.numMesh.mesh.Mesh: __add__(), __radd__(), __rmul__(), getExteriorFaces(), getFaceCellIDs(), getFaceCenters(), getInteriorFaces(), getVertexCoords()

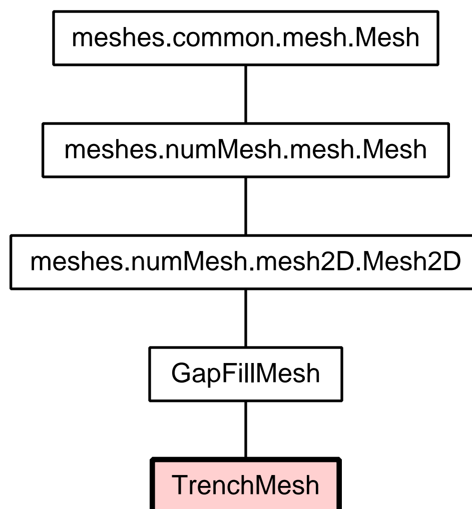
Inherited from fipy.meshes.common.mesh.Mesh: __repr__(), getCellCenters(), getCellVolumes(), getCells(), getDim(), getFaces(), getFacesBack(), getFacesBottom(), getFacesDown(), getFacesFront(), getFacesLeft(), getFacesRight(), getFacesTop(), getFacesUp(), getNearestCell(), getNumberOfCells(), setScale()

Instance Variables

Inherited from fipy.meshes.numMesh.mesh.Mesh: cellNormals

Inherited from fipy.meshes.common.mesh.Mesh: cellAreas, cellToCellIDsFilled

Class TrenchMesh



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.getNumberOfCells() - len(numerix.nonzero(mesh.getElectrolyteMask())[0])
150

>>> from fipy.variables.cellVariable import CellVariable
>>> var = CellVariable(mesh = mesh, value = 0.)

>>> from fipy.terms.diffusionTerm import DiffusionTerm
>>> eq = DiffusionTerm()

>>> from fipy.boundaryConditions.fixedValue import FixedValue

>>> eq.solve(var, boundaryConditions = (FixedValue(mesh.getFacesBottom(), 0.),
...                                     FixedValue(mesh.getFacesTop(), domainHeight)))
```

Evaluate the result:

```
>>> centers = mesh.getCellCenters()[1].copy() ## ensure contiguous array for inlining
>>> localErrors = (centers - var)**2 / centers**2
>>> globalError = numerix.sqrt(numerix.sum(localErrors) / mesh.getNumberOfCells())
>>> argmax = numerix.argmax(localErrors)
>>> print numerix.sqrt(localErrors[argmax]) < 0.051
1
>>> print globalError < 0.02
1
```

Methods

__init__(*self*, *trenchDepth*=None, *trenchSpacing*=None, *boundaryLayerDepth*=None, *cellSize*=None, *aspectRatio*=None, *angle*=0.0, *bowWidth*=0.0, *overBumpRadius*=0.0, *overBumpWidth*=0.0)

`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.

Overrides: fipy.meshes.common.mesh.Mesh.__init__()

getElectrolyteMask(*self*)

__getstate__(*self*)

Overrides: fipy.meshes.numMesh.mesh.Mesh.__getstate__()

__setstate__(*self*, *dict*)

Overrides: fipy.meshes.numMesh.mesh.Mesh.__setstate__()

getTopFaces(*self*)

Included to not break the interface

getBottomFaces(*self*)

Included to not break the interface

Inherited from fipy.models.levelSet.electroChem.gapFillMesh.GapFillMesh:
buildTransitionMesh(), getCellIDsAboveFineRegion(), getFineMesh()

Inherited from fipy.meshes.numMesh.mesh2D.Mesh2D: __mul__(), extrude()

Inherited from fipy.meshes.numMesh.mesh.Mesh: __add__(), __radd__(), __rmul__(), getExteriorFaces(), getFaceCellIDs(), getFaceCenters(), getInteriorFaces(), getVertexCoords()

Inherited from fipy.meshes.common.mesh.Mesh: __repr__(), getCellCenters(), getCellVolumes(), getCells(), getDim(), getFaces(), getFacesBack(), getFacesBottom(), getFacesDown(), getFacesFront(), getFacesLeft(), getFacesRight(), getFacesTop(), getFacesUp(), getNearestCell(), getNumberOfCells(), setScale()

Instance Variables

Inherited from fipy.meshes.numMesh.mesh.Mesh: cellNormals

Inherited from fipy.meshes.common.mesh.Mesh: cellAreas, cellToCellIDsFilled

4.13 Module fipy.models.levelSet.electroChem.metalIonDiffusionEquation

Functions

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,

$$D = D_c \text{ when } \phi > 0$$

$$D = 0 \text{ when } \phi \leq 0$$

The velocity of the interface generally has a linear dependence on ion concentration. The following boundary condition applies at the zero level set,

$$D \hat{n} \cdot \nabla c = \frac{v(c)}{\Omega} \text{ at } \phi = 0$$

where

$$v(c) = c V_0$$

The test case below is for a 1D steady state problem. The solution is given by:

$$c(x) = \frac{c^\infty}{\Omega D / V_0 + L} (x - L) + c^\infty$$

This is the test case,

```
>>> from fipy.meshes.grid1D import Grid1D
>>> nx = 11
>>> dx = 1.
>>> mesh = Grid1D(nx = nx, dx = dx)
>>> x, = mesh.getCellCenters()
>>> from fipy.variables.cellVariable import CellVariable
>>> ionVar = CellVariable(mesh = mesh, value = 1.)
>>> from fipy.models.levelSet.distanceFunction.distanceVariable \
...  import DistanceVariable
>>> disVar = DistanceVariable(mesh = mesh,
...  value = (x - 0.5) - 0.99,
...  hasOld = 1)

>>> v = 1.
>>> diffusion = 1.
>>> omega = 1.
```

```
>>> cinf = 1.
>>> from fipy.boundaryConditions.fixedValue import FixedValue
>>> eqn = buildMetalIonDiffusionEquation(ionVar = ionVar,
...  distanceVar = disVar,
...  depositionRate = v * ionVar,
...  diffusionCoeff = diffusion,
...  metalIonMolarVolume = omega)
>>> bc = (FixedValue(mesh.getFacesRight(), cinf),)
>>> for i in range(10):
...  eqn.solve(ionVar, dt = 1000, boundaryConditions = bc)
>>> L = (nx - 1) * dx - dx / 2
>>> gradient = cinf / (omega * diffusion / v + L)
>>> answer = gradient * (x - L - dx * 3 / 2) + cinf
>>> answer[0] = 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.

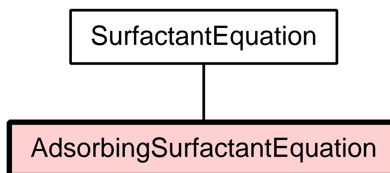
4.14 Module fipy.models.levelSet.electroChem.metalIonSourceVariable

4.15 Module fipy.models.levelSet.electroChem.test

4.16 Package fipy.models.levelSet.surfactant

4.17 Module fipy.models.levelSet.surfactant.adsorbingSurfactantEquation

Class AdsorbingSurfactantEquation



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 $S_c = kc(1 - \theta_{\text{other}})$ 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.grid2D import Grid2D
>>> dx = .5
>>> dy = 2.3
>>> dt = 0.25
>>> k = 0.56
>>> initialValue = 0.1
>>> c = 0.2

>>> from fipy.meshes.grid2D import Grid2D
>>> mesh = Grid2D(dx = dx, dy = dy, nx = 5, ny = 1)
>>> distanceVar = DistanceVariable(mesh = mesh,
```

```
...                               value = (-dx*3/2, -dx/2, dx/2,
...                                         3*dx/2,  5*dx/2),
...                               hasOld = 1)
>>> surfactantVar = SurfactantVariable(value = (0, 0, initialValue, 0 ,0),
...                                    distanceVar = distanceVar)
>>> bulkVar = CellVariable(mesh = mesh, value = (c , c, c, c, c))
>>> eqn = AdsorbingSurfactantEquation(surfactantVar = surfactantVar,
...                                   distanceVar = distanceVar,
...                                   bulkVar = bulkVar,
...                                   rateConstant = k)
>>> eqn.solve(surfactantVar, dt = dt)
>>> answer = (initialValue + dt * k * c) / (1 + dt * k * c)
>>> print numerix.allclose(surfactantVar.getInterfaceVar(),
...                  numerix.array((0, 0, answer, 0, 0)))
1
```

The following test case is for two surfactant variables. One has more surface affinity than the other.

```
>>> from fipy.models.levelSet.distanceFunction.distanceVariable \
...     import DistanceVariable
>>> from fipy.models.levelSet.surfactant.surfactantVariable \
...     import SurfactantVariable
>>> from fipy.meshes.grid2D import Grid2D
>>> dx = 0.5
>>> dy = 2.73
>>> dt = 0.001
>>> k0 = 1.
>>> k1 = 10.
>>> theta0 = 0.
>>> theta1 = 0.
>>> c0 = 1.
>>> c1 = 1.
>>> totalSteps = 100
>>> mesh = Grid2D(dx = dx, dy = dy, nx = 5, ny = 1)
>>> distanceVar = DistanceVariable(mesh = mesh,
...                               value = dx * (numerix.arange(5) - 1.5),
...                               hasOld = 1)
>>> var0 = SurfactantVariable(value = (0, 0, theta0, 0 ,0),
...                          distanceVar = distanceVar)
>>> var1 = SurfactantVariable(value = (0, 0, 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.getInterfaceVar(),
...                  numerix.array((0, 0, answer0, 0, 0)), rtol = 1e-2)
1
>>> print numerix.allclose(var1.getInterfaceVar(),
...                  numerix.array((0, 0, answer1, 0, 0)), rtol = 1e-2)
1
>>> dt = 0.1
>>> for step in range(10):
...     eqn0.solve(var0, dt = dt)
...     eqn1.solve(var1, dt = dt)
>>> print var0.getInterfaceVar()[2] + var1.getInterfaceVar()[2]
1.0
```

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)
>>> print numerix.allclose(var0.getInterfaceVar()[2], 0)
1
```

The following test case is to fix a bug that allows the accelerator to become negative.

```
>>> nx = 5
>>> ny = 5
>>> mesh = Grid2D(dx = 1., dy = 1., nx = nx, ny = ny)
```

```
>>> values = numerix.ones(mesh.getNumberOfCells(), 'd')
>>> values[0:nx] = -1
>>> for i in range(ny):
...     values[i * nx] = -1

>>> disVar = DistanceVariable(mesh = mesh, value = values, hasOld = 1)
>>> disVar.calcDistanceFunction()

>>> levVar = SurfactantVariable(value = 0.5, distanceVar = disVar)
>>> accVar = SurfactantVariable(value = 0.5, distanceVar = disVar)

>>> levEq = AdsorbingSurfactantEquation(levVar,
...                                     distanceVar = disVar,
...                                     bulkVar = 0,
...                                     rateConstant = 0)

>>> accEq = AdsorbingSurfactantEquation(accVar,
...                                     distanceVar = disVar,
...                                     bulkVar = 0,
...                                     rateConstant = 0,
...                                     otherVar = levVar,
...                                     otherBulkVar = 0,
...                                     otherRateConstant = 0)

>>> extVar = CellVariable(mesh = mesh, value = accVar.getInterfaceVar())

>>> from fipy.models.levelSet.advection.higherOrderAdvectionEquation \
...     import buildHigherOrderAdvectionEquation
>>> advEq = buildHigherOrderAdvectionEquation(advectionCoeff = extVar)

>>> dt = 0.1

>>> for i in range(50):
...     disVar.calcDistanceFunction()
...     extVar.setValue(numerix.array(accVar.getInterfaceVar()))
...     disVar.extendVariable(extVar)
...     disVar.updateOld()
...     advEq.solve(disVar, dt = dt)
...     levEq.solve(levVar, dt = dt)
...     accEq.solve(accVar, dt = dt)

>>> print numerix.sum(accVar < -1e-10) == 0
1
```

Methods

`__init__`(*self*, *surfactantVar*=None, *distanceVar*=None, *bulkVar*=None, *rateConstant*=None, *otherVar*=None, *otherBulkVar*=None, *otherRateConstant*=None, *consumptionCoeff*=None)

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.

Overrides: fipy.models.levelSet.surfactant.surfactantEquation.SurfactantEquation.__init__()

`solve`(*self*, *var*, *boundaryConditions*=(), *solver*=LinearPCGSolver(tolerance=1e-10, iterations=1000), *dt*=1.0)

Builds and solves the `AdsorbingSurfactantEquation`'s linear system once.

Parameters

var: A `SurfactantVariable` to be solved for. Provides the initial condition, the old value and holds the solution on completion.

solver: The iterative solver to be used to solve the linear system of equations.

boundaryConditions: A tuple of boundaryConditions.

dt: The time step size.

Overrides: fipy.models.levelSet.surfactant.surfactantEquation.SurfactantEquation.solve()

`sweep`(*self*, *var*, *solver*=`LinearLUSolver(tolerance=1e-10, iterations=10)`, *boundaryConditions*=`()`, *dt*=`1.0`, *underRelaxation*=`None`, *residualFn*=`None`)

Builds and solves the `AdsorbingSurfactantEquation`'s linear system once. This method also recalculates and returns the residual as well as applying under-relaxation.

Parameters

var: The variable to be solved for. Provides the initial condition, the old value and holds the solution on completion.

solver: The iterative solver to be used to solve the linear system of equations.

boundaryConditions: A tuple of boundaryConditions.

dt: The time step size.

underRelaxation: Usually a value between `0` and `1` or `None` in the case of no under-relaxation

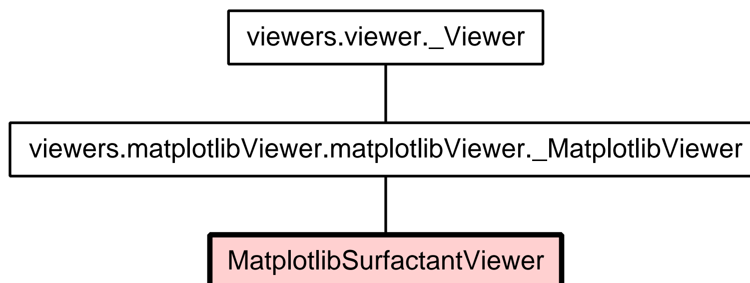
Overrides: fipy.models.levelSet.surfactant.surfactantEquation.SurfactantEquation.sweep()

4.18 Module fipy.models.levelSet.surfactant.convectionCoeff

4.19 Module fipy.models.levelSet.surfactant.lines

4.20 Module fipy.models.levelSet.surfactant.matplotlibSurfactantViewer

Class MatplotlibSurfactantViewer



The `MatplotlibSurfactantViewer` creates a viewer with the Matplotlib python plotting package that displays a `DistanceVariable`.

Methods

__init__(*self*, *distanceVar*, *surfactantVar*=None, *levelSetValue*=0.0, *title*=None, *smooth*=0, *zoomFactor*=1.0, *animate*=False, *limits*={}, ***kwlimits*)

Create a `MatplotlibSurfactantViewer`.

```
>>> from fipy import *
>>> m = Grid2D(nx=100, ny=100)
>>> x, y = m.getCellCenters()
>>> v = CellVariable(mesh=m, value=x**2 + y**2 - 10**2)
>>> s = CellVariable(mesh=m, value=sin(x / 10) * cos(y / 30))
>>> viewer = MatplotlibSurfactantViewer(distanceVar=v, surfactantVar=s)
>>> for r in range(1,200):
...  v.setValue(x**2 + y**2 - r**2)
...  viewer.plot()

>>> from fipy import *
>>> dx = 1.
>>> dy = 1.
>>> nx = 11
>>> ny = 11
>>> Lx = ny * dy
>>> Ly = nx * dx
>>> mesh = Grid2D(dx = dx, dy = dy, nx = nx, ny = ny)
>>> # from fipy.models.levelSet.distanceFunction.distanceVariable import DistanceVariable
>>> var = DistanceVariable(mesh = mesh, value = -1)

>>> x, y = mesh.getCellCenters()

>>> var.setValue(1, where=(x - Lx / 2.)**2 + (y - Ly / 2.)**2 < (Lx / 4.)**2)
>>> var.calcDistanceFunction()
>>> viewer = MatplotlibSurfactantViewer(var, smooth = 2)
>>> viewer.plot()
>>> viewer._promptForOpinion()
>>> del viewer

>>> var = DistanceVariable(mesh = mesh, value = -1)

>>> var.setValue(1, where=(y > 2. * Ly / 3.) | ((x > Lx / 2.) & (y > Ly / 3.)) |
((y < Ly / 6.) & (x > Lx / 2)))
>>> var.calcDistanceFunction()
>>> viewer = MatplotlibSurfactantViewer(var)
>>> viewer.plot()
>>> viewer._promptForOpinion()
>>> del viewer

>>> viewer = MatplotlibSurfactantViewer(var, smooth = 2)
>>> viewer.plot()
>>> viewer._promptForOpinion()
>>> del viewer
```

Parameters

distanceVar: 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)

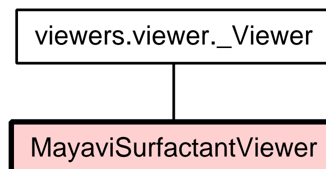
Overrides: fipy.viewers.viewer._Viewer.__init__()

Inherited from fipy.viewers.matplotlibViewer.matplotlibViewer._MatplotlibViewer: plot()

Inherited from fipy.viewers.viewer._Viewer: getVars(), plotMesh(), setLimits()

4.21 Module fipy.models.levelSet.surfactant.mayaviSurfactantViewer

Class MayaviSurfactantViewer



The `MayaviSurfactantViewer` creates a viewer with the Mayavi python plotting package that displays a `DistanceVariable`.

Methods

`__init__`(*self*, *distanceVar*, *surfactantVar*=`None`, *levelSetValue*=`0.0`, *title*=`None`, *smooth*=`0`, *zoomFactor*=`1.0`, *animate*=`False`, *limits*={}, ***kwlimits*)

Create a `MayaviSurfactantViewer`.

```
>>> from fipy import *
>>> dx = 1.
>>> dy = 1.
>>> nx = 11
>>> ny = 11
>>> Lx = ny * dy
>>> Ly = nx * dx
>>> mesh = Grid2D(dx = dx, dy = dy, nx = nx, ny = ny)
>>> # from fipy.models.levelSet.distanceFunction.distanceVariable import DistanceVariable
>>> var = DistanceVariable(mesh = mesh, value = -1)

>>> x, y = mesh.getCellCenters()

>>> var.setValue(1, where=(x - Lx / 2.)**2 + (y - Ly / 2.)**2 < (Lx / 4.)**2)
>>> var.calcDistanceFunction()
>>> viewer = MayaviSurfactantViewer(var, smooth = 2)
>>> viewer.plot()
>>> viewer._promptForOpinion()
>>> del viewer

>>> var = DistanceVariable(mesh = mesh, value = -1)

>>> var.setValue(1, where=(y > 2. * Ly / 3.) | ((x > Lx / 2.) & (y > Ly / 3.)) |
((y < Ly / 6.) & (x > Lx / 2)))
>>> var.calcDistanceFunction()
>>> viewer = MayaviSurfactantViewer(var)
>>> viewer.plot()
>>> viewer._promptForOpinion()
>>> del viewer

>>> viewer = MayaviSurfactantViewer(var, smooth = 2)
>>> viewer.plot()
>>> viewer._promptForOpinion()
>>> del viewer
```

Parameters

distanceVar: 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)

Overrides: fipy.viewers.viewer._Viewer.__init__()

`plot`(*self*, *filename*=`None`)

Update the display of the viewed variables.

Parameters

filename: If not `None`, the name of a file to save the image into.

Overrides: fipy.viewers.viewer._Viewer.plot() *(inherited documentation)*

Inherited from fipy.viewers.viewer._Viewer: getVars(), plotMesh(), setLimits()

4.22 Module fipy.models.levelSet.surfactant.surfactantBulkDiffusionEquation

Functions

`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,

$$D = D_c \text{ when } \phi > 0$$

$$D = 0 \text{ when } \phi \leq 0$$

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

$$D \hat{n} \cdot \nabla c = -kc(1 - \theta) \text{ at } \phi = 0.$$

Parameters

bulkVar: The bulk surfactant concentration variable.

distanceVar: A `DistanceVariable` object

surfactantVar: A `SurfactantVariable` object

otherSurfactantVar: Any other surfactants that may remove this one.

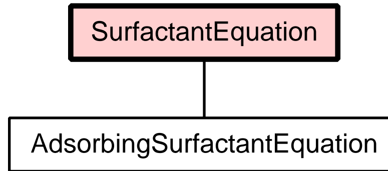
diffusionCoeff: A float or a `FaceVariable`.

transientCoeff: In general 1 is used.

rateConstant: The adsorption coefficient.

4.23 Module fipy.models.levelSet.surfactant.surfactantEquation

Class SurfactantEquation



Known Subclasses:
fipy.models.levelSet.surfactant.adsorbingSurfactantEquation.AdsorbingSurfactantEquation

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.

Methods

`__init__`(*self*, *distanceVar*=`None`)

Creates a `SurfactantEquation` object.

Parameters

distanceVar: The `DistanceVariable` that marks the interface.

`solve`(*self*, *var*, *boundaryConditions*=`()`, *solver*=`LinearLUSolver(tolerance=1e-10, iterations=10)`, *dt*=`1.0`)

Builds and solves the `SurfactantEquation`'s linear system once.

Parameters

var: A `SurfactantVariable` to be solved for. Provides the initial condition, the old value and holds the solution on completion.

solver: The iterative solver to be used to solve the linear system of equations.

boundaryConditions: A tuple of boundaryConditions.

dt: The time step size.

`sweep`(*self*, *var*, *solver*=`LinearLUSolver(tolerance=1e-10, iterations=10)`, *boundaryConditions*=`()`, *dt*=`1.0`, *underRelaxation*=`None`, *residualFn*=`None`)

Builds and solves the `Term`'s linear system once. This method also recalculates and returns the residual as well as applying under-relaxation.

Parameters

var: The variable to be solved for. Provides the initial condition, the old value and holds the solution on completion.

solver: The iterative solver to be used to solve the linear system of equations.

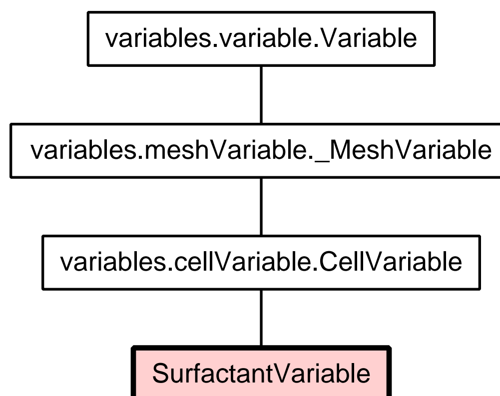
boundaryConditions: A tuple of boundaryConditions.

dt: The time step size.

underRelaxation: Usually a value between `0` and `1` or `None` in the case of no under-relaxation

4.24 Module fipy.models.levelSet.surfactant.surfactantVariable

Class SurfactantVariable



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).

Methods

__init__(*self*, *value*=0.0, *distanceVar*=None, *name*='surfactant variable', *hasOld*=False)

A simple 1D test:

```
>>> from fipy.meshes.grid1D import Grid1D
>>> mesh = Grid1D(dx = 1., nx = 4)
>>> from fipy.models.levelSet.distanceFunction.distanceVariable \
...  import DistanceVariable
>>> distanceVariable = DistanceVariable(mesh = mesh,
...  value = (-1.5, -0.5, 0.5, 941.5))
>>> surfactantVariable = SurfactantVariable(value = 1,
...  distanceVar = distanceVariable)
>>> print numerix.allclose(surfactantVariable, (0, 0., 1., 0))
1
```

A 2D test case:

```
>>> from fipy.meshes.grid2D import Grid2D
>>> mesh = Grid2D(dx = 1., dy = 1., nx = 3, ny = 3)
>>> distanceVariable = DistanceVariable(mesh = mesh,
...  value = (1.5, 0.5, 1.5,
...  0.5,-0.5, 0.5,
...  1.5, 0.5, 1.5))
>>> surfactantVariable = SurfactantVariable(value = 1,
...  distanceVar = distanceVariable)
>>> print numerix.allclose(surfactantVariable, (0, 1, 0, 1, 0, 1, 0, 1, 0))
1
```

Another 2D test case:

```
>>> mesh = Grid2D(dx = .5, dy = .5, nx = 2, ny = 2)
>>> distanceVariable = DistanceVariable(mesh = mesh,
...  value = (-0.5, 0.5, 0.5, 1.5))
>>> surfactantVariable = SurfactantVariable(value = 1,
...  distanceVar = distanceVariable)
>>> print numerix.allclose(surfactantVariable,
...  (0, numerix.sqrt(2), numerix.sqrt(2), 0))
1
```

Parameters

value: The initial value.

distanceVar: A `DistanceVariable` object.

name: The name of the variable.

Overrides: fipy.variables.variable.Variable.__init__()

getInterfaceVar(*self*)

Returns the `SurfactantVariable` rendered as an `_InterfaceSurfactantVariable` which evaluates the surfactant concentration as an area concentration the interface rather than a volumetric concentration.

copy(*self*)

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]))
```

Overrides: fipy.variables.variable.Variable.copy() *(inherited documentation)*

Inherited from fipy.variables.cellVariable.CellVariable: __call__(), __getstate__(), __setstate__(), getArithmeticFaceValue(), getCellVolumeAverage(), getFaceGrad(), getFaceGradAverage(), getFaceValue(), getGaussGrad(), getGrad(), getHarmonicFaceValue(), getLeastSquaresGrad(), getMinmodFaceValue(), getOld(), updateOld()

Inherited from fipy.variables.meshVariable._MeshVariable: __repr__(), dot(), getMesh(), getRank(), getShape(), rdot(), setValue()

Inherited from fipy.variables.variable.Variable: __abs__(), __add__(), __and__(), __array__(), __array_wrap__(), __div__(), __eq__(), __float__(), __ge__(), __getitem__(), __gt__(), __int__(), __iter__(), __le__(),

__len__(), __lt__(), __mod__(), __mul__(), __ne__(), __neg__(), __new__(), __nonzero__(), __or__(), __pos__(), __pow__(), __radd__(), __rdiv__(), __rmul__(), __rpow__(), __rsub__(), __setitem__(), __str__(), __sub__(), all(), allclose(), allequal(), any(), arccos(), arccosh(), arcsin(), arcsinh(), arctan(), arctan2(), arctanh(), cacheMe(), ceil(), conjugate(), cos(), cosh(), dontCacheMe(), exp(), floor(), getMag(), getName(), getNumericValue(), getSubscribedVariables(), getUnit(), getValue(), getsctype(), inBaseUnits(), inUnitsOf(), itemset(), log(), log10(), max(), min(), put(), reshape(), setName(), setUnit(), sign(), sin(), sinh(), sqrt(), sum(), take(), tan(), tanh(), tostring(), transpose()

Properties

Inherited from fipy.variables.variable.Variable: shape

Class Variables

Inherited from fipy.variables.variable.Variable: __array_priority__

4.25 Module fipy.models.levelSet.test

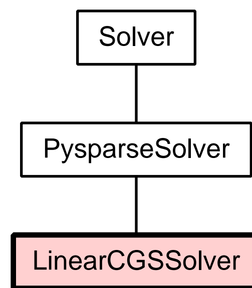
4.26 Module fipy.models.test

Package fipy.solvers

5.1 Package fipy.solvers.pysparse

5.2 Module fipy.solvers.pysparse.linearCGSSolver

Class LinearCGSSolver



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.

Methods

`__init__`(*self*, **args*, ***kwargs*)

Create a `Solver` object.

Parameters

tolerance: The required error tolerance.

iterations: The maximum number of iterative steps to perform.

steps: A deprecated name for `iterations`.

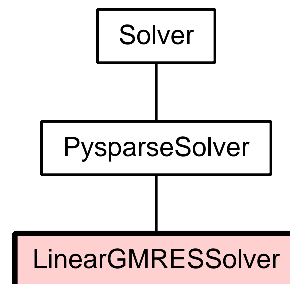
precon: Preconditioner to use. This parameter is only available for Trilinos solvers.

Overrides: fipy.solvers.solver.Solver.__init__() *(inherited documentation)*

Inherited from fipy.solvers.solver.Solver: __repr__()

5.3 Module fipy.solvers.pysparse.linearGMRESSolver

Class LinearGMRESSolver



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.

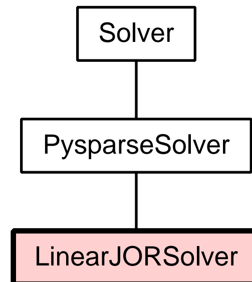
Methods

Inherited from fipy.solvers.pysparse.pysparseSolver.PysparseSolver: __init__()

Inherited from fipy.solvers.solver.Solver: __repr__()

5.4 Module fipy.solvers.pysparse.linearJORSolver

Class LinearJORSolver



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

Methods

`__init__`(*self*, *tolerance*=`1e-10`, *iterations*=`1000`, *steps*=`None`, *relaxation*=`1.0`, *precon*=`None`)

The `Solver` class should not be invoked directly.

Parameters

tolerance: The required error tolerance.

iterations: The maximum number of iterative steps to perform.

steps: A deprecated name for `iterations`.

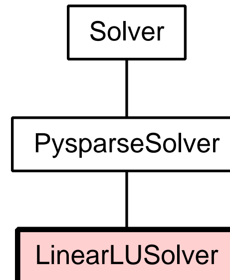
relaxation: The relaxation.

Overrides: fipy.solvers.solver.Solver.`__init__`()

Inherited from fipy.solvers.solver.Solver: `__repr__`()

5.5 Module fipy.solvers.pysparse.linearLUSolver

Class LinearLUSolver



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.

Methods

`__init__`(*self*, *tolerance*=`1e-10`, *iterations*=`10`, *steps*=`None`, *precon*=`None`)

Creates a `LinearLUSolver`.

Parameters

tolerance: The required error tolerance.

iterations: The number of LU decompositions to perform.

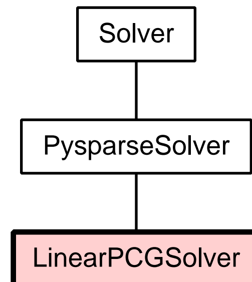
steps: A deprecated name for `iterations`. For large systems a number of iterations is generally required.

Overrides: fipy.solvers.solver.Solver.__init__()

Inherited from fipy.solvers.solver.Solver: __repr__()

5.6 Module fipy.solvers.pysparse.linearPCGSolver

Class LinearPCGSolver



The `LinearPCGSolver` solves a linear system of equations using the preconditioned conjugate gradient method (PCG) with symmetric successive over-relaxation (SSOR) preconditioning. The PCG method solves systems with a symmetric positive definite coefficient matrix.

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

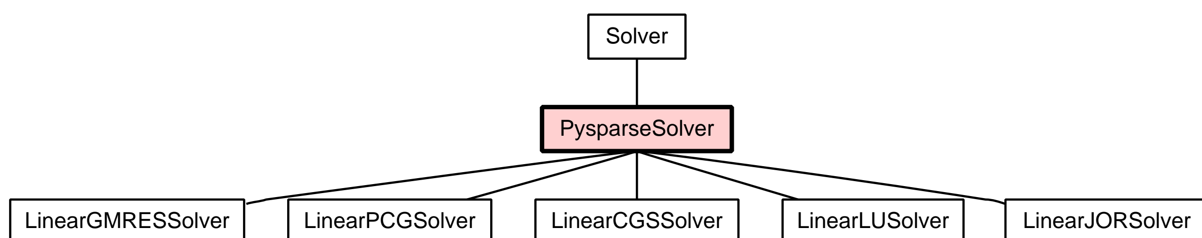
Methods

Inherited from fipy.solvers.pysparse.pysparseSolver.PysparseSolver: __init__()

Inherited from fipy.solvers.solver.Solver: __repr__()

5.7 Module fipy.solvers.pysparse.pysparseSolver

Class PysparseSolver



Known Subclasses: fipy.solvers.pysparse.linearGMRESSolver.LinearGMRESSolver, fipy.solvers.pysparse.linearPCGSolver.LinearPCGSolver, fipy.solvers.pysparse.linearCGSSolver.LinearCGSSolver, fipy.solvers.pysparse.linearLUSolver.LinearLUSolver, fipy.solvers.pysparse.linearJORSolver.LinearJORSolver

The base `pysparseSolver` class.

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

Methods

`__init__`(*self*, **args*, ***kwargs*)

Create a `Solver` object.

Parameters

tolerance: The required error tolerance.

iterations: The maximum number of iterative steps to perform.

steps: A deprecated name for `iterations`.

precon: Preconditioner to use. This parameter is only available for Trilinos solvers.

Overrides: fipy.solvers.solver.Solver.__init__() *(inherited documentation)*

Inherited from fipy.solvers.solver.Solver: __repr__()

5.8 Module fipy.solvers.solver

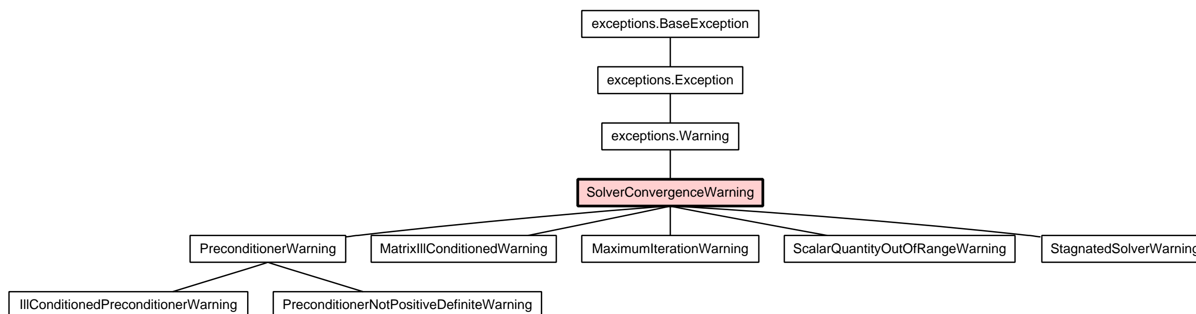
The iterative solvers may output warnings if the solution is considered unsatisfactory. If you are not interested in these warnings, you can invoke python with a warning filter such as:

```
$ python -Wignore::fipy.SolverConvergenceWarning myscript.py
```

If you are extremely concerned about your preconditioner for some reason, you can abort whenever it has problems with:

```
$ python -Werror::fipy.PreconditionerWarning myscript.py
```

Class SolverConvergenceWarning



Known Subclasses: fipy.solvers.solver.PreconditionerWarning, fipy.solvers.solver.MatrixIllConditionedWarning, fipy.solvers.solver.MaximumIterationWarning, fipy.solvers.solver.ScalarQuantityOutOfRangeWarning, fipy.solvers.solver.StagnatedSolverWarning

Methods

`__init__`(*self*, *solver*, *iter*, *relres*)

x.__init__(...) initializes x; see x.__class__.__doc__ for signature

Overrides: exceptions.BaseException.__init__() *(inherited documentation)*

`__str__`(*self*)

str(x)

Overrides: exceptions.BaseException.__str__() *(inherited documentation)*

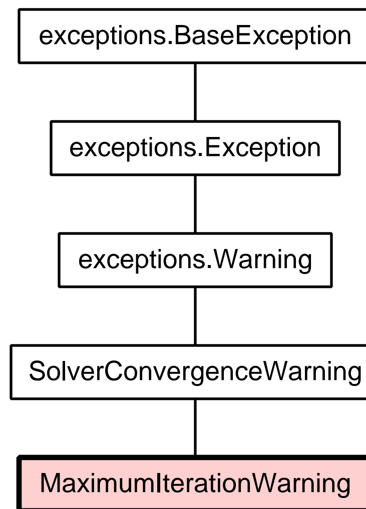
Inherited from exceptions.Warning: __new__()

Inherited from exceptions.BaseException: __delattr__(), __getattribute__(), __getitem__(), __reduce__(), __repr__(), __setattr__(), __setstate__()

Properties

Inherited from exceptions.BaseException: args, message

Class MaximumIterationWarning



Methods

`__str__(self)`

str(x)

Overrides: exceptions.BaseException.__str__() *(inherited documentation)*

Inherited from fipy.solvers.solver.SolverConvergenceWarning: __init__()

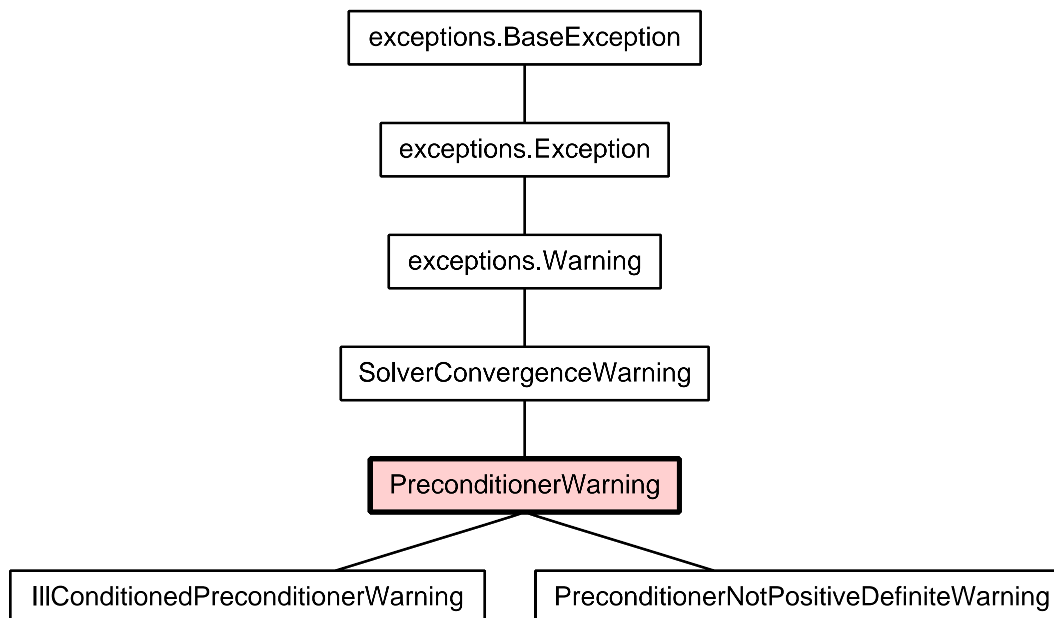
Inherited from exceptions.Warning: __new__()

Inherited from exceptions.BaseException: __delattr__(), __getattribute__(), __getitem__(), __reduce__(), __repr__(), __setattr__(), __setstate__()

Properties

Inherited from exceptions.BaseException: args, message

Class PreconditionerWarning



Known Subclasses: fipy.solvers.solver.IllConditionedPreconditionerWarning, fipy.solvers.solver.PreconditionerNotPositiveDefiniteWarning

Methods

Inherited from fipy.solvers.solver.SolverConvergenceWarning: __init__(), __str__()

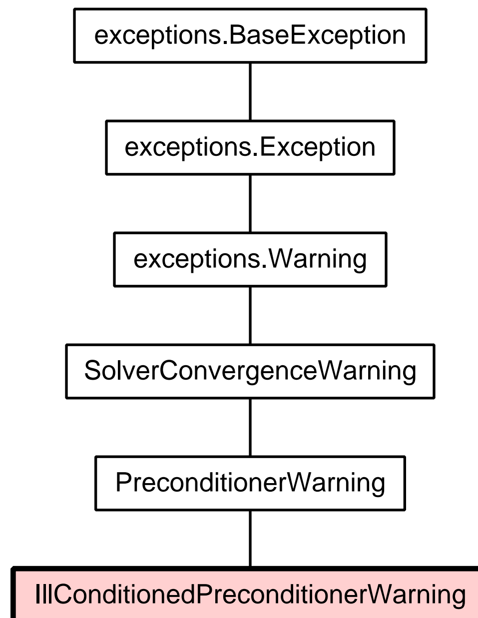
Inherited from exceptions.Warning: __new__()

Inherited from exceptions.BaseException: __delattr__(), __getattribute__(), __getitem__(), __reduce__(), __repr__(), __setattr__(), __setstate__()

Properties

Inherited from exceptions.BaseException: args, message

Class IllConditionedPreconditionerWarning



Methods

`__str__(self)`

str(x)

Overrides: exceptions.BaseException.__str__() *(inherited documentation)*

Inherited from fipy.solvers.solver.SolverConvergenceWarning: __init__()

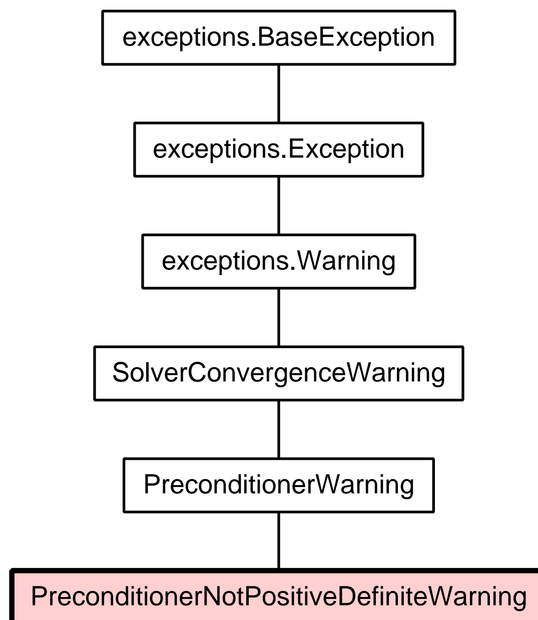
Inherited from exceptions.Warning: __new__()

Inherited from exceptions.BaseException: __delattr__(), __getattribute__(), __getitem__(), __reduce__(), __repr__(), __setattr__(), __setstate__()

Properties

Inherited from exceptions.BaseException: args, message

Class PreconditionerNotPositiveDefiniteWarning



Methods

`__str__`(*self*)

---

str(x)

Overrides: exceptions.BaseException.__str__() *(inherited documentation)*

Inherited from fipy.solvers.solver.SolverConvergenceWarning: __init__()

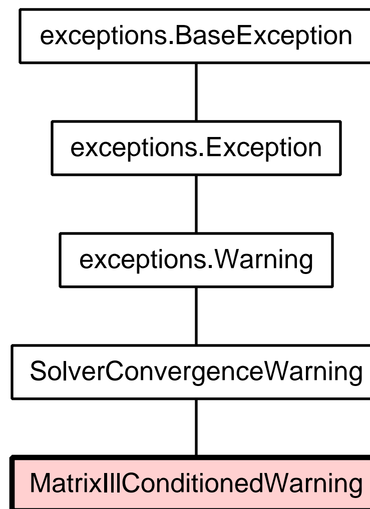
Inherited from exceptions.Warning: __new__()

Inherited from exceptions.BaseException: __delattr__(), __getattribute__(), __getitem__(), __reduce__(), __repr__(), __setattr__(), __setstate__()

Properties

Inherited from exceptions.BaseException: args, message

Class MatrixIllConditionedWarning



Methods

`__str__(self)`

---

str(x)

Overrides: exceptions.BaseException.__str__() *(inherited documentation)*

Inherited from fipy.solvers.solver.SolverConvergenceWarning: __init__()

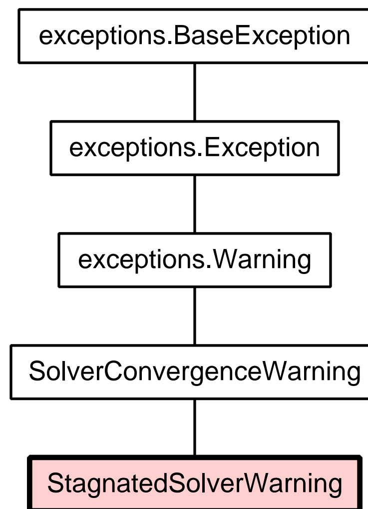
Inherited from exceptions.Warning: __new__()

Inherited from exceptions.BaseException: __delattr__(), __getattribute__(), __getitem__(), __reduce__(), __repr__(), __setattr__(), __setstate__()

Properties

Inherited from exceptions.BaseException: args, message

Class StagnatedSolverWarning



Methods

`__str__`(*self*)

---

str(x)

Overrides: exceptions.BaseException.__str__() *(inherited documentation)*

Inherited from fipy.solvers.solver.SolverConvergenceWarning: __init__()

Inherited from exceptions.Warning: __new__()

Inherited from exceptions.BaseException: __delattr__(), __getattribute__(), __getitem__(), __reduce__(), __repr__(), __setattr__(), __setstate__()

Properties

Inherited from exceptions.BaseException: args, message

Class ScalarQuantityOutOfRangeWarning

exceptions.BaseException
exceptions.Exception
exceptions.Warning
SolverConvergenceWarning
ScalarQuantityOutOfRangeWarning

Methods

__str__(*self*)

str(x)

Overrides: exceptions.BaseException.__str__() *(inherited documentation)*

Inherited from fipy.solvers.solver.SolverConvergenceWarning: __init__()

Inherited from exceptions.Warning: __new__()

Inherited from exceptions.BaseException: __delattr__(), __getattribute__(), __getitem__(), __reduce__(), __repr__(), __setattr__(), __setstate__()

Properties

Inherited from exceptions.BaseException: args, message

Class Solver

Solver
trilinos.trilinosSolver.TrilinosSolver
pysparse.pysparseSolver.PysparseSolver
trilinos.trilinosAztecOOSolver.TrilinosAztecOOSolver
trilinos.linearLUSolver.LinearLUSolver
trilinos.trilinosMLTest.TrilinosMLTest
pysparse.linearGMRESSolver.LinearGMRESSolver
pysparse.linearPCGSolver.LinearPCGSolver
pysparse.linearCGSSolver.LinearCGSSolver
...
trilinos.linearCGSSolver.LinearCGSSolver
trilinos.linearBicgstabSolver.LinearBicgstabSolver
...

Known Subclasses: fipy.solvers.trilinos.trilinosSolver.TrilinosSolver, fipy.solvers.pysparse.pysparseSolver.PysparseSolver

The base `LinearXSolver` class.

Attention!

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

Methods

`__init__`(*self*, *tolerance*=`1e-10`, *iterations*=`1000`, *steps*=`None`, *precon*=`None`)

Create a `Solver` object.

Parameters

tolerance: The required error tolerance.

iterations: The maximum number of iterative steps to perform.

steps: A deprecated name for `iterations`.

precon: Preconditioner to use. This parameter is only available for Trilinos solvers.

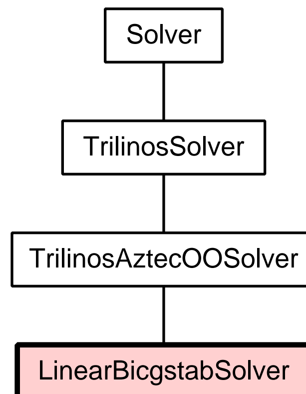
`__repr__`(*self*)

5.9 Module fipy.solvers.test

5.10 Package fipy.solvers.trilinos

5.11 Module fipy.solvers.trilinos.linearBicgstabSolver

Class LinearBicgstabSolver



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

Methods

`__init__`(*self*, *tolerance*=`1e-10`, *iterations*=`1000`, *steps*=`None`, *precon*=`JacobiPreconditioner()`)

Create a `Solver` object.

Parameters

tolerance: The required error tolerance.

iterations: The maximum number of iterative steps to perform.

steps: A deprecated name for `iterations`.

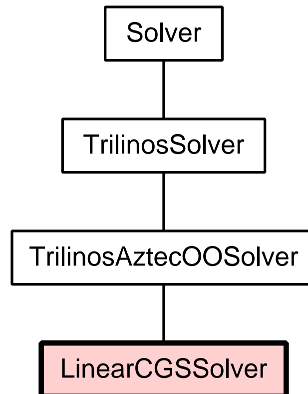
precon: Preconditioner to use.

Overrides: fipy.solvers.solver.Solver.__init__()

Inherited from fipy.solvers.solver.Solver: __repr__()

5.12 Module fipy.solvers.trilinos.linearCGSSolver

Class LinearCGSSolver



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

Methods

`__init__`(*self*, *tolerance*=`1e-10`, *iterations*=`1000`, *steps*=`None`, *precon*=`MultilevelSGSPreconditioner()`)

Create a `Solver` object.

Parameters

tolerance: The required error tolerance.

iterations: The maximum number of iterative steps to perform.

steps: A deprecated name for `iterations`.

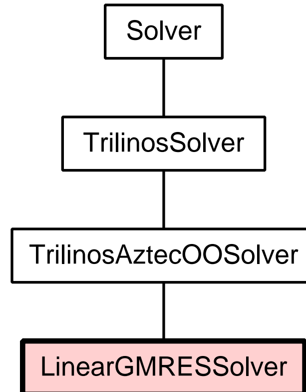
precon: Preconditioner to use.

Overrides: fipy.solvers.solver.Solver.__init__()

Inherited from fipy.solvers.solver.Solver: __repr__()

5.13 Module fipy.solvers.trilinos.linearGMRESSolver

Class LinearGMRESSolver



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

Methods

`__init__`(*self*, *tolerance*=`1e-10`, *iterations*=`1000`, *steps*=`None`, *precon*=`MultilevelDDPreconditioner()`)

Create a `Solver` object.

Parameters

tolerance: The required error tolerance.

iterations: The maximum number of iterative steps to perform.

steps: A deprecated name for `iterations`.

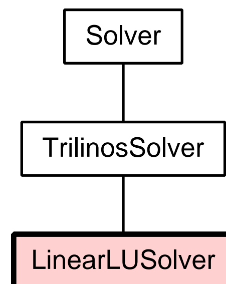
precon: Preconditioner to use.

Overrides: fipy.solvers.solver.Solver.__init__()

Inherited from fipy.solvers.solver.Solver: __repr__()

5.14 Module fipy.solvers.trilinos.linearLUSolver

Class LinearLUSolver



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

Methods

__init__(*self*, *tolerance*=`1e-10`, *iterations*=`5`, *steps*=`None`, *precon*=`None`)

Create a `Solver` object.

Parameters

tolerance: The required error tolerance.

iterations: The maximum number of iterative steps to perform.

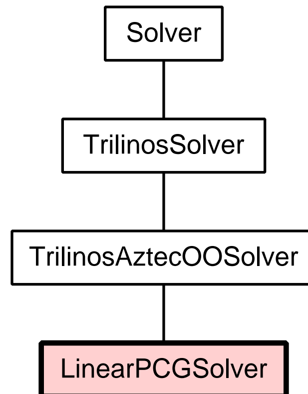
steps: A deprecated name for `iterations`.

Overrides: fipy.solvers.solver.Solver.__init__()

Inherited from fipy.solvers.solver.Solver: __repr__()

5.15 Module fipy.solvers.trilinos.linearPCGSolver

Class LinearPCGSolver



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

Methods

`__init__`(*self*, *tolerance*=`1e-10`, *iterations*=`1000`, *steps*=`None`, *precon*=`MultilevelSGSPreconditioner()`)

Create a `Solver` object.

Parameters

tolerance: The required error tolerance.

iterations: The maximum number of iterative steps to perform.

steps: A deprecated name for `iterations`.

precon: Preconditioner to use.

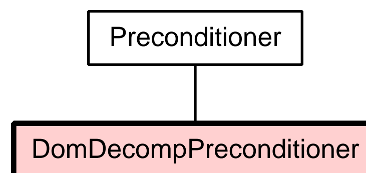
Overrides: fipy.solvers.solver.Solver.__init__()

Inherited from fipy.solvers.solver.Solver: __repr__()

5.16 Package fipy.solvers.trilinos.preconditioners

5.17 Module fipy.solvers.trilinos.preconditioners.domDecompPreconditioner

Class DomDecompPreconditioner



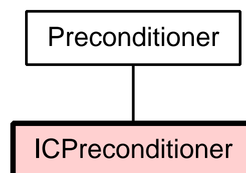
Domain Decomposition preconditioner for Trilinos solvers.

Methods

Inherited from fipy.solvers.trilinos.preconditioners.preconditioner.Preconditioner: __init__()

5.18 Module fipy.solvers.trilinos.preconditioners.icPreconditioner

Class ICPreconditioner



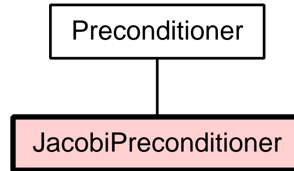
Incomplete Cholesky Preconditioner from IFPACK for Trilinos Solvers.

Methods

Inherited from fipy.solvers.trilinos.preconditioners.preconditioner.Preconditioner: __init__()

5.19 Module fipy.solvers.trilinos.preconditioners.jacobiPreconditioner

Class JacobiPreconditioner



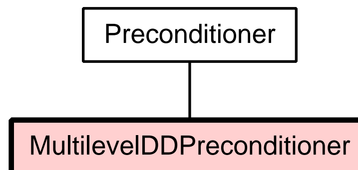
Jacobi Preconditioner for Trilinos solvers.

Methods

Inherited from fipy.solvers.trilinos.preconditioners.preconditioner.Preconditioner: __init__()

5.20 Module fipy.solvers.trilinos.preconditioners.multilevelDDPreconditioner

Class MultilevelDDPreconditioner



Multilevel preconditioner for Trilinos solvers using Aztec precondtioners (DomDecomp, ILU(fill=0)) as smoothers.

Methods

`__init__`(*self*, *levels*=10)

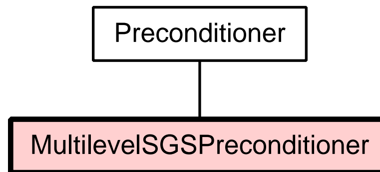
Initialize the multilevel preconditioner

- `levels`: Maximum number of levels

Overrides: fipy.solvers.trilinos.preconditioners.preconditioner.Preconditioner.__init__()

5.21 Module fipy.solvers.trilinos.preconditioners.multilevelSGSPreconditioner

Class MultilevelSGSPreconditioner



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

Methods

`__init__`(*self*, *levels*=10)

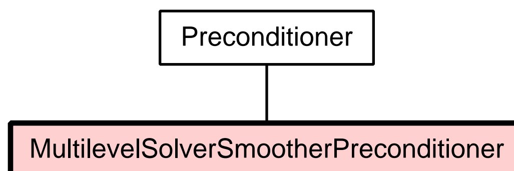
Initialize the multilevel preconditioner

- `levels`: Maximum number of levels

Overrides: fipy.solvers.trilinos.preconditioners.preconditioner.Preconditioner.__init__()

5.22 Module fipy.solvers.trilinos.preconditioners.multilevelSolverSmootherPreconditioner

Class MultilevelSolverSmootherPreconditioner



Multilevel preconditioner for Trilinos solvers using Aztec solvers as smoothers.

Methods

`__init__`(*self*, *levels*=10)

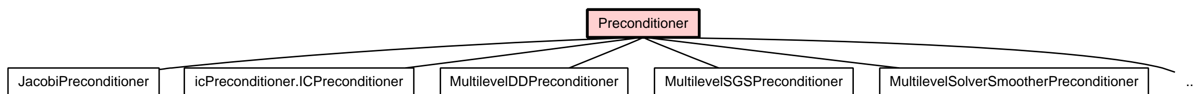
Initialize the multilevel preconditioner

- `levels`: Maximum number of levels

Overrides: fipy.solvers.trilinos.preconditioners.preconditioner.Preconditioner.__init__()

5.23 Module fipy.solvers.trilinos.preconditioners.preconditioner

Class Preconditioner



Known Subclasses: fipy.solvers.trilinos.preconditioners.jacobiPreconditioner.JacobiPreconditioner, fipy.solvers.trilinos.preconditioners.icPreconditioner.ICPreconditioner, fipy.solvers.trilinos.preconditioners.multilevelDDPreconditioner.MultilevelDDPreconditioner, fipy.solvers.trilinos.preconditioners.multilevelSGSPreconditioner.MultilevelSGSPreconditioner, fipy.solvers.trilinos.preconditioners.multilevelSolverSmootherPreconditioner.MultilevelSolverSmootherPreconditioner, fipy.solvers.trilinos.preconditioners.domDecompPreconditioner.DomDecompPreconditioner

The base Preconditioner class.

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

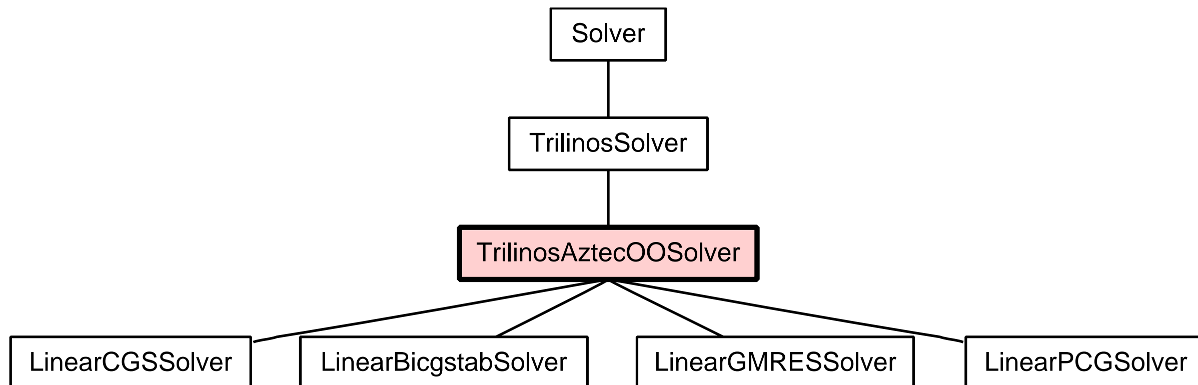
Methods

`__init__`(*self*)

Create a `Preconditioner` object.

5.24 Module fipy.solvers.trilinos.trilinosAztecOOSolver

Class TrilinosAztecOOSolver



Known Subclasses: fipy.solvers.trilinos.linearCGSSolver.LinearCGSSolver, fipy.solvers.trilinos.linearBicgstabSolver.LinearBicgstabSolver, fipy.solvers.trilinos.linearGMRESSolver.LinearGMRESSolver, fipy.solvers.trilinos.linearPCGSolver.LinearPCGSolver

Attention!

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

Methods

`__init__`(*self*, *tolerance*=`1e-10`, *iterations*=`1000`, *steps*=`None`, *precon*=`JacobiPreconditioner()`)

Create a `Solver` object.

Parameters

tolerance: The required error tolerance.

iterations: The maximum number of iterative steps to perform.

steps: A deprecated name for `iterations`.

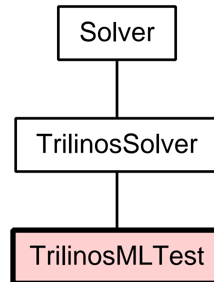
precon: Preconditioner object to use.

Overrides: fipy.solvers.solver.Solver.`__init__`()

Inherited from fipy.solvers.solver.Solver: __repr__()

5.25 Module fipy.solvers.trilinos.trilinosMLTest

Class TrilinosMLTest



This solver class does not actually solve the system, but outputs information about what ML preconditioner settings will work best.

Methods

`__init__`(*self*, *tolerance*=`1e-10`, *iterations*=`5`, *steps*=`None`, *MLOptions*={}, *testUnsupported*=`False`)

For detailed information on the possible parameters for ML, see http://trilinos.sandia.gov/packages/ml/documentation.html

Currently, passing options to Aztec through ML is not supported.

Parameters

tolerance: The required error tolerance.

iterations: The maximum number of iterations to perform per test.

steps: A deprecated name for `iterations`.

MLOptions: Options to pass to ML. A dictionary of {option:value} pairs. This will be passed to ML.SetParameterList.

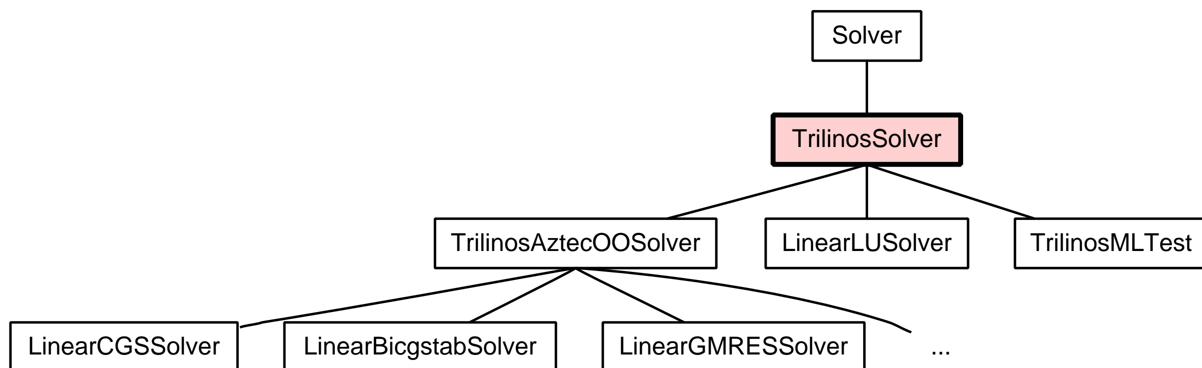
testUnsupported: test smoothers that are not currently implemented in preconditioner objects.

Overrides: fipy.solvers.solver.Solver.__init__()

Inherited from fipy.solvers.solver.Solver: __repr__()

5.26 Module fipy.solvers.trilinos.trilinosSolver

Class TrilinosSolver



Known Subclasses: fipy.solvers.trilinos.trilinosAztecOOSolver.TrilinosAztecOOSolver, fipy.solvers.trilinos.linearLUSolver.LinearLUSolver, fipy.solvers.trilinos.trilinosMLTest.TrilinosMLTest

Attention!

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

Methods

`__init__`(*self*, **args*, ***kwargs*)

Create a `Solver` object.

Parameters

tolerance: The required error tolerance.

iterations: The maximum number of iterative steps to perform.

steps: A deprecated name for `iterations`.

precon: Preconditioner to use. This parameter is only available for Trilinos solvers.

Overrides: fipy.solvers.solver.Solver.__init__() *(inherited documentation)*

Inherited from fipy.solvers.solver.Solver: __repr__()

Package fipy.steppers

6.1 Functions

`residual(`*var*, *matrix*, *RHSvector*`)`

Determines the residual for the current solution matrix and variable.

Parameters

var: The `CellVariable` in question, *prior* to solution.

matrix: The coefficient matrix at this step/sweep

RHSvector: The

Returns:

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

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

`error(`*var*, *matrix*, *RHSvector*, *norm*`)`

Parameters

var: The `CellVariable` in question.

matrix: *(ignored)*

RHSvector: *(ignored)*

norm: A function that will normalize its `array` argument and return a single number

Returns:

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

where $\|\vec{x}\|_?$ is the normalization of $\vec{x}$ provided by ‘norm()‘.

`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 $\vec{x}$.

`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 $\vec{x}$.

`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 $\vec{x}$.

`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

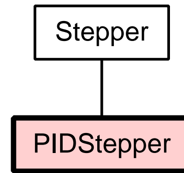
args: The unnamed function argument `list`

kwargs: The named function argument `dict`

Returns: the final residual

6.2 Module fipy.steppers.pidStepper

Class PIDStepper



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},
}
```

Methods

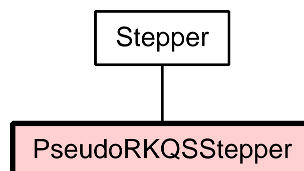
__init__(*self*, *vardata*=(), *proportional*=0.075, *integral*=0.175, *derivative*=0.01)

Overrides: fipy.steppers.stepper.Stepper.__init__()

Inherited from fipy.steppers.stepper.Stepper: failFn(), step(), successFn(), sweepFn()

6.3 Module fipy.steppers.pseudoRKQSStepper

Class PseudoRKQSStepper



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.

Methods

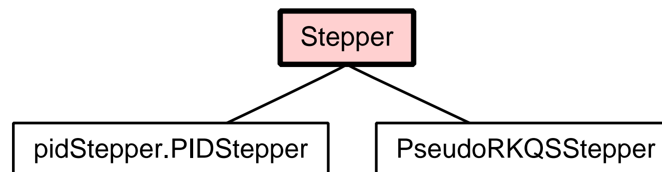
`__init__`(*self*, *vardata*=`()`, *safety*=`0.9`, *pgrow*=`-0.2`, *pshrink*=`-0.25`, *errcon*=`0.000189`)

Overrides: fipy.steppers.stepper.Stepper.__init__()

Inherited from fipy.steppers.stepper.Stepper: failFn(), step(), successFn(), sweepFn()

6.4 Module fipy.steppers.stepper

Class Stepper



Known Subclasses: fipy.steppers.pidStepper.PIDStepper, fipy.steppers.pseudoRKQSStepper.PseudoRKQSStepper

Methods

`__init__`(*self*, *vardata*=`()`)

`sweepFn`(*vardata*, *dt*, **args*, ***kwargs*)

`successFn`(*vardata*, *dt*, *dtPrev*, *elapsed*, **args*, ***kwargs*)

---

failFn(*vardata*, *dt*, **args*, ***kwargs*)

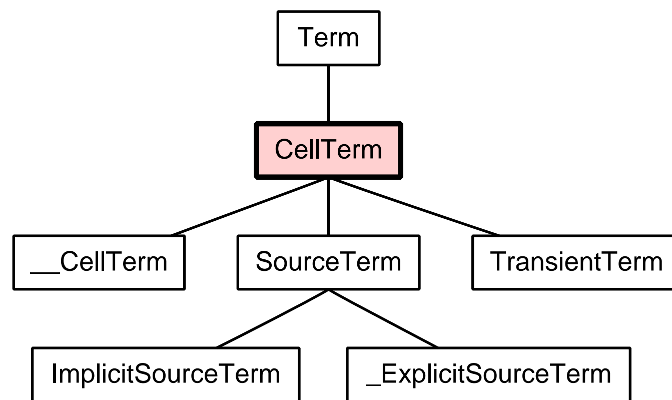
---

step(*self*, *dt*, *dtTry*=None, *dtMin*=None, *dtPrev*=None, *sweepFn*=None, *successFn*=None, *failFn*=None, **args*, ***kwargs*)

Package fipy.terms

7.1 Module fipy.terms.cellTerm

Class CellTerm



Known Subclasses: fipy.terms.cellTerm.__CellTerm, fipy.terms.sourceTerm.SourceTerm, fipy.terms.transientTerm.TransientTerm

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

Methods

`__init__`(*self*, *coeff*=1.0)

Create a `Term`.

Parameters

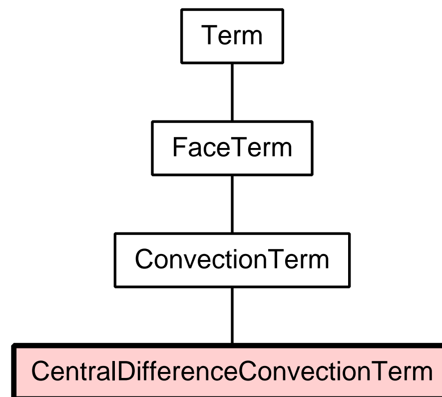
coeff: The coefficient for the term. A `CellVariable` or number. `FaceVariable` objects are also acceptable for diffusion or convection terms.

Overrides: fipy.terms.term.Term.__init__() *(inherited documentation)*

Inherited from fipy.terms.term.Term: __add__(), __eq__(), __neg__(), __pos__(), __radd__(), __repr__(), __rsub__(), __sub__(), cacheMatrix(), cacheRHSvector(), copy(), getMatrix(), getRHSvector(), justResidualVector(), residualVectorAndNorm(), solve(), sweep()

7.2 Module fipy.terms.centralDiffConvectionTerm

Class CentralDifferenceConvectionTerm



The `CentralDifferenceConvectionTerm` 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” in the main FiPy guide[2, § 3.5].

Methods

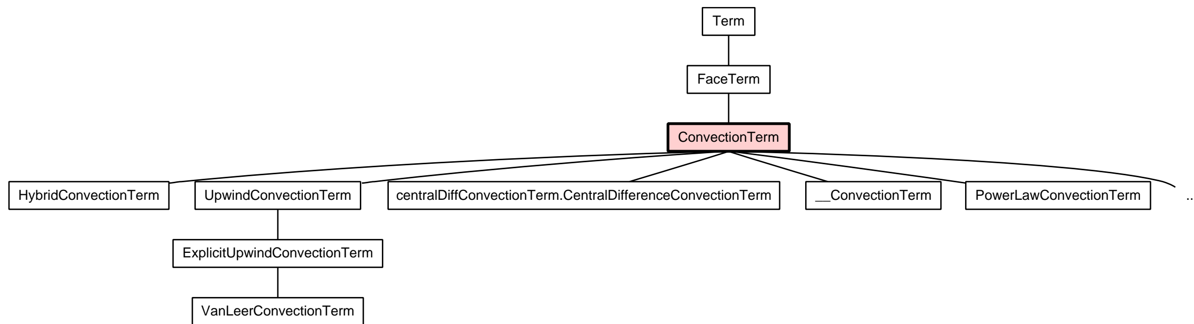
Inherited from fipy.terms.convectionTerm.ConvectionTerm: __add__(), __init__()

Inherited from fipy.terms.term.Term: __eq__(), __neg__(), __pos__(), __radd__(), __repr__(), __rsub__(), __sub__(), cacheMatrix(), cacheRHSvector(), copy(), getMatrix(), getRHSvector(), justResidualVector(), residualVectorAndNorm(), solve(), sweep()

7.3 Module fipy.terms.collectedDiffusionTerm

7.4 Module fipy.terms.convectionTerm

Class ConvectionTerm



Known Subclasses: fipy.terms.hybridConvectionTerm.HybridConvectionTerm, fipy.terms.upwindConvectionTerm.UpwindConvectionTerm, fipy.terms.centralDiffConvectionTerm.CentralDifferenceConvectionTerm, fipy.terms.convectionTerm.__ConvectionTerm, fipy.terms.powerLawConvectionTerm.PowerLawConvectionTerm, fipy.terms.exponentialConvectionTerm.ExponentialConvectionTerm

Attention!

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

Methods

`__init__`(*self*, *coeff*=`1.0`, *diffusionTerm*=`None`)

Create a `ConvectionTerm` object.

```
>>> from fipy.meshes.grid1D import Grid1D
>>> from fipy.variables.cellVariable import CellVariable
>>> from fipy.variables.faceVariable import FaceVariable
>>> m = Grid1D(nx = 2)
>>> cv = CellVariable(mesh = m)
>>> fv = FaceVariable(mesh = m)
>>> vcv = CellVariable(mesh=m, rank=1)
>>> vfv = FaceVariable(mesh=m, rank=1)
>>> __ConvectionTerm(coeff = cv)
Traceback (most recent call last):
 ...
```

```
TypeError: The coefficient must be a vector value.
>>> __ConvectionTerm(coeff = fv)
Traceback (most recent call last):
 ...
TypeError: The coefficient must be a vector value.
>>> __ConvectionTerm(coeff = vcv)
__ConvectionTerm(coeff=_ArithmeticCellToFaceVariable(value=array([[ 0., 0., 0.]]),
mesh=UniformGrid1D(dx=1.0, nx=2)))
>>> __ConvectionTerm(coeff = vfv)
__ConvectionTerm(coeff=FaceVariable(value=array([[ 0., 0., 0.]]),
mesh=UniformGrid1D(dx=1.0, nx=2)))
>>> __ConvectionTerm(coeff = (1,))
__ConvectionTerm(coeff=(1,))
>>> from fipy.terms.explicitUpwindConvectionTerm import ExplicitUpwindConvectionTerm
>>> ExplicitUpwindConvectionTerm(coeff = (0,)).solve(var = cv)
>>> ExplicitUpwindConvectionTerm(coeff = 1).solve(var = cv)
Traceback (most recent call last):
 ...
TypeError: The coefficient must be a vector value.
>>> from fipy.meshes.grid2D import Grid2D
>>> m2 = Grid2D(nx=2, ny=1)
>>> cv2 = CellVariable(mesh=m2)
>>> vcv2 = CellVariable(mesh=m2, rank=1)
>>> vfv2 = FaceVariable(mesh=m2, rank=1)
>>> __ConvectionTerm(coeff=vcv2)
__ConvectionTerm(coeff=_ArithmeticCellToFaceVariable(value=array([[ 0., 0., 0., 0., 0.,
0., 0.],
 [ 0., 0., 0., 0., 0., 0., 0.]]), mesh=UniformGrid2D(dx=1.0, dy=1.0, nx=2, ny=1)))
>>> __ConvectionTerm(coeff=vfv2)
__ConvectionTerm(coeff=FaceVariable(value=array([[ 0., 0., 0., 0., 0., 0., 0.],
 [ 0., 0., 0., 0., 0., 0., 0.]]), mesh=UniformGrid2D(dx=1.0, dy=1.0, nx=2, ny=1)))
>>> ExplicitUpwindConvectionTerm(coeff = ((0,),(0,))).solve(var=cv2)
>>> ExplicitUpwindConvectionTerm(coeff = (0,0)).solve(var=cv2)
```

Parameters

coeff: The Term's coefficient value.

diffusionTerm: ** deprecated **. The Peclet number is calculated automatically.

Overrides: fipy.terms.term.Term.__init__()

__add__(*self*, *other*)

Add a Term to another Term, number or variable.

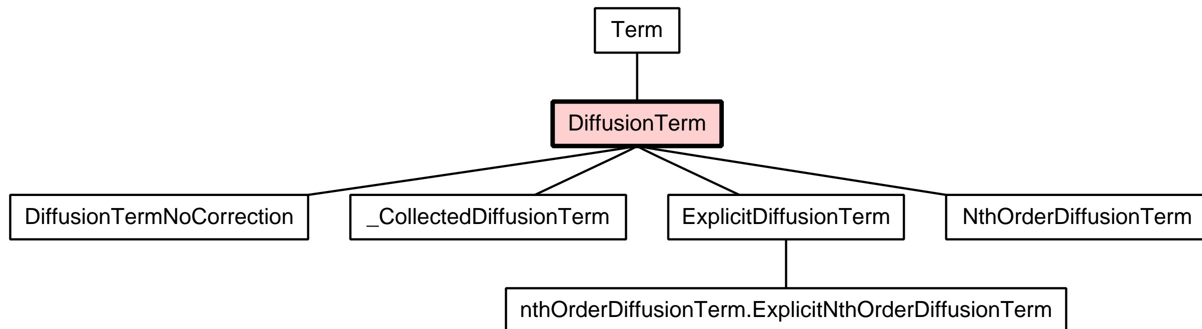
```
>>> __Term(coeff=1.) + 10.
10.0 + __Term(coeff=1.0) == 0
>>> __Term(coeff=1.) + __Term(coeff=2.)
__Term(coeff=3.0)
```

Overrides: fipy.terms.term.Term.__add__() *(inherited documentation)*

Inherited from fipy.terms.term.Term: __eq__(), __neg__(), __pos__(), __radd__(), __repr__(), __rsub__(), __sub__(), cacheMatrix(), cacheRHSvector(), copy(), getMatrix(), getRHSvector(), justResidualVector(), residualVectorAndNorm(), solve(), sweep()

7.5 Module fipy.terms.diffusionTerm

Class DiffusionTerm



Known Subclasses: fipy.terms.diffusionTerm.DiffusionTermNoCorrection, fipy.terms.collectedDiffusionTerm._CollectedDiffusionTerm, fipy.terms.explicitDiffusionTerm.ExplicitDiffusionTerm, fipy.terms.nthOrderDiffusionTerm.NthOrderDiffusionTerm

This term represents a higher order diffusion term. The order of the term is determined by the number of `coeffs`, such that:

```
DiffusionTerm(D1, mesh, bcs)
```

represents a typical 2nd-order diffusion term of the form

$$\nabla \cdot (D_1 \nabla \phi)$$

and:

```
DiffusionTerm((D1,D2), mesh, bcs)
```

represents a 4th-order Cahn-Hilliard term of the form

$$\nabla \cdot \{D_1 \nabla [\nabla \cdot (D_2 \nabla \phi)]\}$$

and so on.

Methods

__add__(*self*, *other*)

Add a `Term` to another `Term`, number or variable.

```
>>> __Term(coeff=1.) + 10.
10.0 + __Term(coeff=1.0) == 0
>>> __Term(coeff=1.) + __Term(coeff=2.)
__Term(coeff=3.0)
```

Overrides: fipy.terms.term.Term.__add__() *(inherited documentation)*

__init__(*self*, *coeff*=(1.0))

Create a `DiffusionTerm`.

Parameters

coeff: `Tuple` or `list` of `FaceVariables` or numbers.

Overrides: fipy.terms.term.Term.__init__()

__neg__(*self*)

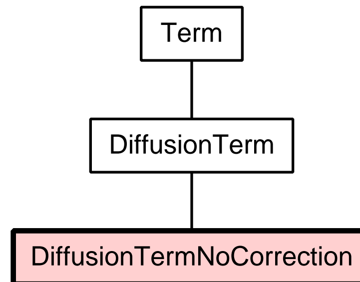
Negate the term.

```
>>> -DiffusionTerm(coeff=[1.])
DiffusionTerm(coeff=[-1.0])

>>> -DiffusionTerm()
DiffusionTerm(coeff=[-1.0])
```

Overrides: fipy.terms.term.Term.__neg__()

Inherited from fipy.terms.term.Term: __eq__(), __pos__(), __radd__(), __repr__(), __rsub__(), __sub__(), cacheMatrix(), cacheRHSvector(), copy(), getMatrix(), getRHSvector(), justResidualVector(), residualVectorAndNorm(), solve(), sweep()

Class DiffusionTermNoCorrection



Methods

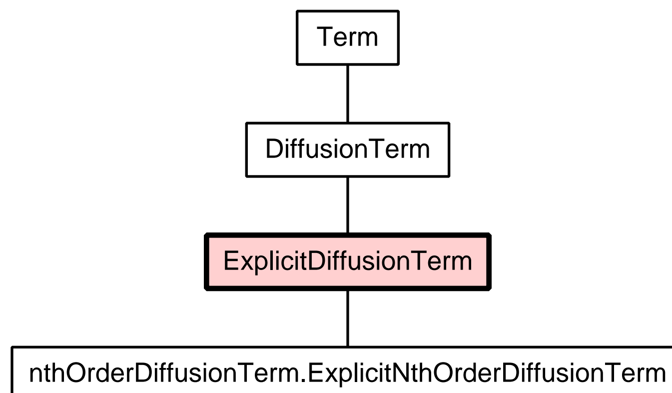
Inherited from fipy.terms.diffusionTerm.DiffusionTerm: __add__(), __init__(), __neg__()

Inherited from fipy.terms.term.Term: __eq__(), __pos__(), __radd__(), __repr__(), __rsub__(), __sub__(), cacheMatrix(), cacheRHSvector(), copy(), getMatrix(), getRHSvector(), justResidualVector(), residualVectorAndNorm(), solve(), sweep()

7.6 Module fipy.terms.equation

7.7 Module fipy.terms.explicitDiffusionTerm

Class ExplicitDiffusionTerm



Known Subclasses: fipy.terms.nthOrderDiffusionTerm.ExplicitNthOrderDiffusionTerm

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.

Methods

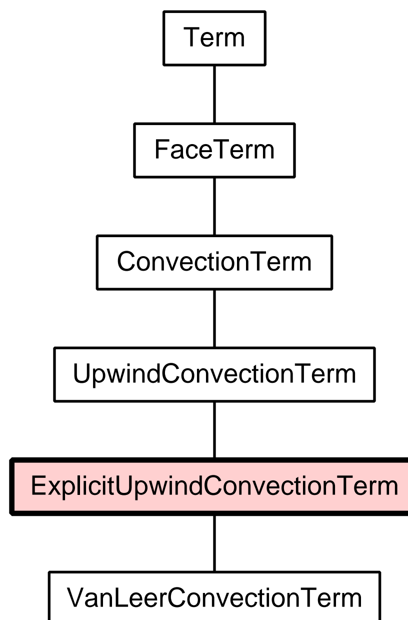
Inherited from fipy.terms.diffusionTerm.DiffusionTerm: __add__(), __init__(), __neg__()

Inherited from fipy.terms.term.Term: __eq__(), __pos__(), __radd__(), __repr__(), __rsub__(), __sub__(), cacheMatrix(), cacheRHSvector(), copy(), getMatrix(), getRHSvector(), justResidualVector(), residualVectorAndNorm(), solve(), sweep()

7.8 Module fipy.terms.explicitSourceTerm

7.9 Module fipy.terms.explicitUpwindConvectionTerm

Class ExplicitUpwindConvectionTerm



Known Subclasses: fipy.terms.vanLeerConvectionTerm.VanLeerConvectionTerm

The discretization for the `ExplicitUpwindConvectionTerm` 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" in the main FiPy guide[2, § 3.5].

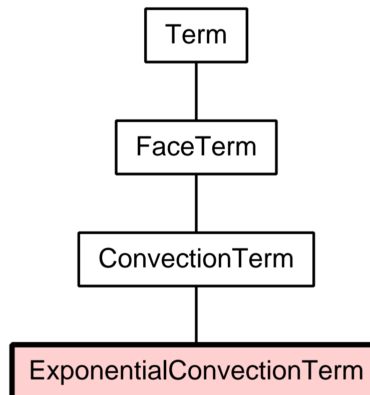
Methods

Inherited from fipy.terms.convectionTerm.ConvectionTerm: __add__(), __init__()

Inherited from fipy.terms.term.Term: __eq__(), __neg__(), __pos__(), __radd__(), __repr__(), __rsub__(), __sub__(), cacheMatrix(), cacheRHSvector(), copy(), getMatrix(), getRHSvector(), justResidualVector(), residualVectorAndNorm(), solve(), sweep()

7.10 Module fipy.terms.exponentialConvectionTerm

Class ExponentialConvectionTerm



The discretization for the `ExponentialConvectionTerm` 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" in the main FiPy guide[2, § 3.5].

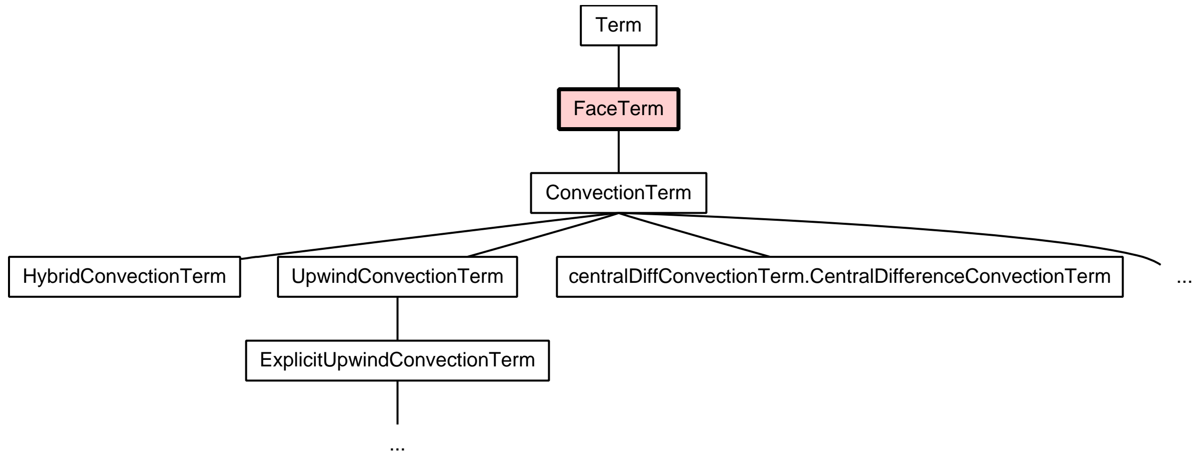
Methods

Inherited from fipy.terms.convectionTerm.ConvectionTerm: __add__(), __init__()

Inherited from fipy.terms.term.Term: __eq__(), __neg__(), __pos__(), __radd__(), __repr__(), __rsub__(), __sub__(), cacheMatrix(), cacheRHSvector(), copy(), getMatrix(), getRHSvector(), justResidualVector(), residualVectorAndNorm(), solve(), sweep()

7.11 Module fipy.terms.faceTerm

Class FaceTerm



Known Subclasses: fipy.terms.convectionTerm.ConvectionTerm

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

Methods

`__init__`(*self*, *coeff*=1.0)

Create a `Term`.

Parameters

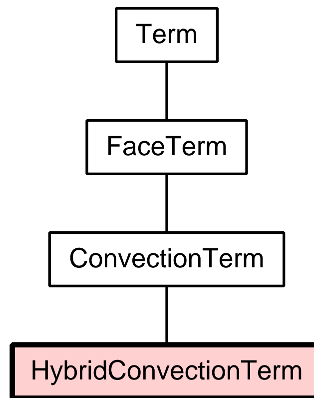
coeff: The coefficient for the term. A `CellVariable` or number. `FaceVariable` objects are also acceptable for diffusion or convection terms.

Overrides: fipy.terms.term.Term.__init__() *(inherited documentation)*

Inherited from fipy.terms.term.Term: __add__(), __eq__(), __neg__(), __pos__(), __radd__(), __repr__(), __rsub__(), __sub__(), cacheMatrix(), cacheRHSvector(), copy(), getMatrix(), getRHSvector(), justResidualVector(), residualVectorAndNorm(), solve(), sweep()

7.12 Module fipy.terms.hybridConvectionTerm

Class HybridConvectionTerm



The discretization for the `HybridConvectionTerm` 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" in the main FiPy guide[2, § 3.5].

Methods

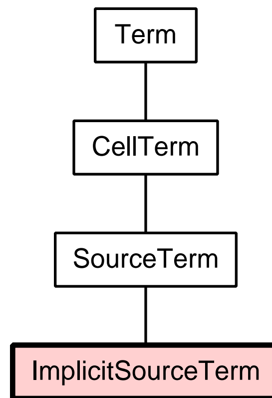
Inherited from fipy.terms.convectionTerm.ConvectionTerm: __add__(), __init__()

Inherited from fipy.terms.term.Term: __eq__(), __neg__(), __pos__(), __radd__(), __repr__(), __rsub__(), __sub__(), cacheMatrix(), cacheRHSvector(), copy(), getMatrix(), getRHSvector(), justResidualVector(), residualVectorAndNorm(), solve(), sweep()

7.13 Module fipy.terms.implicitDiffusionTerm

7.14 Module fipy.terms.implicitSourceTerm

Class ImplicitSourceTerm



The `ImplicitSourceTerm` represents

$$\int_V \phi S \, dV \simeq \phi_P S_P V_P$$

where S is the `coeff` value.

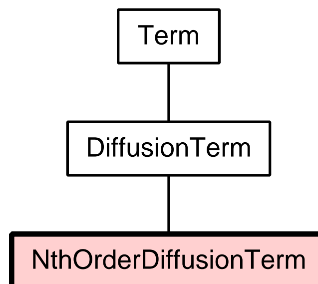
Methods

Inherited from fipy.terms.sourceTerm.SourceTerm: __add__(), __init__()

Inherited from fipy.terms.term.Term: __eq__(), __neg__(), __pos__(), __radd__(), __repr__(), __rsub__(), __sub__(), cacheMatrix(), cacheRHSvector(), copy(), getMatrix(), getRHSvector(), justResidualVector(), residualVectorAndNorm(), solve(), sweep()

7.15 Module fipy.terms.nthOrderDiffusionTerm

Class NthOrderDiffusionTerm



Methods

`__init__`(*self*, *coeff*)

Attention!

This class is deprecated. Use `ImplicitDiffusionTerm` instead.

Parameters

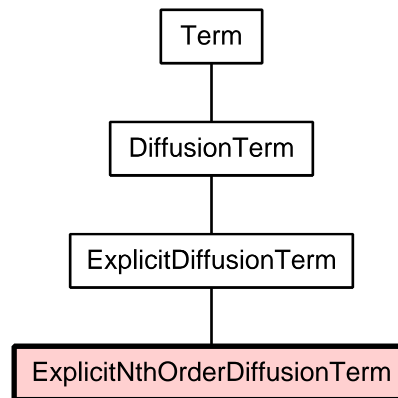
coeff: `Tuple` or `list` of `FaceVariables` or numbers.

Overrides: fipy.terms.term.Term.__init__()

Inherited from fipy.terms.diffusionTerm.DiffusionTerm: __add__(), __neg__()

Inherited from fipy.terms.term.Term: __eq__(), __pos__(), __radd__(), __repr__(), __rsub__(), __sub__(), cacheMatrix(), cacheRHSvector(), copy(), getMatrix(), getRHSvector(), justResidualVector(), residualVectorAndNorm(), solve(), sweep()

Class ExplicitNthOrderDiffusionTerm



Methods

`__init__`(*self*, *coeff*)

Attention!

This class is deprecated. Use `ExplicitDiffusionTerm` instead.

Parameters

coeff: `Tuple` or `list` of `FaceVariables` or numbers.

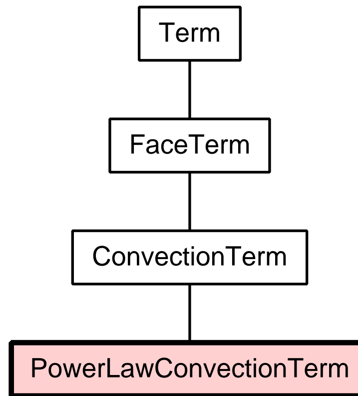
Overrides: fipy.terms.term.Term.__init__()

Inherited from fipy.terms.diffusionTerm.DiffusionTerm: __add__(), __neg__()

Inherited from fipy.terms.term.Term: __eq__(), __pos__(), __radd__(), __repr__(), __rsub__(), __sub__(), cacheMatrix(), cacheRHSvector(), copy(), getMatrix(), getRHSvector(), justResidualVector(), residualVectorAndNorm(), solve(), sweep()

7.16 Module fipy.terms.powerLawConvectionTerm

Class PowerLawConvectionTerm



The discretization for the `PowerLawConvectionTerm` 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" in the main FiPy guide[2, § 3.5].

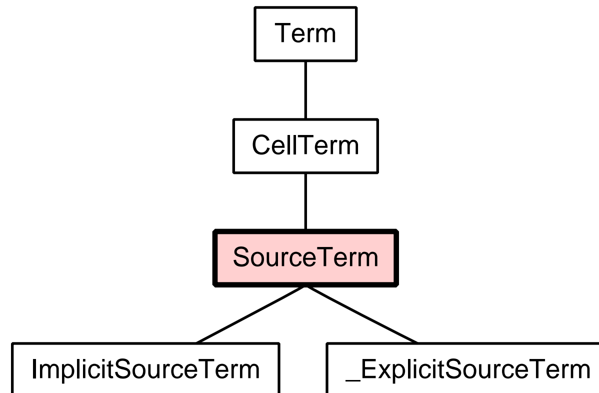
Methods

Inherited from fipy.terms.convectionTerm.ConvectionTerm: __add__(), __init__()

Inherited from fipy.terms.term.Term: __eq__(), __neg__(), __pos__(), __radd__(), __repr__(), __rsub__(), __sub__(), cacheMatrix(), cacheRHSvector(), copy(), getMatrix(), getRHSvector(), justResidualVector(), residualVectorAndNorm(), solve(), sweep()

7.17 Module fipy.terms.sourceTerm

Class SourceTerm



Known Subclasses: fipy.terms.implicitSourceTerm.ImplicitSourceTerm, fipy.terms.explicitSourceTerm._ExplicitSourceTerm

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

Methods

`__init__`(*self*, *coeff*=0.0)

Create a `Term`.

Parameters

coeff: The coefficient for the term. A `CellVariable` or number. `FaceVariable` objects are also acceptable for diffusion or convection terms.

Overrides: fipy.terms.term.Term.`__init__`() *(inherited documentation)*

`__add__`(*self*, *other*)

Add a `Term` to another `Term`, number or variable.

```
>>> __Term(coeff=1.) + 10.
10.0 + __Term(coeff=1.0) == 0
```

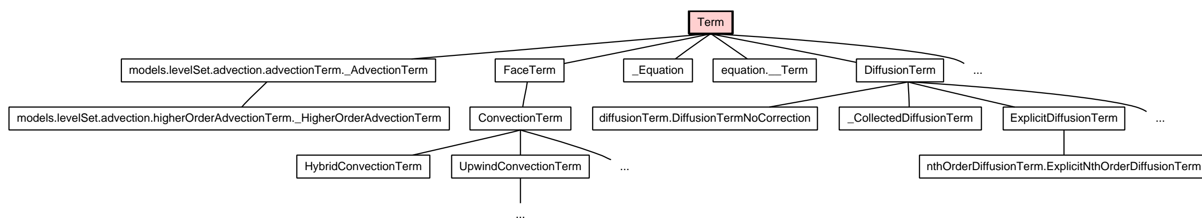
```
>>> __Term(coeff=1.) + __Term(coeff=2.)
__Term(coeff=3.0)
```

Overrides: fipy.terms.term.Term.__add__() *(inherited documentation)*

Inherited from fipy.terms.term.Term: __eq__(), __neg__(), __pos__(), __radd__(), __repr__(), __rsub__(), __sub__(), cacheMatrix(), cacheRHSvector(), copy(), getMatrix(), getRHSvector(), justResidualVector(), residualVectorAndNorm(), solve(), sweep()

7.18 Module fipy.terms.term

Class Term



Known Subclasses: fipy.models.levelSet.advection.advectionTerm._AdvectionTerm, fipy.terms.faceTerm.FaceTerm, fipy.terms.equation._Equation, fipy.terms.equation.__Term, fipy.terms.diffusionTerm.DiffusionTerm, fipy.terms.cellTerm.CellTerm, fipy.terms.term.__Term

Attention!

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

Methods

`__init__`(*self*, *coeff*=1.0)

Create a `Term`.

Parameters

coeff: The coefficient for the term. A `CellVariable` or number. `FaceVariable` objects are also acceptable for diffusion or convection terms.

`copy`(*self*)

`solve`(*self*, *var*, *solver*=`None`, *boundaryConditions*=`()`, *dt*=`1.0`)

Builds and solves the `Term`'s linear system once. This method does not return the residual. It should be used when the residual is not required.

Parameters

var: The variable to be solved for. Provides the initial condition, the old value and holds the solution on completion.

solver: The iterative solver to be used to solve the linear system of equations. Defaults to `LinearPCGSolver` for Pysparse and `LinearLUSolver` for Trilinos.

boundaryConditions: A tuple of boundaryConditions.

dt: The time step size.

`sweep`(*self*, *var*, *solver*=`None`, *boundaryConditions*=`()`, *dt*=`1.0`, *underRelaxation*=`None`, *residualFn*=`None`)

Builds and solves the `Term`'s linear system once. This method also recalculates and returns the residual as well as applying under-relaxation.

Parameters

var: The variable to be solved for. Provides the initial condition, the old value and holds the solution on completion.

solver: The iterative solver to be used to solve the linear system of equations. Defaults to `LinearPCGSolver` for Pysparse and `LinearLUSolver` for Trilinos.

boundaryConditions: A tuple of boundaryConditions.

dt: The time step size.

underRelaxation: Usually a value between `0` and `1` or `None` in the case of no under-relaxation

residualFn: A function that takes var, matrix, and RHSvector arguments, used to customize the residual calculation.

`justResidualVector`(*self*, *var*, *solver*=`None`, *boundaryConditions*=`()`, *dt*=`1.0`, *underRelaxation*=`None`, *residualFn*=`None`)

Builds and the `Term`'s linear system once. This method also recalculates and returns the residual as well as applying under-relaxation.

Parameters

var: The variable to be solved for. Provides the initial condition, the old value and holds the solution on completion.

solver: The iterative solver to be used to solve the linear system of equations. Defaults to `LinearPCGSolver` for Pysparse and `LinearLUSolver` for Trilinos.

boundaryConditions: A tuple of boundaryConditions.

dt: The time step size.

underRelaxation: Usually a value between `0` and `1` or `None` in the case of no under-relaxation

residualFn: A function that takes var, matrix, and RHSvector arguments used to customize the residual calculation.

`residualVectorAndNorm`(*self*, *var*, *solver*=`None`, *boundaryConditions*=`()`, *dt*=`1.0`, *underRelaxation*=`None`, *normFn*=`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.

`cacheMatrix`(*self*)

Informs `solve()` and `sweep()` to cache their matrix so that `getMatrix()` can return the matrix.

`getMatrix`(*self*)

Return the matrix caculated in `solve()` or `sweep()`. The cacheMatrix() method should be called before `solve()` or `sweep()` to cache the matrix.

`cacheRHSvector`(*self*)

Informs `solve()` and `sweep()` to cache their right hand side vector so that `getRHSvector()` can return it.

`getRHSvector`(*self*)

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

`__add__`(*self*, *other*)

Add a `Term` to another `Term`, number or variable.

```
>>> __Term(coeff=1.) + 10.
10.0 + __Term(coeff=1.0) == 0
>>> __Term(coeff=1.) + __Term(coeff=2.)
__Term(coeff=3.0)
```

__radd__(*self*, *other*)

Add a number or variable to a `Term`.

```
>>> 10. + __Term(coeff=1.)
10.0 + __Term(coeff=1.0) == 0
```

__neg__(*self*)

Negate a `Term`.

```
>>> -__Term(coeff=1.)
__Term(coeff=-1.0)
```

__pos__(*self*)

Posate a `Term`.

```
>>> +__Term(coeff=1.)
__Term(coeff=1.0)
```

__sub__(*self*, *other*)

Subtract a `Term` from a `Term`, number or variable.

```
>>> __Term(coeff=1.) - 10.
-10.0 + __Term(coeff=1.0) == 0
>>> __Term(coeff=1.) - __Term(coeff=2.)
__Term(coeff=-1.0)
```

__rsub__(*self*, *other*)

Subtract a `Term`, number or variable from a `Term`.

```
>>> 10. - __Term(coeff=1.)
10.0 + __Term(coeff=-1.0) == 0
```

__eq__(*self*, *other*)

This method allows Terms to be equated in a natural way. Note that the following does not return False.

```
>>> __Term(coeff=1.) == __Term(coeff=2.)
__Term(coeff=-1.0)
```

it is equivalent to,

```
>>> __Term(coeff=1.) - __Term(coeff=2.)
__Term(coeff=-1.0)
```

A Term can also equate with a number.

```
>>> __Term(coeff=1.) == 1.
-1.0 + __Term(coeff=1.0) == 0
```

Likewise for integers.

```
>>> __Term(coeff=1.) == 1
-1 + __Term(coeff=1.0) == 0
```

Equating to zero is allowed, of course

```
>>> __Term(coeff=1.) == 0
__Term(coeff=1.0)
>>> 0 == __Term(coeff=1.)
__Term(coeff=1.0)
```

__repr__(*self*)

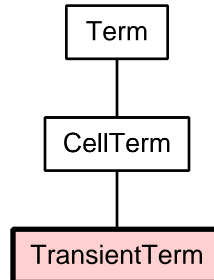
The representation of a Term object is given by,

```
>>> print __Term(123.456)
__Term(coeff=123.456)
```

7.19 Module fipy.terms.test

7.20 Module fipy.terms.transientTerm

Class TransientTerm



The `TransientTerm` represents

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

where ρ is the `coeff` value.

The following test case verifies that variable coefficients and old coefficient values work correctly. We will solve the following equation

$$\frac{\partial \phi^2}{\partial t} = k.$$

The analytic solution is given by

$$\phi = \sqrt{\phi_0^2 + kt},$$

where ϕ_0 is the initial value.

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

```
>>> from fipy.meshes.grid1D import Grid1D
>>> mesh = Grid1D(nx = 1)
>>> from fipy.variables.cellVariable import CellVariable
>>> var = CellVariable(mesh = mesh, value = phi0, hasOld = 1)
>>> from fipy.terms.transientTerm import TransientTerm
>>> from fipy.terms.implicitSourceTerm import ImplicitSourceTerm
```

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

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

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

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

Compare the final result with the analytical solution.

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

Methods

`__add__`(*self*, *other*)

Add a `Term` to another `Term`, number or variable.

```
>>> __Term(coeff=1.) + 10.
10.0 + __Term(coeff=1.0) == 0
>>> __Term(coeff=1.) + __Term(coeff=2.)
__Term(coeff=3.0)
```

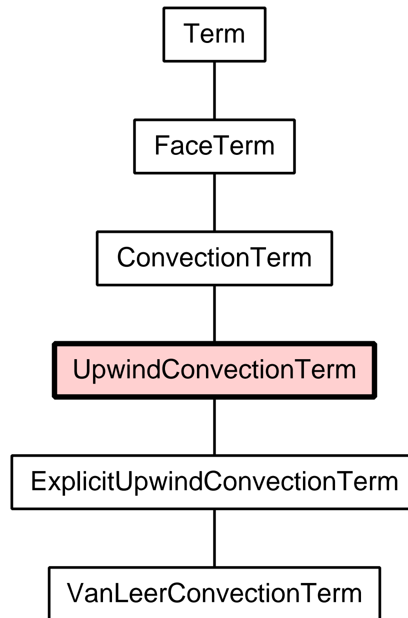
Overrides: fipy.terms.term.Term.__add__() *(inherited documentation)*

Inherited from fipy.terms.cellTerm.CellTerm: __init__()

Inherited from fipy.terms.term.Term: __eq__(), __neg__(), __pos__(), __radd__(), __repr__(), __rsub__(), __sub__(), cacheMatrix(), cacheRHSvector(), copy(), getMatrix(), getRHSvector(), justResidualVector(), residualVectorAndNorm(), solve(), sweep()

7.21 Module fipy.terms.upwindConvectionTerm

Class UpwindConvectionTerm



Known Subclasses: fipy.terms.explicitUpwindConvectionTerm.ExplicitUpwindConvectionTerm

The discretization for the `UpwindConvectionTerm` 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" in the main FiPy guide[2, § 3.5].

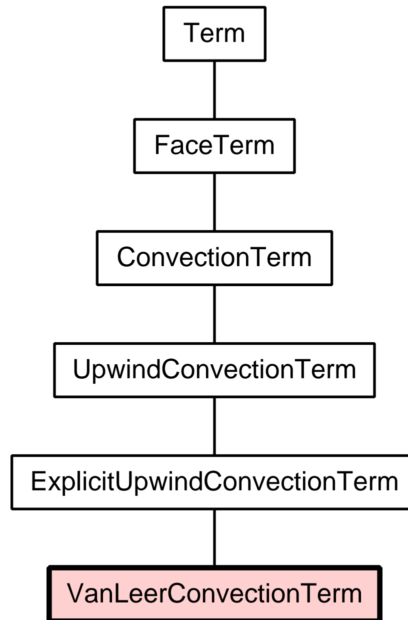
Methods

Inherited from fipy.terms.convectionTerm.ConvectionTerm: __add__(), __init__()

Inherited from fipy.terms.term.Term: __eq__(), __neg__(), __pos__(), __radd__(), __repr__(), __rsub__(), __sub__(), cacheMatrix(), cacheRHSvector(), copy(), getMatrix(), getRHSvector(), justResidualVector(), residualVectorAndNorm(), solve(), sweep()

7.22 Module fipy.terms.vanLeerConvectionTerm

Class VanLeerConvectionTerm



Methods

Inherited from fipy.terms.convectionTerm.ConvectionTerm: __add__(), __init__()

Inherited from fipy.terms.term.Term: __eq__(), __neg__(), __pos__(), __radd__(), __repr__(), __rsub__(), __sub__(), cacheMatrix(), cacheRHSvector(), copy(), getMatrix(), getRHSvector(), justResidualVector(), residualVectorAndNorm(), solve(), sweep()

7.23 Module fipy.test

7.24 Package fipy.tests

unit testing scripts no chapter heading

7.25 Module fipy.tests.doctestPlus

Functions

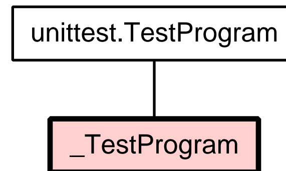
`execButNoTest(name='__main__')`

7.26 Module fipy.tests.lateImportTest

7.27 Module fipy.tests.testBase

7.28 Module fipy.tests.testProgram

Class _TestProgram



Methods

parseArgs(*self*, *argv*)

Overrides: unittest.TestProgram.parseArgs()

Inherited from unittest.TestProgram: __init__(), createTests(), runTests(), usageExit()

Class Variables

Inherited from unittest.TestProgram: USAGE

Package fipy.tools

8.1 Variables

ALLOW_THREADS = 1

BUFSIZE = 10000

CLIP = 0

ERR_CALL = 3

ERR_DEFAULT = 0

ERR_DEFAULT2 = 2084

ERR_IGNORE = 0

ERR_LOG = 5

ERR_PRINT = 4

ERR_RAISE = 2

ERR_WARN = 1

FLOATING_POINT_SUPPORT = 1

FPE_DIVIDEBYZERO = 1

FPE_INVALID = 8

FPE_OVERFLOW = 2

FPE_UNDERFLOW = 4

*False_* = False

Inf = inf

Infinity = inf

MAXDIMS = 32

NAN = nan

NINF = -inf

NZERO = -0.0

NaN = nan

PINF = inf

PZERO = 0.0

RAISE = 2

SHIFT_DIVIDEBYZERO = 0

SHIFT_INVALID = 9

SHIFT_OVERFLOW = 3

SHIFT_UNDERFLOW = 6

ScalarType = (<type 'int'>, <type 'float'>, <type 'complex'>, <type 'l...

*True_* = True

UFUNC_BUFSIZE_DEFAULT = 10000

UFUNC_PYVALS_NAME = 'UFUNC_PYVALS'

WRAP = 1

absolute = <ufunc 'absolute'>

add = <ufunc 'add'>

bitwise_and = <ufunc 'bitwise_and'>

bitwise_not = <ufunc 'invert'>

bitwise_or = <ufunc 'bitwise_or'>

bitwise_xor = <ufunc 'bitwise_xor'>

*c_* = <numpy.lib.index_tricks.CClass object at 0x1827e30>

cast = {<type 'numpy.int64'>: <function <lambda> at 0xed2730>, <...

conj = <ufunc 'conjugate'>

deg2rad = <ufunc 'deg2rad'>

degrees = <ufunc 'degrees'>

divide = <ufunc 'divide'>

e = 2.71828182846

equal = <ufunc 'equal'>

exp2 = <ufunc 'exp2'>

expm1 = <ufunc 'expm1'>

fabs = <ufunc 'fabs'>

floor_divide = <ufunc 'floor_divide'>

fmax = <ufunc 'fmax'>

fmin = <ufunc 'fmin'>

fmod = <ufunc 'fmod'>

frexp = <ufunc 'frexp'>

greater = <ufunc 'greater'>

greater_equal = <ufunc 'greater_equal'>

hypot = <ufunc 'hypot'>

index_exp = <numpy.lib.index_tricks.IndexExpression object at 0x1827eb0>

inf = inf

infty = inf

invert = <ufunc 'invert'>

isfinite = <ufunc 'isfinite'>

isinf = <ufunc 'isinf'>

isnan = <ufunc 'isnan'>

ldexp = <ufunc 'ldexp'>

left_shift = <ufunc 'left_shift'>

less = <ufunc 'less'>

less_equal = <ufunc 'less_equal'>

little_endian = True

log1p = <ufunc 'log1p'>

logaddexp = <ufunc 'logaddexp'>

logaddexp2 = <ufunc 'logaddexp2'>

logical_and = <ufunc 'logical_and'>

logical_not = <ufunc 'logical_not'>

logical_or = <ufunc 'logical_or'>

logical_xor = <ufunc 'logical_xor'>

maximum = <ufunc 'maximum'>

mgrid = <numpy.lib.index_tricks.nd_grid object at 0x181e950>

minimum = <ufunc 'minimum'>

mod = <ufunc 'remainder'>

modf = <ufunc 'modf'>

multiply = <ufunc 'multiply'>

nan = nan

nbytes = {<type 'numpy.int64'>: 8, <type 'numpy.int16'>: 2, <type ...

negative = <ufunc 'negative'>

newaxis = None

not_equal = <ufunc 'not_equal'>

ogrid = <numpy.lib.index_tricks.nd_grid object at 0x181ed30>

ones_like = <ufunc 'ones_like'>

pi = 3.14159265359

power = <ufunc 'power'>

*r_* = <numpy.lib.index_tricks.RClass object at 0x18270d0>

rad2deg = <ufunc 'rad2deg'>

radians = <ufunc 'radians'>

reciprocal = <ufunc 'reciprocal'>

remainder = <ufunc 'remainder'>

right_shift = <ufunc 'right_shift'>

rint = <ufunc 'rint'>

*s_* = <numpy.lib.index_tricks.IndexExpression object at 0x1827ef0>

sctypeDict = {0: <type 'numpy.bool_'>, 1: <type 'numpy.int8'>, 2: <typ...

sctypeNA = {'?': 'Bool', 'B': 'UInt8', 'Bool': <type 'numpy.bool_'>,...

sctypes = {'complex': [<type 'numpy.complex64'>, <type 'numpy.compl...

signbit = <ufunc 'signbit'>

square = <ufunc 'square'>

subtract = <ufunc 'subtract'>

true_divide = <ufunc 'true_divide'>

trunc = <ufunc 'trunc'>

typeDict = {0: <type 'numpy.bool_'>, 1: <type 'numpy.int8'>, 2: <typ...

typeNA = {'?': 'Bool', 'B': 'UInt8', 'Bool': <type 'numpy.bool_'>,...

typecodes = {'All': '?bhilqpBHILQPfdgFDGSUVO', 'AllFloat': 'fdgFDG', ...

8.2 Package fipy.tools.dimensions

8.3 Module fipy.tools.dimensions.DictWithDefault

8.4 Module fipy.tools.dimensions.NumberDict

8.5 Module fipy.tools.dimensions.physicalField

Physical quantities with units.

This module derives from Konrad Hinsen's PhysicalQuantity [3].

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:

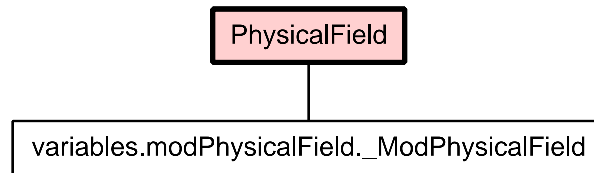
```
1 acres = 4046.8564224 m**2
  1 amu = 1.6605402e-27 kg
  1 Ang = 1e-10 m
  1 atm = 101325.0 kg/s**2/m
    1 b = 1e-28 m
  1 bar = 100000.0 kg/s**2/m
 1 Bohr = 5.29177208115e-11 m
 1 Btui = 1055.05585262 m**2*kg/s**2
    1 c = 299792458.0 m/s
  1 cal = 4.184 m**2*kg/s**2
 1 cali = 4.1868 m**2*kg/s**2
   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
```

Variables

unit = 's'

Class PhysicalField



Known Subclasses: fipy.variables.modPhysicalField._ModPhysicalField

Physical field or quantity with units

Methods

`__init__`(*self*, *value*, *unit*=`None`, *array*=`None`)

Physical Fields can be constructed in one of two ways:

- `PhysicalField(*value*, *unit*)`, where `*value*` is a number of arbitrary type and `*unit*` is a string containing the unit name

```
>>> print PhysicalField(value = 10., unit = 'm')
10.0 m
```

- `PhysicalField(*string*)`, where `*string*` contains both the value and the unit. This form is provided to make interactive use more convenient

```
>>> print PhysicalField(value = "10. m")
10.0 m
```

Dimensionless quantities, with a `unit` of 1, can be specified in several ways

```
>>> print PhysicalField(value = "1")
1.0 1
>>> print PhysicalField(value = 2., unit = " ")
2.0 1
>>> print PhysicalField(value = 2.)
2.0 1
```

Physical arrays are also possible (and are the reason this code was adapted from Konrad Hinsen's original PhysicalQuantity). The `value` can be a Numeric `array`:

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

or a `tuple`:

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

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

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

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

Overrides: object.__init__()

copy(*self*)

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
```

__str__(*self*)

Return human-readable form of a physical quantity

```
>>> print PhysicalField(value = 3., unit = "eV")
3.0 eV
```

Overrides: object.__str__()

__repr__(*self*)

Return representation of a physical quantity suitable for re-use

```
>>> PhysicalField(value = 3., unit = "eV")
PhysicalField(3.0,'eV')
```

Overrides: object.__repr__()

tostring(*self*, *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
```

__add__(*self*, *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
```

__radd__(*self*, *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
```

add(*self*, *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
```

__sub__(*self*, *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
```

subtract(*self*, *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
```

__rsub__(*self*, *other*)

__mul__(*self*, *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
```

`__rmul__`(*self*, *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
```

`multiply`(*self*, *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
```

`__div__`(*self*, *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
```

divide(*self*, *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
```

__rdiv__(*self*, *other*)

__mod__(*self*, *other*)

Return the remainder of dividing two physical quantities. The unit of the result is the unit of the first operand divided by the unit of the second.

```
>>> print PhysicalField(11., 'm') % PhysicalField(2., 's')
1.0 m/s
```

__pow__(*self*, *other*)

Raise a `PhysicalField` to a power. The unit is raised to the same power.

```
>>> print PhysicalField(10., 'm')**2
100.0 m**2
```

__rpow__(*self*, *other*)

__abs__(*self*)

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

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

__pos__(*self*)

__neg__(*self*)

Return the negative of the quantity. The `unit` is unchanged.

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

sign(*self*)

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.]]
```

__nonzero__(*self*)

Test if the quantity is zero.

Should this only pass if the unit offset is zero?

__getitem__(*self*, *index*)

Return the specified element of the array. The unit of the result will be the unit of the array.

```
>>> a = PhysicalField(((3.,4.),(5.,6.)),"m")
>>> print a[1,1]
6.0 m
```

__setitem__(*self*, *index*, *value*)

Assign the specified element of the array, performing appropriate conversions.

```
>>> a = PhysicalField(((3.,4.),(5.,6.)),"m")
>>> a[0,1] = PhysicalField("6 ft")
>>> print a
[[ 3. 1.8288]
 [ 5. 6. ]] m
>>> a[1,0] = PhysicalField("2 min")
Traceback (most recent call last):
 ...
TypeError: Incompatible units
```

itemset(*self*, *value*)

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

```
>>> a = PhysicalField(4.,"m")
>>> a.itemset(PhysicalField("6 ft"))
>>> print a
1.8288 m
>>> a = PhysicalField(((3.,4.),(5.,6.)),"m")
>>> a.itemset(PhysicalField("6 ft"))
Traceback (most recent call last):
 ...
ValueError: can only place a scalar for an array of size 1
>>> a.itemset(PhysicalField("2 min"))
Traceback (most recent call last):
 ...
TypeError: Incompatible units
```

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

Required to prevent numpy not calling the reverse binary operations. Both the following tests are examples ufuncs.

```
>>> print type(numerix.array([1.0, 2.0]) * PhysicalField([1.0, 2.0],
unit="m"))
<class 'fipy.tools.dimensions.physicalField.PhysicalField'>
```

For not very intelligible reasons, the `PhysicalField`

ess gets cast away if there are no units. Probably not harmful, so not worth investigating

```
>>> print type(numerix.array([1.0, 2.0]) * PhysicalField([1.0, 2.0]))
<type 'numpy.ndarray'>
>>> from scipy.special import gamma as Gamma
>>> print type(Gamma(PhysicalField([1.0, 2.0])))
<type 'numpy.ndarray'>
```

`__array__`(*self*, *t*=None)

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

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

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

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

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

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

`__float__`(*self*)

Return a dimensionless PhysicalField quantity as a float.

```
>>> float(PhysicalField("2. m/m"))
2.0
```

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

```
>>> print round(float(PhysicalField("2. deg")), 6)
0.034907
```

If the quantity is not dimensionless, the conversion fails.

```
>>> float(PhysicalField("2. m"))
Traceback (most recent call last):
 ...
TypeError: Not possible to convert a PhysicalField with dimensions to float
```

Just as a Numeric `array` cannot be cast to float, neither can PhysicalField arrays

```
>>> float(PhysicalField(((2.,3.),(4.,5.)),"m/m"))
Traceback (most recent call last):
 ...
TypeError: only length-1 arrays can be converted to Python scalars
```

__gt__(*self*, *other*)

Compare `self` to `other`, returning an array of boolean values corresponding to the test against each element.

```
>>> a = PhysicalField(((3.,4.),(5.,6.)),"m")
>>> print numerix.allclose(a > PhysicalField("13 ft"),
...  [[False, True], [ True, True]])
True
```

Appropriately formatted dimensional quantity strings can also be compared.

```
>>> print numerix.allclose(a > "13 ft",
...  [[False, True], [ True, True]])
True
```

Arrays are compared element to element

```
>>> print numerix.allclose(a > PhysicalField(((3.,13.),(17.,6.)),"ft"),
...  [[ True, True], [False, True]])
True
```

Units must be compatible

```
>>> print a > PhysicalField("1 lb")
Traceback (most recent call last):
 ...
TypeError: Incompatible units
```

And so must array dimensions

```
>>> print a > PhysicalField(((3.,13.,4.),(17.,6.,2.)),"ft")
Traceback (most recent call last):
 ...
ValueError: shape mismatch: objects cannot be broadcast to a single shape
```

`__lt__`(*self*, *other*)

`__le__`(*self*, *other*)

`__eq__`(*self*, *other*)

`__ne__`(*self*, *other*)

`__ge__`(*self*, *other*)

`__len__`(*self*)

`convertToUnit`(*self*, *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
```

inRadians(*self*)

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
```

inDimensionless(*self*)

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
```

inUnitsOf(*self*, **units*)

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

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

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

```
>>> t = PhysicalField(314159., 's')
>>> [str(element) for element in t.inUnitsOf('d','h','min','s')]
['3.0 d', '15.0 h', '15.0 min', '59.0 s']
```

`getsctype`(*self*, *default*=`None`)

Returns the Numpy sctype of the underlying array.

```
>>> PhysicalField(1, 'm').getsctype()
<type 'numpy.int32'>
>>> PhysicalField(1., 'm').getsctype()
<type 'numpy.float64'>
>>> PhysicalField((1,1.), 'm').getsctype()
<type 'numpy.float64'>
```

`getUnit`(*self*)

Return the unit object of `self`.

```
>>> PhysicalField("1 m").getUnit()
<PhysicalUnit m>
```

`setUnit`(*self*, *unit*)

Change the unit object of `self` to `unit`

```
>>> a = PhysicalField(value="1 m")
>>> a.setUnit("m**2/s")
>>> print a
1.0 m**2/s
```

`getNumericValue`(*self*)

Return the `PhysicalField` without units, after conversion to base SI units.

```
>>> print round(PhysicalField("1 inch").getNumericValue(), 6)
0.0254
```

inBaseUnits(*self*)

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

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

inSIUnits(*self*)

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

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

isCompatible(*self*, *unit*)

arccos(*self*)

Return the inverse cosine of the PhysicalField in radians

```
>>> print PhysicalField(0).arccos()
1.57079632679 rad
```

The input PhysicalField must be dimensionless

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

arccosh(*self*)

Return the inverse hyperbolic cosine of the PhysicalField

```
>>> print PhysicalField(2).arccosh()
1.31695789692
```

The input `PhysicalField` must be dimensionless

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

`arcsin(`*self*`)`

Return the inverse sine of the `PhysicalField` in radians

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

The input `PhysicalField` must be dimensionless

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

`sqrt(`*self*`)`

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
```

`sin(`*self*`)`

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(*self*)

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
```

cos(*self*)

Return the cosine of the `PhysicalField`

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

The units of the `PhysicalField` must be an angle

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

cosh(*self*)

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
```

`tan`(*self*)

Return the tangent of the `PhysicalField`

```
>>> round(PhysicalField(numerix.pi/4,"rad").tan(), 6)
1.0
>>> round(PhysicalField(45,"deg").tan(), 6)
1.0
```

The units of the `PhysicalField` must be an angle

```
>>> PhysicalField(45.,"m").tan()
Traceback (most recent call last):
 ...
TypeError: Incompatible units
```

`tanh`(*self*)

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
```

`arctan2`(*self*, *other*)

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

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

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

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

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

arctan(*self*)

Return the arctangent of the `PhysicalField` in radians

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

The input `PhysicalField` must be dimensionless

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

arctanh(*self*)

Return the inverse hyperbolic tangent of the `PhysicalField`

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

The input `PhysicalField` must be dimensionless

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

log(*self*)

Return the natural logarithm of the `PhysicalField`

```
>>> print round(PhysicalField(10).log(), 6)
2.302585
```

The input `PhysicalField` must be dimensionless

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

`log10`(*self*)

Return the base-10 logarithm of the `PhysicalField`

```
>>> print round(PhysicalField(10.).log10(), 6)
1.0
```

The input `PhysicalField` must be dimensionless

```
>>> print round(PhysicalField("1. m").log10(), 6)
Traceback (most recent call last):
 ...
TypeError: Incompatible units
```

`floor`(*self*)

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

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

`ceil`(*self*)

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

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

`conjugate`(*self*)

Return the complex conjugate of the `PhysicalField`.

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

`dot`(*self*, *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
```

take(*self*, *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
```

put(*self*, *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
```

getShape(*self*)

reshape(*self*, *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
```

sum(*self*, *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
```

allclose(*self*, *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(*self*, *other*)

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

Properties

shape

Tuple of array dimensions.

Get: fipy.tools.dimensions.physicalField.PhysicalField.getShape()

Class Variables

__array_priority__ = `100.0`

Class PhysicalUnit

A `PhysicalUnit` represents the units of a `PhysicalField`.

Methods

`__init__`(*self*, *names*, *factor*, *powers*, *offset*=0)

This class is not generally not instantiated by users of this module, but rather it is created in the process of constructing a `PhysicalField`.

Parameters

names: 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.

`__repr__`(*self*)

Return representation of a physical unit

```
>>> PhysicalUnit('m', 1., [1,0,0,0,0,0,0,0,0])
<PhysicalUnit m>
```

__str__(*self*)

Return representation of a physical unit

```
>>> PhysicalUnit('m', 1., [1,0,0,0,0,0,0,0,0])
<PhysicalUnit m>
```

__cmp__(*self*, *other*)

Determine if units are identical

```
>>> a = PhysicalField("1. m")
>>> b = PhysicalField("3. ft")
>>> a.getUnit() == b.getUnit()
0
>>> a.getUnit() == b.inBaseUnits().getUnit()
1
```

Units can only be compared with other units

```
>>> a.getUnit() == 3
Traceback (most recent call last):
 ...
TypeError: PhysicalUnits can only be compared with other PhysicalUnits
```

__mul__(*self*, *other*)

Multiply units together

```
>>> a = PhysicalField("1. m")
>>> b = PhysicalField("3. ft")
>>> a.getUnit() * b.getUnit()
<PhysicalUnit ft*m>
>>> a.getUnit() * b.inBaseUnits().getUnit()
<PhysicalUnit m**2>
>>> c = PhysicalField("1. s")
>>> d = PhysicalField("3. Hz")
>>> c.getUnit() * d.getUnit()
<PhysicalUnit Hz*s>
>>> c.getUnit() * d.inBaseUnits().getUnit()
<PhysicalUnit 1>
```

or multiply units by numbers

```
>>> a.getUnit() * 3.
<PhysicalUnit m*3.0>
```

Units must have zero offset to be multiplied

```
>>> e = PhysicalField("1. kB")
>>> f = PhysicalField("25. degC")
>>> e.getUnit() * f.getUnit()
Traceback (most recent call last):
 ...
TypeError: cannot multiply units with non-zero offset
>>> e.getUnit() * f.inBaseUnits().getUnit()
<PhysicalUnit kB*K>
```

__rmul__(*self*, *other*)

Multiply units together

```
>>> a = PhysicalField("1. m")
>>> b = PhysicalField("3. ft")
>>> a.getUnit() * b.getUnit()
<PhysicalUnit ft*m>
>>> a.getUnit() * b.inBaseUnits().getUnit()
<PhysicalUnit m**2>
>>> c = PhysicalField("1. s")
>>> d = PhysicalField("3. Hz")
>>> c.getUnit() * d.getUnit()
<PhysicalUnit Hz*s>
>>> c.getUnit() * d.inBaseUnits().getUnit()
<PhysicalUnit 1>
```

or multiply units by numbers

```
>>> a.getUnit() * 3.
<PhysicalUnit m*3.0>
```

Units must have zero offset to be multiplied

```
>>> e = PhysicalField("1. kB")
>>> f = PhysicalField("25. degC")
>>> e.getUnit() * f.getUnit()
Traceback (most recent call last):
 ...
TypeError: cannot multiply units with non-zero offset
>>> e.getUnit() * f.inBaseUnits().getUnit()
<PhysicalUnit kB*K>
```

__div__(*self*, *other*)

Divide one unit by another

```
>>> a = PhysicalField("1. m")
>>> b = PhysicalField("3. ft")
>>> a.getUnit() / b.getUnit()
<PhysicalUnit m/ft>
>>> a.getUnit() / b.inBaseUnits().getUnit()
<PhysicalUnit 1>
>>> c = PhysicalField("1. s")
>>> d = PhysicalField("3. Hz")
>>> c.getUnit() / d.getUnit()
<PhysicalUnit s/Hz>
>>> c.getUnit() / d.inBaseUnits().getUnit()
<PhysicalUnit s**2/1>
```

or divide units by numbers

```
>>> a.getUnit() / 3.
<PhysicalUnit m/3.0>
```

Units must have zero offset to be divided

```
>>> e = PhysicalField("1. J")
>>> f = PhysicalField("25. degC")
>>> e.getUnit() / f.getUnit()
Traceback (most recent call last):
 ...
TypeError: cannot divide units with non-zero offset
>>> e.getUnit() / f.inBaseUnits().getUnit()
<PhysicalUnit J/K>
```

`__rdiv__`(*self*, *other*)

Divide something by a unit

```
>>> a = PhysicalField("1. m")
>>> 3. / a.getUnit()
<PhysicalUnit 3.0/m>
```

Units must have zero offset to be divided

```
>>> b = PhysicalField("25. degC")
>>> 3. / b.getUnit()
Traceback (most recent call last):
 ...
TypeError: cannot divide units with non-zero offset
>>> 3. / b.inBaseUnits().getUnit()
<PhysicalUnit 3.0/K>
```

`__pow__`(*self*, *other*)

Raise a unit to an integer power

```
>>> a = PhysicalField("1. m")
>>> a.getUnit()**2
<PhysicalUnit m**2>
>>> a.getUnit()**-2
<PhysicalUnit 1/m**2>
```

Non-integer powers are not supported

```
>>> a.getUnit()**0.5
Traceback (most recent call last):
 ...
TypeError: Illegal exponent
```

Units must have zero offset to be exponentiated

```
>>> b = PhysicalField("25. degC")
>>> b.getUnit()**2
Traceback (most recent call last):
 ...
TypeError: cannot exponentiate units with non-zero offset
>>> b.inBaseUnits().getUnit()**2
<PhysicalUnit K**2>
```

`conversionFactorTo`(*self*, *other*)

Return the multiplication factor between two physical units

```
>>> a = PhysicalField("1. mm")
>>> b = PhysicalField("1. inch")
>>> print round(b.getUnit().conversionFactorTo(a.getUnit()), 6)
25.4
```

Units must have the same fundamental SI units

```
>>> c = PhysicalField("1. K")
>>> c.getUnit().conversionFactorTo(a.getUnit())
Traceback (most recent call last):
 ...
TypeError: Incompatible units
```

If units have different offsets, they must have the same factor

```
>>> d = PhysicalField("1. degC")
>>> c.getUnit().conversionFactorTo(d.getUnit())
1.0
>>> e = PhysicalField("1. degF")
```

```
>>> c.getUnit().conversionFactorTo(e.getUnit())
Traceback (most recent call last):
 ...
TypeError: Unit conversion (K to degF) cannot be expressed as a simple multiplicative
factor
```

conversionTupleTo(*self*, *other*)

Return a `tuple` of the multiplication factor and offset between two physical units

```
>>> a = PhysicalField("1. K").getUnit()
>>> b = PhysicalField("1. degF").getUnit()
>>> [str(round(element,6)) for element in b.conversionTupleTo(a)]
['0.555556', '459.67']
```

isCompatible(*self*, *other*)

Returns a list of which fundamental SI units are compatible between `self` and `other`

```
>>> a = PhysicalField("1. mm")
>>> b = PhysicalField("1. inch")
>>> print numerix.allclose(a.getUnit().isCompatible(b.getUnit()),
...  [True, True, True, True, True, True, True, True, True])
True
>>> c = PhysicalField("1. K")
>>> print numerix.allclose(a.getUnit().isCompatible(c.getUnit()),
...  [False, True, True, True, False, True, True, True, True])
True
```

isDimensionless(*self*)

Returns `True` if the unit is dimensionless

```
>>> PhysicalField("1. m/m").getUnit().isDimensionless()
1
>>> PhysicalField("1. inch").getUnit().isDimensionless()
0
```

isAngle(*self*)

Returns `True` if the unit is an angle

```
>>> PhysicalField("1. deg").getUnit().isAngle()
1
>>> PhysicalField("1. rad").getUnit().isAngle()
1
>>> PhysicalField("1. inch").getUnit().isAngle()
0
```

`isInverseAngle`(*self*)

Returns `True` if the 1 divided by the unit is an angle

```
>>> PhysicalField("1. deg**-1").getUnit().isInverseAngle()
1
>>> PhysicalField("1. 1/rad").getUnit().isInverseAngle()
1
>>> PhysicalField("1. inch").getUnit().isInverseAngle()
0
```

`isDimensionlessOrAngle`(*self*)

Returns `True` if the unit is dimensionless or an angle

```
>>> PhysicalField("1. m/m").getUnit().isDimensionlessOrAngle()
1
>>> PhysicalField("1. deg").getUnit().isDimensionlessOrAngle()
1
>>> PhysicalField("1. rad").getUnit().isDimensionlessOrAngle()
1
>>> PhysicalField("1. inch").getUnit().isDimensionlessOrAngle()
0
```

`setName`(*self*, *name*)

Set the name of the unit to `name`

```
>>> a = PhysicalField("1. m/s").getUnit()
>>> a
<PhysicalUnit m/s>
>>> a.setName('meterpersecond')
>>> a
<PhysicalUnit meterpersecond>
```

`name`(*self*)

Return the name of the unit

```
>>> PhysicalField("1. m").getUnit().name()
'm'
>>> (PhysicalField("1. m") / PhysicalField("1. s")
...  / PhysicalField("1. s")).getUnit().name()
'm/s**2'
```

8.6 Module fipy.tools.dump

Functions

write(*data*, *filename*=None, *extension*='')

Pickle an object and write it to a file. Wrapper for `cPickle.dump()`.

Test to check pickling and unpickling.

```
>>> from fipy.meshes.grid1D import Grid1D
>>> f, tempfile = write(Grid1D(nx = 2))
>>> mesh = read(tempfile, f)
>>> print mesh.getNumberOfCells()
2
```

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.

read(*filename*, *fileobject*=None)

Read a pickled object from a file. Returns the unpickled object. Wrapper for `cPickle.load()`.

Parameters

filename: The name of the file to unpickle the object from.

fileobject: Used to remove temporary files

8.7 Module fipy.tools.inline

Variables

doInline = `False`

inlineFrameComment = `False`

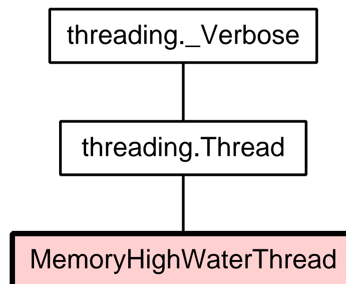
8.8 Module fipy.tools.memoryLeak

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.

8.9 Module fipy.tools.memoryLogger

Class MemoryHighWaterThread



Methods

`__init__(self, pid, sampleTime=1)`

Overrides: threading._Verbose.__init__()

`run(self)`

Overrides: threading.Thread.run()

`stop(self)`

Inherited from threading.Thread: __repr__(), getName(), isAlive(), isDaemon(), join(), setDaemon(), setName(), start()

Class MemoryLogger

Methods

`__init__`(*self*, *sampleTime*=1)

`__del__`(*self*)

`start`(*self*)

`stop`(*self*)

8.10 Module fipy.tools.memoryUsage

This python script is ripped from http://aspn.activestate.com/ASPN/Cookbook/Python/Recipe/286222/index_txt

8.11 Module fipy.tools.numerix

The functions provided in ths module replace the `Numeric` module. The functions work with `Variables`, arrays or numbers. For example, create a `Variable`.

```
>>> from fipy.variables.variable import Variable
>>> var = Variable(value=0)
```

Take the tangent of such a variable. The returned value is itself a `Variable`.

```
>>> v = tan(var)
>>> v
numerix.tan(Variable(value=array(0)))
>>> print float(v)
0.0
```

Take the tangent of a int.

```
>>> tan(0)
0.0
```

Take the tangent of an array.

```
>>> print tan(array((0,0,0)))
[ 0.  0.  0.]
```

Eventually, this module will be the only place in the code where `Numeric` (or `numarray` (or `scipy_core`)) is explicitly imported.

Version: 1.3.0.dev6304

Functions

`zeros(`*a*, *t*=`'l')`

`ones(`*a*, *t*=`'l')`

`getUnit(`*arr*`)`

---

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]
```

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).

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"))
()
```

`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.

`sum`(*arr*, *axis*=0)

The sum of all the elements of `arr` along the specified axis.

`isFloat`(*arr*)

`isInt`(*arr*)

`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
```

`arccos`(*arr*)

Inverse cosine of x, $\cos^{-1} x$

```
>>> print tostring(arccos(0.0), precision=3)
1.571

>>> isnan(arccos(2.0))
True
```

```
>>> print tostring(arccos(array((0,0.5,1.0))), precision=3)
[ 1.571 1.047 0. ]
>>> from fipy.variables.variable import Variable
>>> arccos(Variable(value=(0,0.5,1.0)))
numerix.arccos(Variable(value=array([ 0. , 0.5, 1. ])))
```

Attention!

the next should really return radians, but doesn't

```
>>> print tostring(arccos(Variable(value=(0,0.5,1.0))), precision=3)
[ 1.571 1.047 0. ]
```

arccosh(*arr*)

Inverse hyperbolic cosine of x, $\cosh^{-1} x$

```
>>> print arccosh(1.0)
0.0

>>> isnan(arccosh(0.0))
True

>>> print tostring(arccosh(array((1,2,3))), precision=3)
[ 0. 1.317 1.763]
>>> from fipy.variables.variable import Variable
>>> arccosh(Variable(value=(1,2,3)))
numerix.arccosh(Variable(value=array([1, 2, 3])))
>>> print tostring(arccosh(Variable(value=(1,2,3))), precision=3)
[ 0. 1.317 1.763]
```

arcsin(*arr*)

Inverse sine of x, $\sin^{-1} x$

```
>>> print tostring(arcsin(1.0), precision=3)
1.571

>>> isnan(arcsin(2.0))
True

>>> print tostring(arcsin(array((0,0.5,1.0))), precision=3)
[ 0. 0.524 1.571]
>>> from fipy.variables.variable import Variable
>>> arcsin(Variable(value=(0,0.5,1.0)))
numerix.arcsin(Variable(value=array([ 0. , 0.5, 1. ])))
```

Attention!

the next should really return radians, but doesn't

```
>>> print tostring(arcsin(Variable(value=(0,0.5,1.0))), precision=3)
[ 0. 0.524 1.571]
```

arcsinh(*arr*)

Inverse hyperbolic sine of x, $\sinh^{-1} x$

```
>>> print tostring(arcsinh(1.0), precision=3)
0.881
>>> print tostring(arcsinh(array((1,2,3))), precision=3)
[ 0.881 1.444 1.818]
>>> from fipy.variables.variable import Variable
>>> arcsinh(Variable(value=(1,2,3)))
numerix.arcsinh(Variable(value=array([1, 2, 3])))
>>> print tostring(arcsinh(Variable(value=(1,2,3))), precision=3)
[ 0.881 1.444 1.818]
```

arctan(*arr*)

Inverse tangent of x, $\tan^{-1} x$

```
>>> print tostring(arctan(1.0), precision=3)
0.785
>>> print tostring(arctan(array((0,0.5,1.0))), precision=3)
[ 0. 0.464 0.785]
>>> from fipy.variables.variable import Variable
>>> arctan(Variable(value=(0,0.5,1.0)))
numerix.arctan(Variable(value=array([ 0. , 0.5, 1. ])))
```

Attention!

the next should really return radians, but doesn't

```
>>> print tostring(arctan(Variable(value=(0,0.5,1.0))), precision=3)
[ 0. 0.464 0.785]
```

arctan2(*arr*, *other*)

Inverse tangent of a ratio x/y, $\tan^{-1} \frac{x}{y}$

```
>>> print tostring(arctan2(3.0, 3.0), precision=3)
0.785
>>> print tostring(arctan2(array((0, 1, 2)), 2), precision=3)
[ 0. 0.464 0.785]
>>> from fipy.variables.variable import Variable
>>> arctan2(Variable(value=(0, 1, 2)), 2)
(numerix.arctan2(Variable(value=array([0, 1, 2])), 2))
```

Attention!

the next should really return radians, but doesn't

```
>>> print tostring(arctan2(Variable(value=(0, 1, 2)), 2), precision=3)
[ 0. 0.464 0.785]
```

arctanh(*arr*)

Inverse hyperbolic tangent of x, $\tanh^{-1} x$

```
>>> print tostring(arctanh(0.5), precision=3)
0.549
>>> print tostring(arctanh(array((0,0.25,0.5))), precision=3)
[ 0. 0.255 0.549]
>>> from fipy.variables.variable import Variable
>>> arctanh(Variable(value=(0,0.25,0.5)))
numerix.arctanh(Variable(value=array([ 0. , 0.25, 0.5 ])))
>>> print tostring(arctanh(Variable(value=(0,0.25,0.5))), precision=3)
[ 0. 0.255 0.549]
```

cos(*arr*)

Cosine of x, $\cos x$

```
>>> print allclose(cos(2*pi/6), 0.5)
True
>>> print tostring(cos(array((0,2*pi/6,pi/2))), precision=3, suppress_small=1)
[ 1. 0.5 0. ]
>>> from fipy.variables.variable import Variable
>>> cos(Variable(value=(0,2*pi/6,pi/2), unit="rad"))
numerix.cos(Variable(value=PhysicalField(array([ 0. , 1.04719755, 1.57079633]),'rad')))
>>> print tostring(cos(Variable(value=(0,2*pi/6,pi/2), unit="rad")),
suppress_small=1)
[ 1. 0.5 0. ]
```

cosh(*arr*)

Hyperbolic cosine of x, $\cosh x$

```
>>> print cosh(0)
1.0
>>> print tostring(cosh(array((0,1,2))), precision=3)
[ 1. 1.543 3.762]
>>> from fipy.variables.variable import Variable
>>> cosh(Variable(value=(0,1,2)))
numerix.cosh(Variable(value=array([0, 1, 2])))
>>> print tostring(cosh(Variable(value=(0,1,2))), precision=3)
[ 1. 1.543 3.762]
```

tan(*arr*)

Tangent of x, $\tan x$

```
>>> print tostring(tan(pi/3), precision=3)
1.732
>>> print tostring(tan(array((0,pi/3,2*pi/3))), precision=3)
[ 0. 1.732 -1.732]
>>> from fipy.variables.variable import Variable
>>> tan(Variable(value=(0,pi/3,2*pi/3), unit="rad"))
numerix.tan(Variable(value=PhysicalField(array([ 0. , 1.04719755, 2.0943951 ]),'rad')))
>>> print tostring(tan(Variable(value=(0,pi/3,2*pi/3), unit="rad")), precision=3)
[ 0. 1.732 -1.732]
```

tanh(*arr*)

Hyperbolic tangent of x, $\tanh x$

```
>>> print tostring(tanh(1), precision=3)
0.762
>>> print tostring(tanh(array((0,1,2))), precision=3)
[ 0. 0.762 0.964]
>>> from fipy.variables.variable import Variable
>>> tanh(Variable(value=(0,1,2)))
numerix.tanh(Variable(value=array([0, 1, 2])))
>>> print tostring(tanh(Variable(value=(0,1,2))), precision=3)
[ 0. 0.762 0.964]
```

log10(*arr*)

Base-10 logarithm of x, $\log_{10} x$

```
>>> print log10(10)
1.0
>>> print log10(array((0.1,1,10)))
[-1. 0. 1.]
>>> from fipy.variables.variable import Variable
>>> log10(Variable(value=(0.1,1,10)))
numerix.log10(Variable(value=array([ 0.1, 1. , 10. ])))
>>> print log10(Variable(value=(0.1,1,10)))
[-1. 0. 1.]
```

sin(*arr*)

Sine of x, $\sin x$

```
>>> print sin(pi/6)
0.5
>>> print sin(array((0,pi/6,pi/2)))
[ 0. 0.5 1. ]
>>> from fipy.variables.variable import Variable
>>> sin(Variable(value=(0,pi/6,pi/2), unit="rad"))
numerix.sin(Variable(value=PhysicalField(array([ 0. , 0.52359878, 1.57079633]),'rad')))
>>> print sin(Variable(value=(0,pi/6,pi/2), unit="rad"))
[ 0. 0.5 1. ]
```

sinh(*arr*)

Hyperbolic sine of x, $\sinh x$

```
>>> print sinh(0)
0.0
>>> print tostring(sinh(array((0,1,2))), precision=3)
[ 0. 1.175 3.627]
>>> from fipy.variables.variable import Variable
>>> sinh(Variable(value=(0,1,2)))
numerix.sinh(Variable(value=array([0, 1, 2])))
>>> print tostring(sinh(Variable(value=(0,1,2))), precision=3)
[ 0. 1.175 3.627]
```

sqrt(*arr*)

Square root of x, $\sqrt{x}$

```
>>> print tostring(sqrt(2), precision=3)
1.414
```

```
>>> print tostring(sqrt(array((1,2,3))), precision=3)
[ 1. 1.414 1.732]
>>> from fipy.variables.variable import Variable
>>> sqrt(Variable(value=(1, 2, 3), unit="m**2"))
numerix.sqrt(Variable(value=PhysicalField(array([1, 2, 3]),'m**2')))
>>> print tostring(sqrt(Variable(value=(1, 2, 3), unit="m**2")), precision=3)
[ 1. 1.414 1.732] m
```

`floor`(*arr*)

The largest integer $\leq x$, $\lfloor x \rfloor$

```
>>> print floor(2.3)
2.0
>>> print floor(array((-1.5,2,2.5)))
[-2. 2. 2.]
>>> from fipy.variables.variable import Variable
>>> floor(Variable(value=(-1.5,2,2.5), unit="m**2"))
numerix.floor(Variable(value=PhysicalField(array([-1.5, 2. , 2.5]),'m**2')))
>>> print floor(Variable(value=(-1.5,2,2.5), unit="m**2"))
[-2. 2. 2.] m**2
```

`ceil`(*arr*)

The largest integer $\geq x$, $\lceil x \rceil$

```
>>> print ceil(2.3)
3.0
>>> print ceil(array((-1.5,2,2.5)))
[-1. 2. 3.]
>>> from fipy.variables.variable import Variable
>>> ceil(Variable(value=(-1.5,2,2.5), unit="m**2"))
numerix.ceil(Variable(value=PhysicalField(array([-1.5, 2. , 2.5]),'m**2')))
>>> print ceil(Variable(value=(-1.5,2,2.5), unit="m**2"))
[-1. 2. 3.] m**2
```

`sign`(*arr*)

`exp`(*arr*)

Natural exponent of x, e^x

log(*arr*)

Natural logarithm of x, $\ln x \equiv \log_e x$

```
>>> print tostring(log(10), precision=3)
2.303
>>> print tostring(log(array((0.1,1,10))), precision=3)
[-2.303 0. 2.303]
>>> from fipy.variables.variable import Variable
>>> log(Variable(value=(0.1,1,10)))
numerix.log(Variable(value=array([ 0.1, 1. , 10. ])))
>>> print tostring(log(Variable(value=(0.1,1,10))), precision=3)
[-2.303 0. 2.303]
```

conjugate(*arr*)

Complex conjugate of $z = x + iy$, $z^\star = x - iy$

```
>>> print conjugate(3 + 4j) == 3 - 4j
True
>>> print allclose(conjugate(array((3 + 4j, -2j, 10))), (3 - 4j, 2j, 10))
1
>>> from fipy.variables.variable import Variable
>>> var = conjugate(Variable(value=(3 + 4j, -2j, 10), unit="ohm"))
>>> print var.getUnit()
<PhysicalUnit ohm>
>>> print allclose(var.getNumericValue(), (3 - 4j, 2j, 10))
1
```

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 Numeric.dot on an array of vectors

Test that Variables are returned as Variables.

```
>>> from fipy.meshes.grid2D import Grid2D
>>> mesh = Grid2D(nx=2, ny=1)
>>> from fipy.variables.cellVariable import CellVariable
>>> v1 = CellVariable(mesh=mesh, value=((0,1),(2,3)), rank=1)
>>> v2 = array(((0,1),(2, 3)))
>>> dot(v1, v2)._getVariableClass()
<class 'fipy.variables.cellVariable.CellVariable'>
>>> dot(v2, v1)._getVariableClass()
```

```
<class 'fipy.variables.cellVariable.CellVariable'>
>>> print rank(dot(v2, v1))
0
>>> print dot(v1, v2)
[ 4 10]
>>> dot(v1, v1)._getVariableClass()
<class 'fipy.variables.cellVariable.CellVariable'>
>>> print dot(v1, v1)
[ 4 10]
>>> type(dot(v2, v2))
<type 'numpy.ndarray'>
>>> print dot(v2, v2)
[ 4 10]
```

`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.

`allequal`(*first*, *second*)

Returns `true` if every element of `first` is equal to the corresponding element of `second`.

`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`.

`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`.

take(*a*, *indices*, *axis*=0, *fill_value*=None)

Selects the elements of `a` corresponding to `indices`.

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
```

obj2sctype(*rep*, *default*=None)

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`.

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`.

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`.

Variables

numpy_version = `'new'`

ALLOW_THREADS = `1`

BUFSIZE = `10000`

CLIP = `0`

ERR_CALL = `3`

ERR_DEFAULT = `0`

ERR_DEFAULT2 = `2084`

ERR_IGNORE = `0`

ERR_LOG = `5`

ERR_PRINT = `4`

ERR_RAISE = `2`

ERR_WARN = `1`

FLOATING_POINT_SUPPORT = `1`

FPE_DIVIDEBYZERO = `1`

FPE_INVALID = `8`

FPE_OVERFLOW = `2`

FPE_UNDERFLOW = `4`

*False_* = `False`

Inf = `inf`

Infinity = `inf`

MAXDIMS = `32`

NAN = `nan`

NINF = `-inf`

NZERO = `-0.0`

NaN = `nan`

PINF = inf

PZERO = 0.0

RAISE = 2

SHIFT_DIVIDEBYZERO = 0

SHIFT_INVALID = 9

SHIFT_OVERFLOW = 3

SHIFT_UNDERFLOW = 6

ScalarType = (<type 'int'>, <type 'float'>, <type 'complex'>, <type 'l...

*True_* = True

UFUNC_BUFSIZE_DEFAULT = 10000

UFUNC_PYVALS_NAME = 'UFUNC_PYVALS'

WRAP = 1

absolute = <ufunc 'absolute'>

add = <ufunc 'add'>

bitwise_and = <ufunc 'bitwise_and'>

bitwise_not = <ufunc 'invert'>

bitwise_or = <ufunc 'bitwise_or'>

bitwise_xor = <ufunc 'bitwise_xor'>

*c_* = <numpy.lib.index_tricks.CClass object at 0x1827e30>

cast = {<type 'numpy.int64'>: <function <lambda> at 0xed2730>, <...

conj = <ufunc 'conjugate'>

deg2rad = <ufunc 'deg2rad'>

degrees = <ufunc 'degrees'>

divide = <ufunc 'divide'>

e = 2.71828182846

equal = <ufunc 'equal'>

exp2 = <ufunc 'exp2'>

expm1 = <ufunc 'expm1'>

fabs = <ufunc 'fabs'>

floor_divide = <ufunc 'floor_divide'>

fmax = <ufunc 'fmax'>

fmin = <ufunc 'fmin'>

fmod = <ufunc 'fmod'>

frexp = <ufunc 'frexp'>

greater = <ufunc 'greater'>

greater_equal = <ufunc 'greater_equal'>

hypot = <ufunc 'hypot'>

index_exp = <numpy.lib.index_tricks.IndexExpression object at 0x1827eb0>

inf = inf

infty = inf

invert = <ufunc 'invert'>

isfinite = <ufunc 'isfinite'>

isinf = <ufunc 'isinf'>

isnan = <ufunc 'isnan'>

ldexp = <ufunc 'ldexp'>

left_shift = <ufunc 'left_shift'>

less = <ufunc 'less'>

less_equal = <ufunc 'less_equal'>

little_endian = True

log1p = <ufunc 'log1p'>

logaddexp = <ufunc 'logaddexp'>

logaddexp2 = <ufunc 'logaddexp2'>

logical_and = <ufunc 'logical_and'>

logical_not = <ufunc 'logical_not'>

logical_or = <ufunc 'logical_or'>

logical_xor = <ufunc 'logical_xor'>

maximum = <ufunc 'maximum'>

mgrid = <numpy.lib.index_tricks.nd_grid object at 0x181e950>

minimum = <ufunc 'minimum'>

mod = <ufunc 'remainder'>

modf = <ufunc 'modf'>

multiply = <ufunc 'multiply'>

nan = nan

nbytes = {<type 'numpy.int64'>: 8, <type 'numpy.int16'>: 2, <type ...

negative = <ufunc 'negative'>

newaxis = None

not_equal = <ufunc 'not_equal'>

ogrid = <numpy.lib.index_tricks.nd_grid object at 0x181ed30>

ones_like = <ufunc 'ones_like'>

pi = 3.14159265359

power = <ufunc 'power'>

*r_* = <numpy.lib.index_tricks.RClass object at 0x18270d0>

rad2deg = <ufunc 'rad2deg'>

radians = <ufunc 'radians'>

reciprocal = <ufunc 'reciprocal'>

remainder = <ufunc 'remainder'>

right_shift = <ufunc 'right_shift'>

rint = <ufunc 'rint'>

*s_* = <numpy.lib.index_tricks.IndexExpression object at 0x1827ef0>

sctypeDict = {0: <type 'numpy.bool_'>, 1: <type 'numpy.int8'>, 2: <typ...

sctypeNA = {'?': 'Bool', 'B': 'UInt8', 'Bool': <type 'numpy.bool_'>,...

sctypes = {'complex': [<type 'numpy.complex64'>, <type 'numpy.compl...

signbit = <ufunc 'signbit'>

square = <ufunc 'square'>

subtract = <ufunc 'subtract'>

true_divide = <ufunc 'true_divide'>

trunc = <ufunc 'trunc'>

typeDict = {0: <type 'numpy.bool_'>, 1: <type 'numpy.int8'>, 2: <typ...

typeNA = {'?': 'Bool', 'B': 'UInt8', 'Bool': <type 'numpy.bool_'>,...

typecodes = {'All': '?bhilqpBHILQPfdgFDGSUVO', 'AllFloat': 'fdgFDG', ...

8.12 Module fipy.tools.parser

Functions

`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.

8.13 Module fipy.tools.pysparseMatrix

8.14 Module fipy.tools.sparseMatrix

8.15 Module fipy.tools.test

8.16 Module fipy.tools.trilinosMatrix

8.17 Module fipy.tools.vector

Vector utility functions that are inexplicably absent from Numeric

Functions

putAdd(*vector*, *ids*, *additionVector*)

This is a temporary replacement for Numeric.put as it was not doing what we thought it was doing.

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])
```

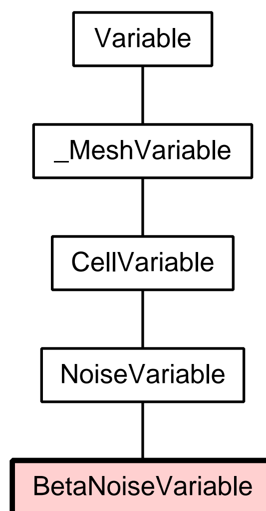
Package fipy.variables

9.1 Module fipy.variables.addOverFacesVariable

9.2 Module fipy.variables.arithmeticCellToFaceVariable

9.3 Module fipy.variables.betaNoiseVariable

Class BetaNoiseVariable



Represents a beta distribution of random numbers with the probability distribution

$$x^{\alpha-1}\frac{\beta^\alpha e^{-\beta x}}{\Gamma(\alpha)}$$

with a shape parameter α, a rate parameter β, and $\Gamma(z) = \int_0^\infty t^{z-1}e^{-t}\,dt$. We generate noise on a uniform cartesian mesh

```
>>> from fipy.variables.variable import Variable
>>> alpha = Variable()
```

```
>>> beta = Variable()
>>> from fipy.meshes.grid2D import Grid2D
>>> noise = BetaNoiseVariable(mesh = Grid2D(nx = 100, ny = 100), alpha = alpha, beta = beta)
```

We histogram the root-volume-weighted noise distribution

```
>>> from fipy.variables.histogramVariable import HistogramVariable
>>> histogram = HistogramVariable(distribution = noise, dx = 0.01, nx = 100)
```

and compare to a Gaussian distribution

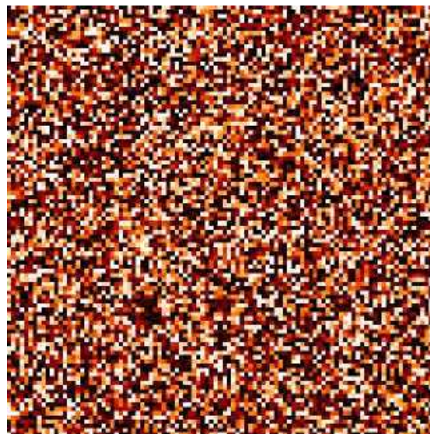
```
>>> from fipy.variables.cellVariable import CellVariable
>>> betadist = CellVariable(mesh = histogram.getMesh())
>>> x = histogram.getMesh().getCellCenters()[0]
```

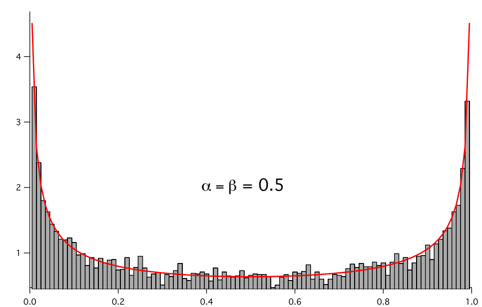
```
>>> if __name__ == '__main__':
...     from fipy import Viewer
...     viewer = Viewer(vars=noise, datamin=0, datamax=1)
...     histoplot = Viewer(vars=(histogram, betadist),
...                        datamin=0, datamax=1.5)
```

```
>>> from fipy.tools.numerix import arange, exp
>>> from scipy.special import gamma as Gamma
```

```
>>> for a in arange(0.5,5,0.5):
...     alpha.setValue(a)
...     for b in arange(0.5,5,0.5):
...         beta.setValue(b)
...         betadist.setValue((Gamma(alpha + beta) / (Gamma(alpha) * Gamma(beta)))
...                           * x**(alpha - 1) * (1 - x)**(beta - 1))
...         if __name__ == '__main__':
...             import sys
...             print >>sys.stderr, "alpha: %g, beta: %g" % (alpha, beta)
...             viewer.plot()
...             histoplot.plot()
```

```
>>> print abs(noise.getFaceGrad().getDivergence().getCellVolumeAverage()) < 5e-15
1
```





Methods

__init__(*self*, *mesh*, *alpha*, *beta*, *name*='', *hasOld*=0)

Create a `Variable`.

```
>>> Variable(value=3)
Variable(value=array(3))
>>> Variable(value=3, unit="m")
Variable(value=PhysicalField(3,'m'))
>>> Variable(value=3, unit="m", array=numerix.zeros((3,2)))
Variable(value=PhysicalField(array([[3, 3],
 [3, 3],
 [3, 3]]),'m'))
```

Parameters

mesh: The mesh on which to define the noise.

alpha: The parameter α.

beta: The parameter β.

Overrides: fipy.variables.variable.Variable.__init__()

Inherited from fipy.variables.noiseVariable.NoiseVariable: copy(), scramble()

Inherited from fipy.variables.cellVariable.CellVariable: __call__(), __getstate__(), __setstate__(), getArithmeticFaceValue(), getCellVolumeAverage(), getFaceGrad(), getFaceGradAverage(), getFaceValue(), getGaussGrad(), getGrad(), getHarmonicFaceValue(), getLeastSquaresGrad(), getMinmodFaceValue(), getOld(), updateOld()

Inherited from fipy.variables.meshVariable._MeshVariable: __repr__(), dot(), getMesh(), getRank(), getShape(), rdot(), setValue()

Inherited from fipy.variables.variable.Variable: __abs__(), __add__(), __and__(), __array__(), __array_wrap__(), __div__(), __eq__(), __float__(), __ge__(), __getitem__(), __gt__(), __int__(), __iter__(), __le__(), __len__(), __lt__(), __mod__(), __mul__(), __ne__(), __neg__(), __new__(), __nonzero__(), __or__(), __pos__(), __pow__(), __radd__(), __rdiv__(), __rmul__(), __rpow__(), __rsub__(), __setitem__(), __str__(), __sub__(), all(), allclose(), allequal(), any(), arccos(), arccosh(), arcsin(), arcsinh(), arctan(), arctan2(), arctanh(), cacheMe(), ceil(), conjugate(), cos(), cosh(), dontCacheMe(), exp(), floor(), getMag(), getName(), getNumericValue(), getSubscribedVariables(), getUnit(), getValue(), getsctype(), inBaseUnits(), inUnitsOf(), itemset(), log(), log10(), max(), min(), put(), reshape(), setName(), setUnit(), sign(), sin(), sinh(), sqrt(), sum(), take(), tan(), tanh(), tostring(), transpose()

Properties

Inherited from fipy.variables.variable.Variable: shape

Class Variables

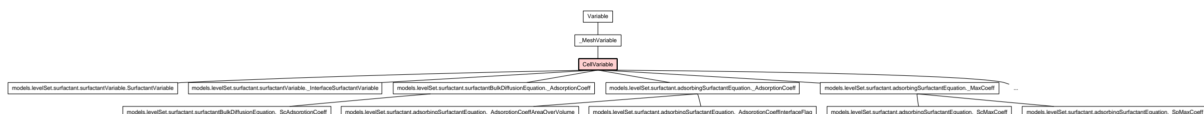
Inherited from fipy.variables.variable.Variable: __array_priority__

9.4 Module fipy.variables.binaryOperatorVariable

9.5 Module fipy.variables.cellToFaceVariable

9.6 Module fipy.variables.cellVariable

Class CellVariable



Known Subclasses: fipy.models.levelSet.surfactant.surfactantVariable.SurfactantVariable, fipy.models.levelSet.surfactant.surfactantVariable._InterfaceSurfactantVariable, fipy.models.levelSet.surfactant.surfactantBulkDiffusionEquation._AdsorptionCoeff, fipy.models.levelSet.surfactant.adsorbingSurfactantEquation._AdsorptionCoeff, fipy.models.levelSet.surfactant.adsorbingSurfactantEquation._MaxCoeff, fipy.models.levelSet.electroChem.metalIonSourceVariable._MetalIonSourceVariable, fipy.models.levelSet.distanceFunction.distanceVariable.DistanceVariable, fipy.variables.noiseVariable.NoiseVariable, fipy.variables.leastSquaresCellGradVariable._LeastSquaresCellGradVariable, fipy.variables.histogramVariable.HistogramVariable, fipy.variables.faceGradContributionsVariable._FaceGradContributions, fipy.variables.cellVariable._ReMeshedCellVariable, fipy.variables.modularVariable.ModularVariable, fipy.variables.addOverFacesVariable._AddOverFacesVariable, fipy.variables.gaussCellGradVariable._GaussCellGradVariable

Represents the field of values of a variable on a `Mesh`.

A `CellVariable` can be `pickled` to persistent storage (disk) for later use:

```
>>> from fipy.meshes.grid2D import Grid2D
>>> mesh = Grid2D(dx = 1., dy = 1., nx = 10, ny = 10)

>>> var = CellVariable(mesh = mesh, value = 1., hasOld = 1, name = 'test')
>>> x, y = mesh.getCellCenters()
>>> var.setValue(x * y)

>>> from fipy.tools import dump
>>> (f, filename) = dump.write(var, extension = '.gz')
>>> unPickledVar = dump.read(filename, f)

>>> print var.allclose(unPickledVar, atol = 1e-10, rtol = 1e-10)
1
```

Methods

`__init__`(*self*, *mesh*, *name*=`''`, *value*=`0.0`, *rank*=`None`, *elementshape*=`None`, *unit*=`None`, *hasOld*=`0`)

Create a `Variable`.

```
>>> Variable(value=3)
Variable(value=array(3))
>>> Variable(value=3, unit="m")
Variable(value=PhysicalField(3,'m'))
>>> Variable(value=3, unit="m", array=numerix.zeros((3,2)))
Variable(value=PhysicalField(array([[3, 3],
```

```
[3, 3],
[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

Overrides: fipy.variables.variable.Variable.__init__() *(inherited documentation)*

`copy`(*self*)

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]))
```

Overrides: fipy.variables.variable.Variable.copy() *(inherited documentation)*

`__call__`(*self*, *points*=None, *order*=0)

Interpolates the CellVariable to a set of points using a method that has a memory requirement on the order of Ncells by Npoints in general, but uses only Ncells when the CellVariable's mesh is a UniformGrid object.

Tests

```
>>> from fipy import *
>>> m = Grid2D(nx=3, ny=2)
>>> v = CellVariable(mesh=m, value=m.getCellCenters()[0])
>>> print v(((0., 1.1, 1.2), (0., 1., 1.)))
[ 0.5 1.5 1.5]
>>> print v(((0., 1.1, 1.2), (0., 1., 1.)), order=1)
[ 0.25 1.1 1.2 ]
>>> m0 = Grid2D(nx=2, ny=2, dx=1., dy=1.)
>>> m1 = Grid2D(nx=4, ny=4, dx=.5, dy=.5)
>>> x, y = m0.getCellCenters()
>>> v0 = CellVariable(mesh=m0, value=x * y)
>>> print v0(m1.getCellCenters())
[ 0.25 0.25 0.75 0.75 0.25 0.25 0.75 0.75 0.75 0.75 2.25 2.25
 0.75 0.75 2.25 2.25]
>>> print v0(m1.getCellCenters(), order=1)
[ 0.125 0.25 0.5 0.625 0.25 0.375 0.875 1. 0.5 0.875
 1.875 2.25 0.625 1. 2.25 2.625]
```

Parameters

points: A point or set of points in the format (X, Y, Z)

order: The order of interpolation, 0 or 1, default is 0

Overrides: fipy.variables.variable.Variable.__call__()

getCellVolumeAverage(*self*)

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.grid2D import Grid2D
>>> mesh = Grid2D(nx = 3, ny = 1, dx = .5, dy = .1)
>>> var = CellVariable(value = (1, 2, 6), mesh = mesh)
>>> print var.getCellVolumeAverage()
3.0
```

getGrad(*self*)

Return $\nabla\phi$ as a rank-1 `CellVariable` (first-order gradient).

getGaussGrad(*self*)

Return $\frac{1}{V_P}\sum_f \vec{n}\phi_f A_f$ as a rank-1 `CellVariable` (first-order gradient).

getLeastSquaresGrad(*self*)

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
>>> print CellVariable(mesh=Grid2D(nx=2, ny=2, dx=0.1, dy=2.0),
value=(0,1,3,6)).getLeastSquaresGrad()
[[8.0 8.0 24.0 24.0]
 [1.2 2.0 1.2 2.0]]
```

```
>>> from fipy import Grid1D
>>> print CellVariable(mesh=Grid1D(dx=(2.0, 1.0, 0.5)), value=(0, 1,
2)).getLeastSquaresGrad()
[[0.461538461538 0.8 1.2]]
```

getArithmeticFaceValue(*self*)

Returns a `FaceVariable` whose value corresponds to the arithmetic interpolation of the adjacent cells:

$$\phi_f = (\phi_1 - \phi_2)\frac{d_{f2}}{d_{12}} + \phi_2$$

```
>>> from fipy.meshes.grid1D import Grid1D
>>> from fipy import numerix
>>> mesh = Grid1D(dx = (1., 1.))
>>> var = CellVariable(mesh = mesh, value = (1, 2))
>>> faceValue = var.getArithmeticFaceValue()[mesh.getInteriorFaces().getValue()][0]
>>> answer = (var[0] - var[1]) * (0.5 / 1.) + var[1]
>>> numerix.allclose(faceValue, answer, atol = 1e-10, rtol = 1e-10)()
1
```

```
>>> mesh = Grid1D(dx = (2., 4.))
>>> var = CellVariable(mesh = mesh, value = (1, 2))
>>> faceValue = var.getArithmeticFaceValue()[mesh.getInteriorFaces().getValue()][0]
>>> answer = (var[0] - var[1]) * (1.0 / 3.0) + var[1]
>>> numerix.allclose(faceValue, answer, atol = 1e-10, rtol = 1e-10)()
1

>>> mesh = Grid1D(dx = (10., 100.))
>>> var = CellVariable(mesh = mesh, value = (1, 2))
>>> faceValue = var.getArithmeticFaceValue()[mesh.getInteriorFaces().getValue()][0]
>>> answer = (var[0] - var[1]) * (5.0 / 55.0) + var[1]
>>> numerix.allclose(faceValue, answer, atol = 1e-10, rtol = 1e-10)()
1
```

getFaceValue(*self*)

Returns a `FaceVariable` whose value corresponds to the arithmetic interpolation of the adjacent cells:

$$\phi_f = (\phi_1 - \phi_2)\frac{d_{f2}}{d_{12}} + \phi_2$$

```
>>> from fipy.meshes.grid1D import Grid1D
>>> from fipy import numerix
>>> mesh = Grid1D(dx = (1., 1.))
>>> var = CellVariable(mesh = mesh, value = (1, 2))
>>> faceValue = var.getArithmeticFaceValue()[mesh.getInteriorFaces().getValue()][0]
>>> answer = (var[0] - var[1]) * (0.5 / 1.) + var[1]
>>> numerix.allclose(faceValue, answer, atol = 1e-10, rtol = 1e-10)()
1

>>> mesh = Grid1D(dx = (2., 4.))
>>> var = CellVariable(mesh = mesh, value = (1, 2))
>>> faceValue = var.getArithmeticFaceValue()[mesh.getInteriorFaces().getValue()][0]
>>> answer = (var[0] - var[1]) * (1.0 / 3.0) + var[1]
>>> numerix.allclose(faceValue, answer, atol = 1e-10, rtol = 1e-10)()
1

>>> mesh = Grid1D(dx = (10., 100.))
>>> var = CellVariable(mesh = mesh, value = (1, 2))
>>> faceValue = var.getArithmeticFaceValue()[mesh.getInteriorFaces().getValue()][0]
>>> answer = (var[0] - var[1]) * (5.0 / 55.0) + var[1]
>>> numerix.allclose(faceValue, answer, atol = 1e-10, rtol = 1e-10)()
1
```

getMinmodFaceValue(*self*)

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)).getMinmodFaceValue()
[1 1 2]
>>> print CellVariable(mesh=Grid1D(nx=2), value=(-1, -2)).getMinmodFaceValue()
[-1 -1 -2]
>>> print CellVariable(mesh=Grid1D(nx=2), value=(-1, 2)).getMinmodFaceValue()
[-1 0 2]
```

getHarmonicFaceValue(*self*)

Returns a `FaceVariable` whose value corresponds to the harmonic interpolation of the adjacent cells:

$$\phi_f = \frac{\phi_1 \phi_2}{(\phi_2 - \phi_1)\frac{d_{f2}}{d_{12}} + \phi_1}$$

```
>>> from fipy.meshes.grid1D import Grid1D
>>> from fipy import numerix
>>> mesh = Grid1D(dx = (1., 1.))
>>> var = CellVariable(mesh = mesh, value = (1, 2))
>>> faceValue = var.getHarmonicFaceValue()[mesh.getInteriorFaces().getValue()]
>>> answer = var[0] * var[1] / ((var[1] - var[0]) * (0.5 / 1.) + var[0])
>>> numerix.allclose(faceValue, answer, atol = 1e-10, rtol = 1e-10)()
1

>>> mesh = Grid1D(dx = (2., 4.))
>>> var = CellVariable(mesh = mesh, value = (1, 2))
>>> faceValue = var.getHarmonicFaceValue()[mesh.getInteriorFaces().getValue()][0]
>>> answer = var[0] * var[1] / ((var[1] - var[0]) * (1.0 / 3.0) + var[0])
>>> numerix.allclose(faceValue, answer, atol = 1e-10, rtol = 1e-10)()
1

>>> mesh = Grid1D(dx = (10., 100.))
>>> var = CellVariable(mesh = mesh, value = (1, 2))
>>> faceValue = var.getHarmonicFaceValue()[mesh.getInteriorFaces().getValue()][0]
>>> answer = var[0] * var[1] / ((var[1] - var[0]) * (5.0 / 55.0) + var[0])
>>> numerix.allclose(faceValue, answer, atol = 1e-10, rtol = 1e-10)()
1
```

getFaceGrad(*self*)

Return $\nabla\phi$ as a rank-1 `FaceVariable` using differencing for the normal direction(second-order gradient).

getFaceGradAverage(*self*)

Return $\nabla\phi$ as a rank-1 `FaceVariable` using averaging for the normal direction(second-order gradient)

getOld(*self*)

Return the values of the `CellVariable` from the previous solution sweep.

Combinations of `CellVariable's` should also return old values.

```
>>> from fipy.meshes.grid1D import Grid1D
>>> mesh = Grid1D(nx = 2)
>>> from fipy.variables.cellVariable import CellVariable
>>> var1 = CellVariable(mesh = mesh, value = (2, 3), hasOld = 1)
>>> var2 = CellVariable(mesh = mesh, value = (3, 4))
>>> v = var1 * var2
>>> print v
[ 6 12]
>>> var1.setValue((3,2))
>>> print v
[9 8]
>>> print v.getOld()
[ 6 12]
```

The following small test is to correct for a bug when the operator does not just use variables.

```
>>> v1 = var1 * 3
>>> print v1
[9 6]
>>> print v1.getOld()
[6 9]
```

updateOld(*self*)

Set the values of the previous solution sweep to the current values.

__getstate__(*self*)

Used internally to collect the necessary information to `pickle` the `CellVariable` to persistent storage.

Overrides: fipy.variables.variable.Variable.__getstate__()

__setstate__(*self*, *dict*)

Used internally to create a new `CellVariable` from `pickled` persistent storage.

Overrides: fipy.variables.variable.Variable.__setstate__()

Inherited from fipy.variables.meshVariable._MeshVariable: __repr__(), dot(), getMesh(), getRank(), getShape(), rdot(), setValue()

Inherited from fipy.variables.variable.Variable: __abs__(), __add__(), __and__(), __array__(), __array_wrap__(), __div__(), __eq__(), __float__(), __ge__(), __getitem__(), __gt__(), __int__(), __iter__(), __le__(), __len__(), __lt__(), __mod__(), __mul__(), __ne__(), __neg__(), __new__(), __nonzero__(), __or__(), __pos__(), __pow__(), __radd__(), __rdiv__(), __rmul__(), __rpow__(), __rsub__(), __setitem__(), __str__(), __sub__(), all(), allclose(), allequal(), any(), arccos(), arccosh(), arcsin(), arcsinh(), arctan(), arctan2(), arctanh(), cacheMe(), ceil(), conjugate(), cos(), cosh(), dontCacheMe(), exp(), floor(), getMag(), getName(), getNumericValue(), getSubscribedVariables(), getUnit(), getValue(), getsctype(), inBaseUnits(), inUnitsOf(), itemset(), log(), log10(), max(), min(), put(), reshape(), setName(), setUnit(), sign(), sin(), sinh(), sqrt(), sum(), take(), tan(), tanh(), tostring(), transpose()

Properties

Inherited from fipy.variables.variable.Variable: shape

Class Variables

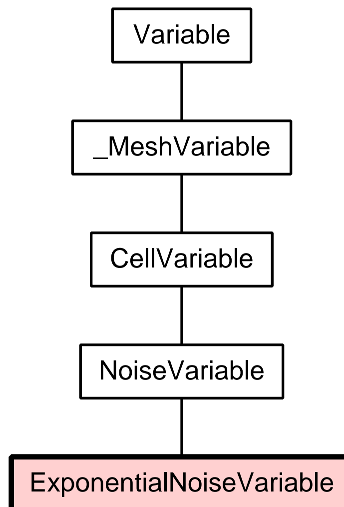
Inherited from fipy.variables.variable.Variable: __array_priority__

9.7 Module fipy.variables.cellVolumeAverageVariable

9.8 Module fipy.variables.constant

9.9 Module fipy.variables.exponentialNoiseVariable

Class ExponentialNoiseVariable



Represents an exponential distribution of random numbers with the probability distribution

$$\mu^{-1} e^{-\frac{x}{\mu}}$$

with a mean parameter μ. We generate noise on a uniform cartesian mesh

```
>>> from fipy.variables.variable import Variable
>>> mean = Variable()
>>> from fipy.meshes.grid2D import Grid2D
>>> noise = ExponentialNoiseVariable(mesh = Grid2D(nx = 100, ny = 100), mean = mean)
```

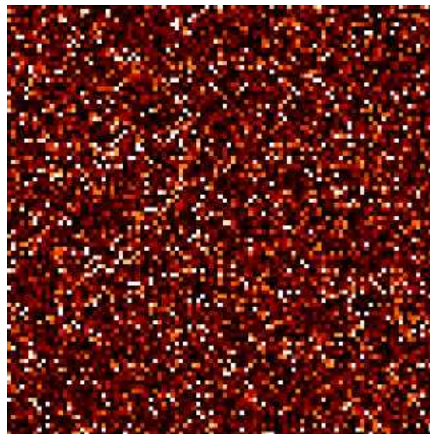
We histogram the root-volume-weighted noise distribution

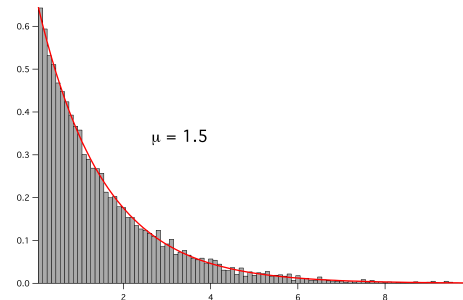
```
>>> from fipy.variables.histogramVariable import HistogramVariable
>>> histogram = HistogramVariable(distribution = noise, dx = 0.1, nx = 100)
```

and compare to a Gaussian distribution

```
>>> from fipy.variables.cellVariable import CellVariable
>>> expdist = CellVariable(mesh = histogram.getMesh())
>>> x = histogram.getMesh().getCellCenters()[0]
```

```
>>> if __name__ == '__main__':
...     from fipy import Viewer
...     viewer = Viewer(vars=noise, datamin=0, datamax=5)
...     histoplot = Viewer(vars=(histogram, expdist),
...                        datamin=0, datamax=1.5)

>>> from fipy.tools.numerix import arange, exp

>>> for mu in arange(0.5,3,0.5):
...     mean.setValue(mu)
...     expdist.setValue((1/mean)*exp(-x/mean))
...     if __name__ == '__main__':
...         import sys
...         print >>sys.stderr, "mean: %g" % mean
...         viewer.plot()
...         histoplot.plot()

>>> print abs(noise.getFaceGrad().getDivergence().getCellVolumeAverage()) < 5e-15
1
```





Methods

__init__(*self*, *mesh*, *mean*=0.0, *name*='', *hasOld*=0)

Create a `Variable`.

```
>>> Variable(value=3)
Variable(value=array(3))
>>> Variable(value=3, unit="m")
Variable(value=PhysicalField(3,'m'))
>>> Variable(value=3, unit="m", array=numerix.zeros((3,2)))
Variable(value=PhysicalField(array([[3, 3],
 [3, 3],
 [3, 3]]),'m'))
```

Parameters

mesh: The mesh on which to define the noise.

mean: The mean of the distribution μ.

Overrides: fipy.variables.variable.Variable.__init__()

Inherited from fipy.variables.noiseVariable.NoiseVariable: copy(), scramble()

Inherited from fipy.variables.cellVariable.CellVariable: __call__(), __getstate__(), __setstate__(), getArithmeticFaceValue(), getCellVolumeAverage(), getFaceGrad(), getFaceGradAverage(), getFaceValue(), getGaussGrad(), getGrad(), getHarmonicFaceValue(), getLeastSquaresGrad(), getMinmodFaceValue(), getOld(), updateOld()

Inherited from fipy.variables.meshVariable._MeshVariable: __repr__(), dot(), getMesh(), getRank(), getShape(), rdot(), setValue()

Inherited from fipy.variables.variable.Variable: __abs__(), __add__(), __and__(), __array__(), __array_wrap__(), __div__(), __eq__(), __float__(), __ge__(), __getitem__(), __gt__(), __int__(), __iter__(), __le__(), __len__(), __lt__(), __mod__(), __mul__(), __ne__(), __neg__(), __new__(), __nonzero__(), __or__(), __pos__(), __pow__(), __radd__(), __rdiv__(), __rmul__(), __rpow__(), __rsub__(), __setitem__(), __str__(), __sub__(), all(), allclose(), allequal(), any(), arccos(), arccosh(), arcsin(), arcsinh(), arctan(), arctan2(), arctanh(), cacheMe(), ceil(), conjugate(), cos(), cosh(), dontCacheMe(), exp(), floor(), getMag(), getName(), getNumericValue(), getSubscribedVariables(), getUnit(), getValue(), getsctype(), inBaseUnits(), inUnitsOf(), itemset(), log(), log10(), max(), min(), put(), reshape(), setName(), setUnit(), sign(), sin(), sinh(), sqrt(), sum(), take(), tan(), tanh(), tostring(), transpose()

Properties

Inherited from fipy.variables.variable.Variable: shape

Class Variables

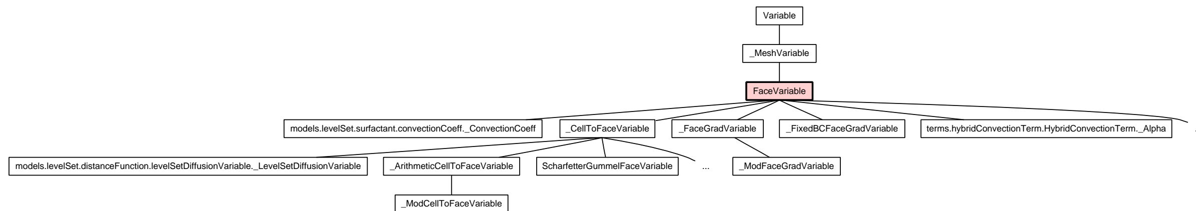
Inherited from fipy.variables.variable.Variable: __array_priority__

9.10 Module fipy.variables.faceGradContributionsVariable

9.11 Module fipy.variables.faceGradVariable

9.12 Module fipy.variables.faceVariable

Class FaceVariable



Known Subclasses: fipy.models.levelSet.surfactant.convectionCoeff._ConvectionCoeff, fipy.variables.cellToFaceVariable._CellToFaceVariable, fipy.variables.faceGradVariable._FaceGradVariable, fipy.variables.fixedBCFaceGradVariable._FixedBCFaceGradVariable, fipy.terms.hybridConvectionTerm.HybridConvectionTerm._Alpha, fipy.terms.upwindConvectionTerm.UpwindConvectionTerm._Alpha, fipy.terms.centralDiffConvectionTerm.CentralDifferenceConvectionTerm._Alpha, fipy.terms.powerLawConvectionTerm.PowerLawConvectionTerm._Alpha, fipy.terms.exponentialConvectionTerm.ExponentialConvectionTerm._Alpha

Methods

copy(*self*)

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]))
```

Overrides: fipy.variables.variable.Variable.copy() *(inherited documentation)*

getDivergence(*self*)

```
>>> from fipy.meshes.grid2D import Grid2D
>>> from fipy.variables.cellVariable import CellVariable
>>> mesh = Grid2D(nx=3, ny=2)
>>> var = CellVariable(mesh=mesh, value=range(mesh.getNumberOfCells()))
>>> print var.getFaceGrad().getDivergence()
[ 4. 3. 2. -2. -3. -4.]
```

Inherited from fipy.variables.meshVariable._MeshVariable: __getstate__(), __init__(), __repr__(), dot(), getMesh(), getRank(), getShape(), rdot(), setValue()

Inherited from fipy.variables.variable.Variable: __abs__(), __add__(), __and__(), __array__(), __array_wrap__(), __call__(), __div__(), __eq__(), __float__(), __ge__(), __getitem__(), __gt__(), __int__(), __iter__(), __le__(), __len__(), __lt__(), __mod__(), __mul__(), __ne__(), __neg__(), __new__(), __nonzero__(), __or__(), __pos__(), __pow__(), __radd__(), __rdiv__(), __rmul__(), __rpow__(), __rsub__(), __setitem__(), __setstate__(), __str__(), __sub__(), all(), allclose(), allequal(), any(), arccos(), arccosh(), arcsin(), arcsinh(), arctan(), arctan2(), arctanh(), cacheMe(), ceil(), conjugate(), cos(), cosh(), dontCacheMe(), exp(), floor(), getMag(), getName(), getNumericValue(), getSubscribedVariables(), getUnit(), getValue(), getsctype(), inBaseUnits(), inUnitsOf(), itemset(), log(), log10(), max(), min(), put(), reshape(), setName(), setUnit(), sign(), sin(), sinh(), sqrt(), sum(), take(), tan(), tanh(), tostring(), transpose()

Properties

Inherited from fipy.variables.variable.Variable: shape

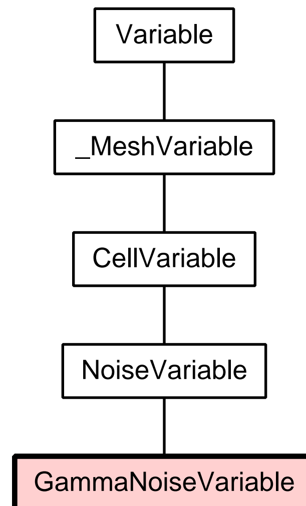
Class Variables

Inherited from fipy.variables.variable.Variable: __array_priority__

9.13 Module fipy.variables.fixedBCFaceGradVariable

9.14 Module fipy.variables.gammaNoiseVariable

Class GammaNoiseVariable



Represents a gamma distribution of random numbers with the probability distribution

$$x^{\alpha-1}\frac{\beta^{\alpha}e^{-\beta x}}{\Gamma(\alpha)}$$

with a shape parameter α, a rate parameter β, and $\Gamma(z) = \int_0^\infty t^{z-1}e^{-t}\,dt$. We generate noise on a uniform cartesian mesh

```
>>> from fipy.variables.variable import Variable
>>> alpha = Variable()
>>> beta = Variable()
>>> from fipy.meshes.grid2D import Grid2D
>>> noise = GammaNoiseVariable(mesh = Grid2D(nx = 100, ny = 100), shape = alpha, rate = beta)
```

We histogram the root-volume-weighted noise distribution

```
>>> from fipy.variables.histogramVariable import HistogramVariable
>>> histogram = HistogramVariable(distribution = noise, dx = 0.1, nx = 300)
```

and compare to a Gaussian distribution

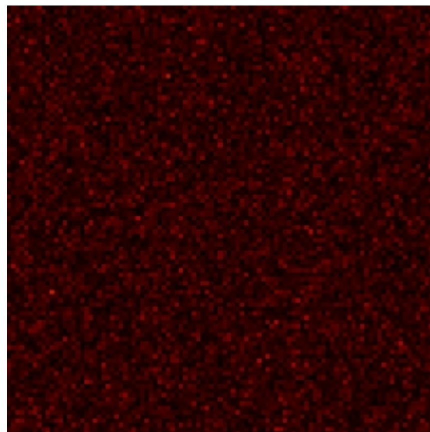
```
>>> from fipy.variables.cellVariable import CellVariable
>>> gammadist = CellVariable(mesh = histogram.getMesh())
>>> x = histogram.getMesh().getCellCenters()[0]
```

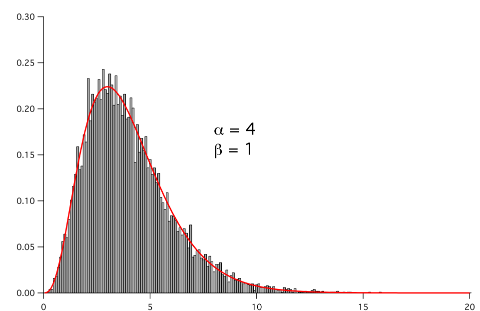
```
>>> if __name__ == '__main__':
...     from fipy import Viewer
...     viewer = Viewer(vars=noise, datamin=0, datamax=30)
...     histoplot = Viewer(vars=(histogram, gammadist),
...                        datamin=0, datamax=1)
```

```
>>> from fipy.tools.numerix import arange, exp
>>> from scipy.special import gamma as Gamma
```

```
>>> for shape in arange(1,8,1):
...     alpha.setValue(shape)
...     for rate in arange(0.5,2.5,0.5):
...         beta.setValue(rate)
...         gammadist.setValue(x**(alpha - 1) * (beta**alpha * exp(-beta * x)) / Gamma(alpha))
...         if __name__ == '__main__':
...             import sys
...             print >>sys.stderr, "alpha: %g, beta: %g" % (alpha, beta)
...             viewer.plot()
...             histoplot.plot()
```

```
>>> print abs(noise.getFaceGrad().getDivergence().getCellVolumeAverage()) < 5e-15
1
```





Methods

__init__(*self*, *mesh*, *shape*, *rate*, *name*='', *hasOld*=0)

Create a `Variable`.

```
>>> Variable(value=3)
Variable(value=array(3))
>>> Variable(value=3, unit="m")
Variable(value=PhysicalField(3,'m'))
>>> Variable(value=3, unit="m", array=numerix.zeros((3,2)))
Variable(value=PhysicalField(array([[3, 3],
 [3, 3],
 [3, 3]]),'m'))
```

Parameters

mesh: The mesh on which to define the noise.

shape: The shape parameter, α.

rate: The rate or inverse scale parameter, β.

Overrides: fipy.variables.variable.Variable.__init__()

Inherited from fipy.variables.noiseVariable.NoiseVariable: copy(), scramble()

Inherited from fipy.variables.cellVariable.CellVariable: __call__(), __getstate__(), __setstate__(), getArithmeticFaceValue(), getCellVolumeAverage(), getFaceGrad(), getFaceGradAverage(), getFaceValue(), getGaussGrad(), getGrad(), getHarmonicFaceValue(), getLeastSquaresGrad(), getMinmodFaceValue(), getOld(), updateOld()

Inherited from fipy.variables.meshVariable._MeshVariable: __repr__(), dot(), getMesh(), getRank(), getShape(), rdot(), setValue()

Inherited from fipy.variables.variable.Variable: __abs__(), __add__(), __and__(), __array__(), __array_wrap__(), __div__(), __eq__(), __float__(), __ge__(), __getitem__(), __gt__(), __int__(), __iter__(), __le__(), __len__(), __lt__(), __mod__(), __mul__(), __ne__(), __neg__(), __new__(), __nonzero__(), __or__(), __pos__(), __pow__(), __radd__(), __rdiv__(), __rmul__(), __rpow__(), __rsub__(), __setitem__(), __str__(), __sub__(), all(), allclose(), allequal(), any(), arccos(), arccosh(), arcsin(), arcsinh(), arctan(), arctan2(), arctanh(), cacheMe(), ceil(), conjugate(), cos(), cosh(), dontCacheMe(), exp(), floor(), getMag(), getName(), getNumericValue(), getSubscribedVariables(), getUnit(), getValue(), getsctype(), inBaseUnits(), inUnitsOf(), itemset(), log(), log10(), max(), min(), put(), reshape(), setName(), setUnit(), sign(), sin(), sinh(), sqrt(), sum(), take(), tan(), tanh(), tostring(), transpose()

Properties

Inherited from fipy.variables.variable.Variable: shape

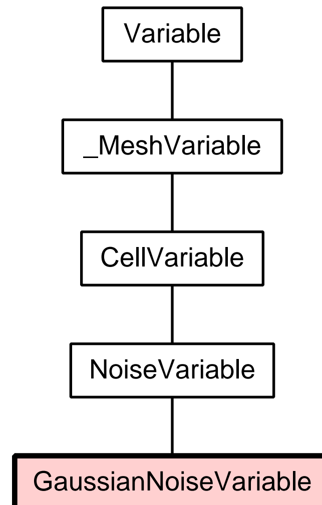
Class Variables

Inherited from fipy.variables.variable.Variable: `__array_priority__`

9.15 Module fipy.variables.gaussCellGradVariable

9.16 Module fipy.variables.gaussianNoiseVariable

Class GaussianNoiseVariable



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.getCellVolumes() * 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.
```

We generate noise on a non-uniform cartesian mesh with cell dimensions of x^2 and y^3.

```
>>> from fipy.meshes.grid2D import Grid2D
>>> mesh = Grid2D(dx = arange(0.1, 5., 0.1)**2, dy = arange(0.1, 3., 0.1)**3)
>>> noise = GaussianNoiseVariable(mesh = mesh, mean = mean,
...                               variance = variance / mesh.getCellVolumes())
```

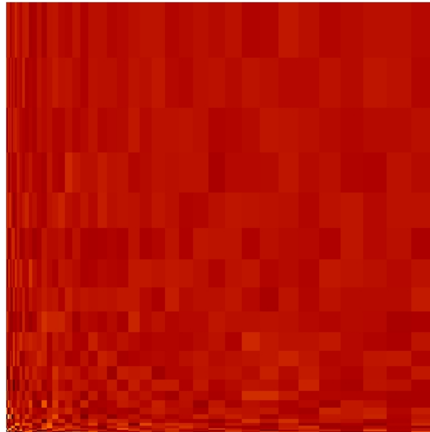
We histogram the root-volume-weighted noise distribution

```
>>> from fipy.variables.histogramVariable import HistogramVariable
>>> histogram = HistogramVariable(distribution = noise * sqrt(mesh.getCellVolumes()),
...                               dx = 0.1, nx = 600, offset = -30)
```

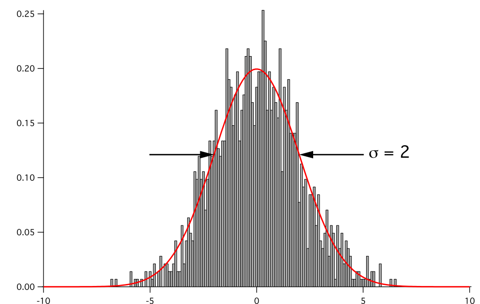
and compare to a Gaussian distribution

```
>>> from fipy.variables.cellVariable import CellVariable
>>> gauss = CellVariable(mesh = histogram.getMesh())
>>> x = histogram.getMesh().getCellCenters()[0]
>>> gauss.setValue((1/(sqrt(variance * 2 * pi))) * exp(-(x - mean)**2 / (2 * variance)))

>>> if __name__ == '__main__':
...     from fipy import viewers
...     viewer = Viewer(vars=noise,
...                     datamin=-5, datamax=5)
...     histoplot = Viewer(vars=(histogram, gauss))

>>> for i in range(10):
...     noise.scramble()
...     if __name__ == '__main__':
...         viewer.plot()
...         histoplot.plot()

>>> print abs(noise.getFaceGrad().getDivergence().getCellVolumeAverage()) < 5e-15
1
```

Note that the noise exhibits larger amplitude in the small cells than in the large ones



but that the root-volume-weighted histogram is Gaussian.



Methods

__init__(*self*, *mesh*, *name*='', *mean*=0.0, *variance*=1.0, *hasOld*=0)

Create a `Variable`.

```
>>> Variable(value=3)
Variable(value=array(3))
>>> Variable(value=3, unit="m")
Variable(value=PhysicalField(3,'m'))
>>> Variable(value=3, unit="m", array=numerix.zeros((3,2)))
Variable(value=PhysicalField(array([[3, 3],
 [3, 3],
 [3, 3]]),'m'))
```

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.

Overrides: fipy.variables.variable.Variable.__init__()

Inherited from fipy.variables.noiseVariable.NoiseVariable: copy(), scramble()

Inherited from fipy.variables.cellVariable.CellVariable: __call__(), __getstate__(), __setstate__(), getArithmeticFaceValue(), getCellVolumeAverage(), getFaceGrad(), getFaceGradAverage(), getFaceValue(), getGaussGrad(), getGrad(), getHarmonicFaceValue(), getLeastSquaresGrad(), getMinmodFaceValue(), getOld(), updateOld()

Inherited from fipy.variables.meshVariable._MeshVariable: __repr__(), dot(), getMesh(), getRank(), getShape(), rdot(), setValue()

Inherited from fipy.variables.variable.Variable: __abs__(), __add__(), __and__(), __array__(), __array_wrap__(), __div__(), __eq__(), __float__(), __ge__(), __getitem__(), __gt__(), __int__(), __iter__(), __le__(), __len__(), __lt__(), __mod__(), __mul__(), __ne__(), __neg__(), __new__(), __nonzero__(), __or__(), __pos__(), __pow__(), __radd__(), __rdiv__(), __rmul__(), __rpow__(), __rsub__(), __setitem__(), __str__(), __sub__(), all(), allclose(), allequal(), any(), arccos(), arccosh(), arcsin(), arcsinh(), arctan(), arctan2(), arctanh(), cacheMe(), ceil(), conjugate(), cos(), cosh(), dontCacheMe(), exp(), floor(), getMag(), getName(), getNumericValue(), getSubscribedVariables(), getUnit(), getValue(), getsctype(), inBaseUnits(), inUnitsOf(), itemset(), log(), log10(), max(), min(), put(), reshape(), setName(), setUnit(), sign(), sin(), sinh(), sqrt(), sum(), take(), tan(), tanh(), tostring(), transpose()

Properties

Inherited from fipy.variables.variable.Variable: shape

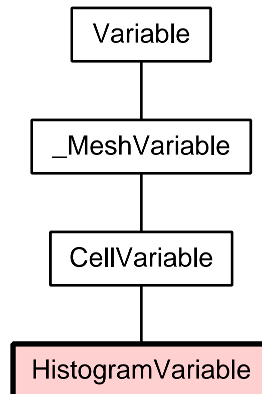
Class Variables

Inherited from fipy.variables.variable.Variable: __array_priority__

9.17 Module fipy.variables.harmonicCellToFaceVariable

9.18 Module fipy.variables.histogramVariable

Class HistogramVariable



Methods

__init__(*self*, *distribution*, *dx*=1.0, *nx*=None, *offset*=0.0)

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

Overrides: fipy.variables.variable.Variable.__init__()

Inherited from fipy.variables.cellVariable.CellVariable: __call__(), __getstate__(), __setstate__(), copy(), getArithmeticFaceValue(), getCellVolumeAverage(), getFaceGrad(), getFaceGradAverage(), getFaceValue(), getGaussGrad(), getGrad(), getHarmonicFaceValue(), getLeastSquaresGrad(), getMinmodFaceValue(), getOld(), updateOld()

Inherited from fipy.variables.meshVariable._MeshVariable: __repr__(), dot(), getMesh(), getRank(), getShape(), rdot(), setValue()

Inherited from fipy.variables.variable.Variable: __abs__(), __add__(), __and__(), __array__(), __array_wrap__(), __div__(), __eq__(), __float__(), __ge__(), __getitem__(), __gt__(), __int__(), __iter__(), __le__(), __len__(), __lt__(), __mod__(), __mul__(), __ne__(), __neg__(), __new__(), __nonzero__(), __or__(), __pos__(), __pow__(), __radd__(), __rdiv__(), __rmul__(), __rpow__(), __rsub__(), __setitem__(), __str__(), __sub__(), all(), allclose(), allequal(), any(), arccos(), arccosh(), arcsin(), arcsinh(), arctan(), arctan2(), arctanh(), cacheMe(), ceil(), conjugate(), cos(), cosh(), dontCacheMe(), exp(), floor(), getMag(), getName(), getNumericValue(), getSubscribedVariables(), getUnit(), getValue(), getsctype(), inBaseUnits(), inUnitsOf(), itemset(), log(), log10(), max(), min(), put(), reshape(), setName(), setUnit(), sign(), sin(), sinh(), sqrt(), sum(), take(), tan(), tanh(), tostring(), transpose()

Properties

Inherited from fipy.variables.variable.Variable: shape

Class Variables

Inherited from fipy.variables.variable.Variable: __array_priority__

9.19 Module fipy.variables.leastSquaresCellGradVariable

9.20 Module fipy.variables.meshVariable

9.21 Module fipy.variables.minmodCellToFaceVariable

9.22 Module fipy.variables.modCellGradVariable

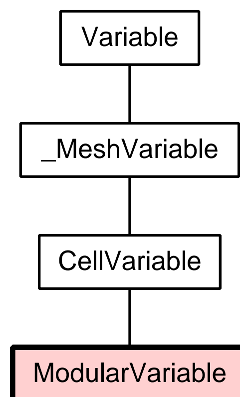
9.23 Module fipy.variables.modCellToFaceVariable

9.24 Module fipy.variables.modFaceGradVariable

9.25 Module fipy.variables.modPhysicalField

9.26 Module fipy.variables.modularVariable

Class ModularVariable



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.grid1D import Grid1D
>>> mesh = Grid1D(nx = 2)
>>> from fipy.tools import numerix
>>> pi = numerix.pi
>>> v1 = ModularVariable(mesh = mesh, value = (2*pi/3, -2*pi/3))
>>> v2 = ModularVariable(mesh = mesh, value = -2*pi/3)
>>> print numerix.allclose(v2 - v1, (2*pi/3, 0))
1
```

Obtaining the arithmetic face value.

```
>>> print numerix.allclose(v1.getArithmeticFaceValue(), (2*pi/3, -pi, -2*pi/3))
1
```

Obtaining the gradient.

```
>>> print numerix.allclose(v1.getGrad(), ((pi/3, pi/3),))
1
```

Obtaining the gradient at the faces.

```
>>> print numerix.allclose(v1.getFaceGrad(), ((0, 2*pi/3, 0),))
1
```

Obtaining the gradient at the faces but without modular arithmetic.

```
>>> print numerix.allclose(v1.getFaceGradNoMod(), ((0, -4*pi/3, 0),))
1
```

Methods

updateOld(*self*)

Set the values of the previous solution sweep to the current values. Test case due to bug.

```
>>> from fipy.meshes.grid1D import Grid1D
>>> mesh = Grid1D(nx = 1)
>>> var = ModularVariable(mesh=mesh, value=1., hasOld=1)
>>> var.updateOld()
>>> var[:] = 2
>>> print var.getOld()
[ 1.] 1
```

Overrides: fipy.variables.cellVariable.CellVariable.updateOld()

getGrad(*self*)

Return $\nabla\phi$ as a rank-1 `CellVariable` (first-order gradient). Adjusted for a `ModularVariable`

Overrides: fipy.variables.cellVariable.CellVariable.getGrad()

getArithmeticFaceValue(*self*)

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`

Overrides: fipy.variables.cellVariable.CellVariable.getArithmeticFaceValue()

`getFaceGrad(`*self*`)`

Return $\nabla \phi$ as a rank-1 `FaceVariable` (second-order gradient). Adjusted for a `ModularVariable`

Overrides: fipy.variables.cellVariable.CellVariable.getFaceGrad()

`getFaceGradNoMod(`*self*`)`

$\nabla \phi$ as a rank-1 `FaceVariable` (second-order gradient). Not adjusted for a `ModularVariable`

`__sub__(`*self*, *other*`)`

Overrides: fipy.variables.variable.Variable.__sub__()

`__rsub__(`*self*, *other*`)`

Overrides: fipy.variables.variable.Variable.__rsub__()

Inherited from fipy.variables.cellVariable.CellVariable: __call__(), __getstate__(), __init__(), __setstate__(), copy(), getCellVolumeAverage(), getFaceGradAverage(), getFaceValue(), getGaussGrad(), getHarmonicFaceValue(), getLeastSquaresGrad(), getMinmodFaceValue(), getOld()

Inherited from fipy.variables.meshVariable._MeshVariable: __repr__(), dot(), getMesh(), getRank(), getShape(), rdot(), setValue()

Inherited from fipy.variables.variable.Variable: __abs__(), __add__(), __and__(), __array__(), __array_wrap__(), __div__(), __eq__(), __float__(), __ge__(), __getitem__(), __gt__(), __int__(), __iter__(), __le__(), __len__(), __lt__(), __mod__(), __mul__(), __ne__(), __neg__(), __new__(), __nonzero__(), __or__(), __pos__(), __pow__(), __radd__(), __rdiv__(), __rmul__(), __rpow__(), __setitem__(), __str__(), all(), allclose(), allequal(), any(), arccos(), arccosh(), arcsin(), arcsinh(), arctan(), arctan2(), arctanh(), cacheMe(), ceil(), conjugate(), cos(), cosh(), dontCacheMe(), exp(), floor(), getMag(), getName(), getNumericValue(), getSubscribedVariables(), getUnit(), getValue(), getsctype(), inBaseUnits(), inUnitsOf(), itemset(), log(), log10(), max(), min(), put(), reshape(), setName(), setUnit(), sign(), sin(), sinh(), sqrt(), sum(), take(), tan(), tanh(), tostring(), transpose()

Properties

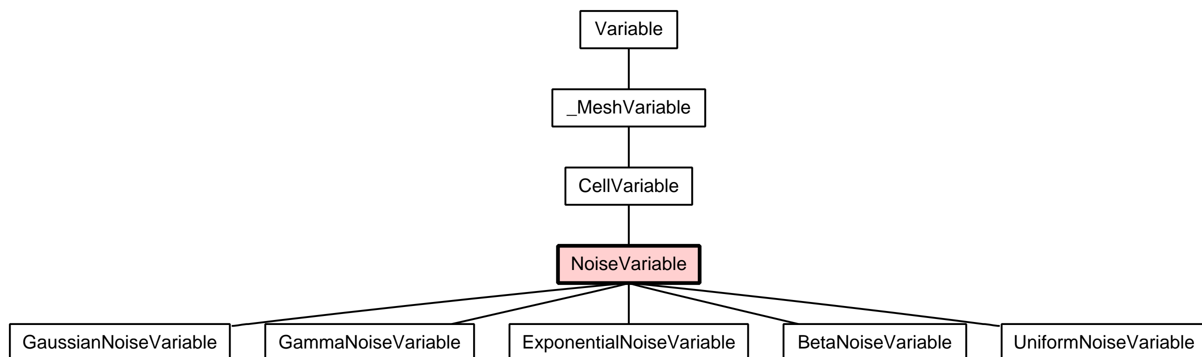
Inherited from fipy.variables.variable.Variable: shape

Class Variables

Inherited from fipy.variables.variable.Variable: __array_priority__

9.27 Module fipy.variables.noiseVariable

Class NoiseVariable



Known Subclasses: fipy.variables.gaussianNoiseVariable.GaussianNoiseVariable, fipy.variables.gammaNoiseVariable.GammaNoiseVariable, fipy.variables.exponentialNoiseVariable.ExponentialNoiseVariable, fipy.variables.betaNoiseVariable.BetaNoiseVariable, fipy.variables.uniformNoiseVariable.UniformNoiseVariable

Attention!

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

A generic base class for sources of noise distributed over the cells of a mesh.

In the event that the noise should be conserved, use:

```
<Specific>NoiseVariable(...).getFaceGrad().getDivergence()
```

The `seed()` and `get_seed()` functions of the `fipy.tools.numerix.random` module can be set and query the random number generated used by all `NoiseVariable` objects.

Methods

`__init__`(*self*, *mesh*, *name*='', *hasOld*=0)

Create a `Variable`.

```
>>> Variable(value=3)
Variable(value=array(3))
>>> Variable(value=3, unit="m")
Variable(value=PhysicalField(3,'m'))
>>> Variable(value=3, unit="m", array=numerix.zeros((3,2)))
Variable(value=PhysicalField(array([[3, 3],
 [3, 3],
 [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

Overrides: fipy.variables.variable.Variable.__init__() *(inherited documentation)*

`copy`(*self*)

Copy the value of the `NoiseVariable` to a static `CellVariable`.

Overrides: fipy.variables.variable.Variable.copy()

`scramble`(*self*)

Generate a new random distribution.

Inherited from fipy.variables.cellVariable.CellVariable: __call__(), __getstate__(), __setstate__(), getArithmeticFaceValue(), getCellVolumeAverage(), getFaceGrad(), getFaceGradAverage(), getFaceValue(), getGaussGrad(), getGrad(), getHarmonicFaceValue(), getLeastSquaresGrad(), getMinmodFaceValue(), getOld(), updateOld()

Inherited from fipy.variables.meshVariable._MeshVariable: __repr__(), dot(), getMesh(), getRank(), getShape(), rdot(), setValue()

Inherited from fipy.variables.variable.Variable: __abs__(), __add__(), __and__(), __array__(), __array_wrap__(), __div__(), __eq__(), __float__(), __ge__(), __getitem__(), __gt__(), __int__(), __iter__(), __le__(), __len__(), __lt__(), __mod__(), __mul__(), __ne__(), __neg__(), __new__(), __nonzero__(), __or__(), __pos__(), __pow__(), __radd__(), __rdiv__(), __rmul__(), __rpow__(), __rsub__(), __setitem__(), __str__(), __sub__(), all(), allclose(), allequal(), any(), arccos(), arccosh(), arcsin(), arcsinh(), arctan(), arctan2(), arctanh(), cacheMe(), ceil(), conjugate(), cos(), cosh(), dontCacheMe(), exp(), floor(), getMag(), getName(), getNumericValue(), getSubscribedVariables(), getUnit(), getValue(), getsctype(), inBaseUnits(), inUnitsOf(), itemset(), log(), log10(), max(), min(), put(), reshape(), setName(), setUnit(), sign(), sin(), sinh(), sqrt(), sum(), take(), tan(), tanh(), tostring(), transpose()

Properties

Inherited from fipy.variables.variable.Variable: shape

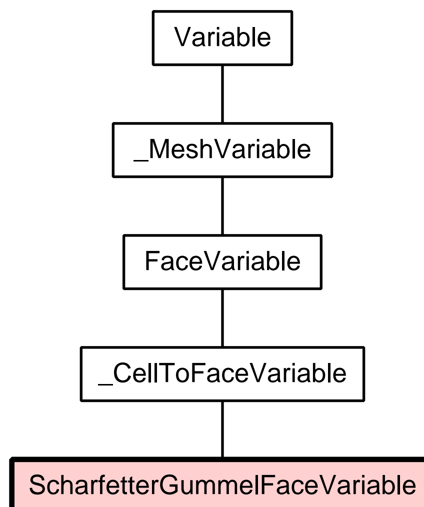
Class Variables

Inherited from fipy.variables.variable.Variable: __array_priority__

9.28 Module fipy.variables.operatorVariable

9.29 Module fipy.variables.scharfetterGummelFaceVariable

Class ScharfetterGummelFaceVariable



Methods

__init__(*self*, *var*, *boundaryConditions*=())

Create a `Variable`.

```
>>> Variable(value=3)
Variable(value=array(3))
>>> Variable(value=3, unit="m")
Variable(value=PhysicalField(3,'m'))
>>> Variable(value=3, unit="m", array=numerix.zeros((3,2)))
Variable(value=PhysicalField(array([[3, 3],
 [3, 3],
 [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

Overrides: fipy.variables.variable.Variable.__init__() *(inherited documentation)*

Inherited from fipy.variables.faceVariable.FaceVariable: copy(), getDivergence()

Inherited from fipy.variables.meshVariable._MeshVariable: __getstate__(), __repr__(), dot(), getMesh(), getRank(), getShape(), rdot(), setValue()

Inherited from fipy.variables.variable.Variable: __abs__(), __add__(), __and__(), __array__(), __array_wrap__(), __call__(), __div__(), __eq__(), __float__(), __ge__(), __getitem__(), __gt__(), __int__(), __iter__(), __le__(), __len__(), __lt__(), __mod__(), __mul__(), __ne__(), __neg__(), __new__(), __nonzero__(), __or__(), __pos__(), __pow__(), __radd__(), __rdiv__(), __rmul__(), __rpow__(), __rsub__(), __setitem__(), __setstate__(), __str__(), __sub__(), all(), allclose(), allequal(), any(), arccos(), arccosh(), arcsin(), arcsinh(), arctan(), arctan2(), arctanh(), cacheMe(), ceil(), conjugate(), cos(), cosh(), dontCacheMe(), exp(), floor(), getMag(), getName(), getNumericValue(), getSubscribedVariables(), getUnit(), getValue(), getsctype(), inBaseUnits(), inUnitsOf(), itemset(), log(), log10(), max(), min(), put(), reshape(), setName(), setUnit(), sign(), sin(), sinh(), sqrt(), sum(), take(), tan(), tanh(), tostring(), transpose()

Properties

Inherited from fipy.variables.variable.Variable: shape

Class Variables

Inherited from fipy.variables.variable.Variable: __array_priority__

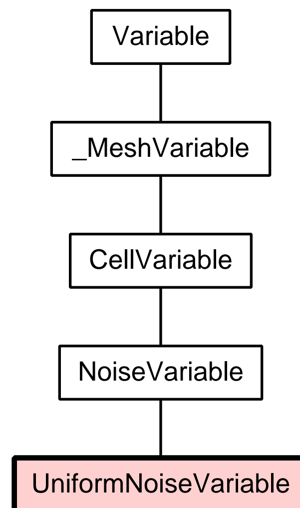
9.30 Module fipy.variables.test

Test numeric implementation of the mesh

9.31 Module fipy.variables.unaryOperatorVariable

9.32 Module fipy.variables.uniformNoiseVariable

Class UniformNoiseVariable



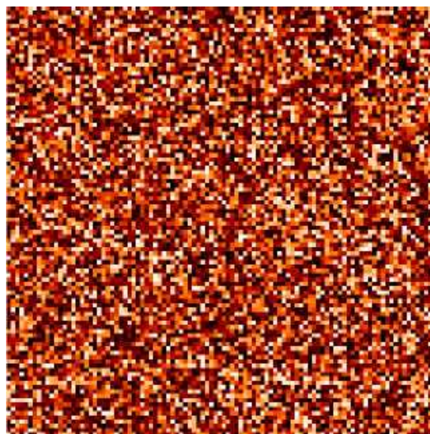
Represents a uniform distribution of random numbers.

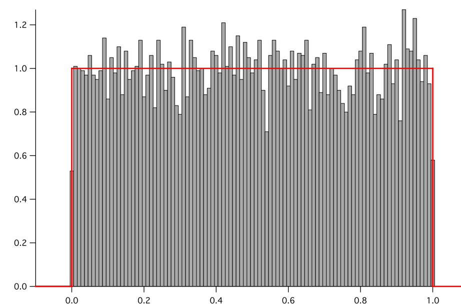
We generate noise on a uniform cartesian mesh

```
>>> from fipy.meshes.grid2D import Grid2D
>>> noise = UniformNoiseVariable(mesh = Grid2D(nx = 100, ny = 100))
```

and histogram the noise

```
>>> from fipy.variables.histogramVariable import HistogramVariable
>>> histogram = HistogramVariable(distribution = noise, dx = 0.01, nx = 120, offset = -.1)

>>> if __name__ == '__main__':
...     from fipy import viewers
...     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()
```





Methods

`__init__`(*self*, *mesh*, *name*=`''`, *minimum*=`0.0`, *maximum*=`1.0`, *hasOld*=`0`)

Create a `Variable`.

```
>>> Variable(value=3)
Variable(value=array(3))
>>> Variable(value=3, unit="m")
Variable(value=PhysicalField(3,'m'))
>>> Variable(value=3, unit="m", array=numerix.zeros((3,2)))
Variable(value=PhysicalField(array([[3, 3],
 [3, 3],
 [3, 3]]),'m'))
```

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.

Overrides: fipy.variables.variable.Variable.__init__()

Inherited from fipy.variables.noiseVariable.NoiseVariable: copy(), scramble()

Inherited from fipy.variables.cellVariable.CellVariable: __call__(), __getstate__(), __setstate__(), getArithmeticFaceValue(), getCellVolumeAverage(), getFaceGrad(), getFaceGradAverage(), getFaceValue(), getGaussGrad(), getGrad(), getHarmonicFaceValue(), getLeastSquaresGrad(), getMinmodFaceValue(), getOld(), updateOld()

Inherited from fipy.variables.meshVariable._MeshVariable: __repr__(), dot(), getMesh(), getRank(), getShape(), rdot(), setValue()

Inherited from fipy.variables.variable.Variable: __abs__(), __add__(), __and__(), __array__(), __array_wrap__(), __div__(), __eq__(), __float__(), __ge__(), __getitem__(), __gt__(), __int__(), __iter__(), __le__(), __len__(), __lt__(), __mod__(), __mul__(), __ne__(), __neg__(), __new__(), __nonzero__(), __or__(), __pos__(), __pow__(), __radd__(), __rdiv__(), __rmul__(), __rpow__(), __rsub__(), __setitem__(), __str__(), __sub__(), all(), allclose(), allequal(), any(), arccos(), arccosh(), arcsin(), arcsinh(), arctan(), arctan2(), arctanh(), cacheMe(), ceil(), conjugate(), cos(), cosh(), dontCacheMe(), exp(), floor(), getMag(), getName(), getNumericValue(), getSubscribedVariables(), getUnit(), getValue(), getsctype(), inBaseUnits(), inUnitsOf(), itemset(), log(), log10(), max(), min(), put(), reshape(), setName(), setUnit(), sign(), sin(), sinh(), sqrt(), sum(), take(), tan(), tanh(), tostring(), transpose()

Properties

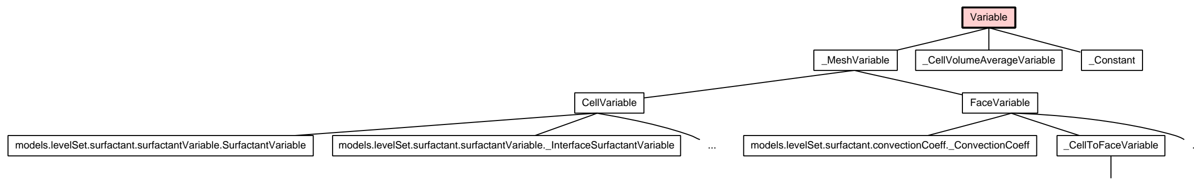
Inherited from fipy.variables.variable.Variable: shape

Class Variables

Inherited from fipy.variables.variable.Variable: __array_priority__

9.33 Module fipy.variables.variable

Class Variable



Known Subclasses: fipy.variables.meshVariable._MeshVariable, fipy.variables.cellVolumeAverageVariable._CellVolumeAverageVariable, fipy.variables.constant._Constant

Methods

`__new__`(*cls*, **args*, ***kwds*)

Overrides: object.`__new__`() *(inherited documentation)*

`__init__`(*self*, *value*=`0.0`, *unit*=`None`, *array*=`None`, *name*=`''`, *cached*=`1`)

Create a `Variable`.

```
>>> Variable(value=3)
Variable(value=array(3))
>>> Variable(value=3, unit="m")
Variable(value=PhysicalField(3,'m'))
>>> Variable(value=3, unit="m", array=numerix.zeros((3,2)))
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

Overrides: object.__init__()

__array_wrap__(*self*, *arr*, *context*=None)

Required to prevent numpy not calling the reverse binary operations. Both the following tests are examples ufuncs.

```
>>> print type(numerix.array([1.0, 2.0]) * Variable([1.0, 2.0]))
<class 'fipy.variables.binaryOperatorVariable.binOp'>

>>> from scipy.special import gamma as Gamma
>>> print type(Gamma(Variable([1.0, 2.0])))
<class 'fipy.variables.unaryOperatorVariable.unOp'>
```

__array__(*self*, *t*=None)

Attempt to convert the `Variable` to a numerix `array` object

```
>>> v = Variable(value=[2,3])
>>> print numerix.array(v)
[2 3]
```

A dimensional `Variable` will convert to the numeric value in the current units

```
>>> v = Variable(value=[2,3], unit="m")
>>> numerix.array(v)
array([2, 3])
```

copy(*self*)

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]))
```

`getUnit`(*self*)

Return the unit object of `self`.

```
>>> Variable(value="1 m").getUnit()
<PhysicalUnit m>
```

`setUnit`(*self*, *unit*)

Change the unit object of `self` to `unit`

```
>>> a = Variable(value="1 m")
>>> a.setUnit("m**2/s")
>>> print a
1.0 m**2/s
```

`inBaseUnits`(*self*)

Return the value of the `Variable` with all units reduced to their base SI elements.

```
>>> e = Variable(value="2.7 Hartree*Nav")
>>> print e.inBaseUnits()
7088849.01085 kg*m**2/s**2/mol
```

`inUnitsOf`(*self*, **units*)

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

```
>>> freeze = Variable('0 degC')
>>> print freeze.inUnitsOf('degF')
32.0 degF
```

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

```
>>> t = Variable(value=314159., unit='s')
>>> [str(element) for element in t.inUnitsOf('d','h','min','s')]
['3.0 d', '15.0 h', '15.0 min', '59.0 s']
```

getName(*self*)

setName(*self*, *name*)

__str__(*self*)

Overrides: object.__str__() *(inherited documentation)*

__repr__(*self*)

Overrides: object.__repr__() *(inherited documentation)*

tostring(*self*, *max_line_width*=75, *precision*=8, *suppress_small*=False, *separator*=' ')

__setitem__(*self*, *index*, *value*)

itemset(*self*, *value*)

put(*self*, *indices*, *value*)

__call__(*self*)

"Evaluate" the `Variable` and return its value

```
>>> a = Variable(value=3)
>>> print a()
3
>>> b = a + 4
>>> b
(Variable(value=array(3)) + 4)
>>> b()
7
```

getValue(*self*)

"Evaluate" the `Variable` and return its value (longhand)

```
>>> a = Variable(value=3)
>>> print a.getValue()
3
>>> b = a + 4
>>> b
(Variable(value=array(3)) + 4)
>>> b.getValue()
7
```

cacheMe(*self*, *recursive*=False)

dontCacheMe(*self*, *recursive*=False)

setValue(*self*, *value*, *unit*=None, *where*=None)

Set the value of the Variable. Can take a masked array.

```
>>> a = Variable((1,2,3))
>>> a.setValue(5, where=(1, 0, 1))
>>> print a
[5 2 5]

>>> b = Variable((4,5,6))
>>> a.setValue(b, where=(1, 0, 1))
>>> print a
[4 2 6]
>>> print b
[4 5 6]
>>> a.setValue(3)
>>> print a
[3 3 3]

>>> b = numerix.array((3,4,5))
>>> a.setValue(b)
>>> a[:] = 1
>>> print b
[3 4 5]

>>> a.setValue((4,5,6), where=(1, 0))
Traceback (most recent call last):
 ....
ValueError: shape mismatch: objects cannot be broadcast to a single shape
```

getNumericValue(*self*)

getShape(*self*)

```
>>> 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,)
```

`getsctype`(*self*, *default*=None)

Returns the Numpy sctype of the underlying array.

```
>>> Variable(1).getsctype()
<type 'numpy.int32'>
>>> Variable(1.).getsctype()
<type 'numpy.float64'>
>>> Variable((1,1.)).getsctype()
<type 'numpy.float64'>
```

`getSubscribedVariables`(*self*)

`__add__`(*self*, *other*)

`__radd__`(*self*, *other*)

`__sub__`(*self*, *other*)

`__rsub__`(*self*, *other*)

`__mul__`(*self*, *other*)

`__rmul__`(*self*, *other*)

__mod__(*self*, *other*)

__pow__(*self*, *other*)

__rpow__(*self*, *other*)

__div__(*self*, *other*)

__rdiv__(*self*, *other*)

__neg__(*self*)

__pos__(*self*)

__abs__(*self*)

Following test it to fix a bug with C inline string using abs() instead of fabs()

```
>>> print abs(Variable(2.3) - Variable(1.2))
1.1
```

`__lt__`(*self*, *other*)

Test if a `Variable` is less than another quantity

```
>>> a = Variable(value=3)
>>> b = (a < 4)
>>> b
(Variable(value=array(3)) < 4)
>>> b()
1
>>> a.setValue(4)
>>> print b()
0
>>> print 1000000000000000000 * Variable(1) < 1.
0
>>> print 1000 * Variable(1) < 1.
0
```

Python automatically reverses the arguments when necessary

```
>>> 4 > Variable(value=3)
(Variable(value=array(3)) < 4)
```

`__le__`(*self*, *other*)

Test if a `Variable` is less than or equal to another quantity

```
>>> a = Variable(value=3)
>>> b = (a <= 4)
>>> b
(Variable(value=array(3)) <= 4)
>>> b()
1
>>> a.setValue(4)
>>> print b()
1
>>> a.setValue(5)
>>> print b()
0
```

`__eq__`(*self*, *other*)

Test if a `Variable` is equal to another quantity

```
>>> a = Variable(value=3)
>>> b = (a == 4)
>>> b
(Variable(value=array(3)) == 4)
>>> b()
0
```

__ne__(*self*, *other*)

Test if a `Variable` is not equal to another quantity

```
>>> a = Variable(value=3)
>>> b = (a != 4)
>>> b
(Variable(value=array(3)) != 4)
>>> b()
1
```

__gt__(*self*, *other*)

Test if a `Variable` is greater than another quantity

```
>>> a = Variable(value=3)
>>> b = (a > 4)
>>> b
(Variable(value=array(3)) > 4)
>>> print b()
0
>>> a.setValue(5)
>>> print b()
1
```

__ge__(*self*, *other*)

Test if a `Variable` is greater than or equal to another quantity

```
>>> a = Variable(value=3)
>>> b = (a >= 4)
>>> b
(Variable(value=array(3)) >= 4)
>>> b()
0
```

```
>>> a.setValue(4)
>>> print b()
1
>>> a.setValue(5)
>>> print b()
1
```

__and__(*self*, *other*)

This test case has been added due to a weird bug that was appearing.

```
>>> a = Variable(value=(0, 0, 1, 1))
>>> b = Variable(value=(0, 1, 0, 1))
>>> numerix.equal((a == 0) & (b == 1), [False, True, False, False]).all()
1
>>> print a & b
[0 0 0 1]
>>> from fipy.meshes.grid1D import Grid1D
>>> mesh = Grid1D(nx=4)
>>> from fipy.variables.cellVariable import CellVariable
>>> a = CellVariable(value=(0, 0, 1, 1), mesh=mesh)
>>> b = CellVariable(value=(0, 1, 0, 1), mesh=mesh)
>>> numerix.equal((a == 0) & (b == 1), [False, True, False, False]).all()
1
>>> print a & b
[0 0 0 1]
```

__or__(*self*, *other*)

This test case has been added due to a weird bug that was appearing.

```
>>> a = Variable(value=(0, 0, 1, 1))
>>> b = Variable(value=(0, 1, 0, 1))
>>> numerix.equal((a == 0) | (b == 1), [True, True, False, True]).all()
1
>>> print a | b
[0 1 1 1]
>>> from fipy.meshes.grid1D import Grid1D
>>> mesh = Grid1D(nx=4)
>>> from fipy.variables.cellVariable import CellVariable
>>> a = CellVariable(value=(0, 0, 1, 1), mesh=mesh)
>>> b = CellVariable(value=(0, 1, 0, 1), mesh=mesh)
>>> numerix.equal((a == 0) | (b == 1), [True, True, False, True]).all()
1
>>> print a | b
[0 1 1 1]
```

__iter__(*self*)

__len__(*self*)

__float__(*self*)

__int__(*self*)

__nonzero__(*self*)

```
>>> print bool(Variable(value=0))
0
>>> print bool(Variable(value=(0, 0, 1, 1)))
Traceback (most recent call last):
 ...
ValueError: The truth value of an array with more than one element is ambiguous. Use a.any()
or a.all()
```

any(*self*, *axis*=None, *out*=None)

```
>>> print Variable(value=0).any()
0
>>> print Variable(value=(0, 0, 1, 1)).any()
1
```

all(*self*, *axis*=None, *out*=None)

```
>>> print Variable(value=(0, 0, 1, 1)).all()
0
>>> print Variable(value=(1, 1, 1, 1)).all()
1
```

arccos(*self*)

arccosh(*self*)

arcsin(*self*)

arcsinh(*self*)

sqrt(*self*)

```
>>> from fipy.meshes.grid1D import Grid1D
>>> mesh= Grid1D(nx=3)

>>> from fipy.variables.cellVariable import CellVariable
>>> var = CellVariable(mesh=mesh, value=((0., 2., 3.),), rank=1)
>>> print (var.dot(var)).sqrt()
[ 0. 2. 3.]
```

tan(*self*)

tanh(*self*)

`arctan(`*self*`)`

`arctanh(`*self*`)`

`exp(`*self*`)`

`log(`*self*`)`

`log10(`*self*`)`

`sin(`*self*`)`

`sinh(`*self*`)`

`cos(`*self*`)`

`cosh`(*self*)

`floor`(*self*)

`ceil`(*self*)

`sign`(*self*)

`conjugate`(*self*)

`arctan2`(*self*, *other*)

`dot`(*self*, *other*, *opShape*=`None`, *operatorClass*=`None`, *axis*=`0`)

`reshape`(*self*, *shape*)

`transpose`(*self*)

---

sum(*self*, *axis*=None)

max(*self*, *axis*=None)

min(*self*, *axis*=None)

__getitem__(*self*, *index*)

"Evaluate" the `Variable` and return the specified element

```
>>> a = Variable(value=((3.,4.),(5.,6.)), unit="m") + "4 m"
>>> print a[1,1]
10.0 m
```

It is an error to slice a `Variable` whose `value` is not sliceable

```
>>> Variable(value=3)[2]
Traceback (most recent call last):
 ...
IndexError: 0-d arrays can't be indexed
```

take(*self*, *ids*, *axis*=0)

allclose(*self*, *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))
False
```

`allequal`(*self*, *other*)

`getMag`(*self*)

`__getstate__`(*self*)

Used internally to collect the necessary information to `pickle` the `Variable` to persistent storage.

`__setstate__`(*self*, *dict*)

Used internally to create a new `Variable` from `pickled` persistent storage.

Properties

shape

Tuple of array dimensions.

Class Variables

__array_priority__ = `100.0`

Package fipy.viewers

10.1 Functions

Viewer(*vars*, *title*=None, *limits*={}, ***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', or 'tsv' 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

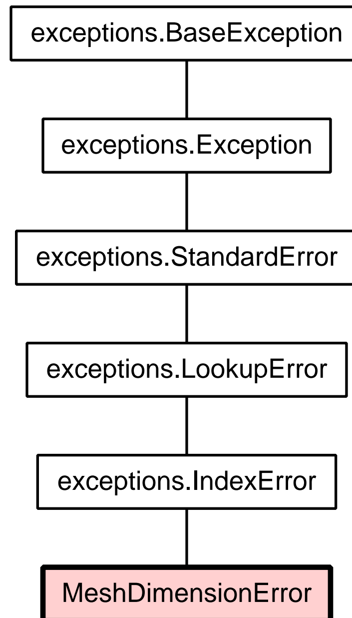
limits: a (deprecated) alternative to limit keyword arguments *(type=dict)*

xmin, xmax, ymin, ymax, zmin, zmax, datamin, datamax: displayed range of data. A 1D `Viewer` will only use `xmin` and `xmax`, a 2D viewer will also use `ymin` and `ymax`, and so on. All viewers will use `datamin` and `datamax`. Any limit set to a (default) value of `None` will autoscale.

make(**args*, ***kwargs*)

A deprecated synonym for `Viewer`

Class MeshDimensionError



Methods

Inherited from exceptions.IndexError: __init__(), __new__()

Inherited from exceptions.BaseException: __delattr__(), __getattribute__(), __getitem__(), __reduce__(), __repr__(), __setattr__(), __setstate__(), __str__()

Properties

Inherited from exceptions.BaseException: args, message

10.2 Package fipy.viewers.gistViewer

Functions

`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

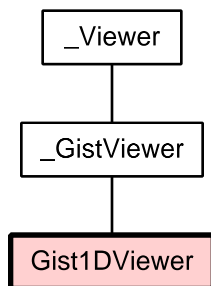
vars: a `CellVariable` or tuple of `CellVariable` objects to plot

title: displayed at the top of the `Viewer` window

limits: a (deprecated) alternative to limit keyword arguments *(type=dict)*

xmin, xmax, ymin, ymax, datamin, datamax: displayed range of data. A 1D `Viewer` will only use `xmin` and `xmax`, a 2D viewer will also use `ymin` and `ymax`. All viewers will use `datamin` and `datamax`. Any limit set to a (default) value of `None` will autoscale.

Class Gist1DViewer



Displays a y vs. x plot of one or more 1D `CellVariable` objects.

```
>>> from fipy import *
>>> mesh = Grid1D(nx=100)
>>> x, = mesh.getCellCenters()
>>> xVar = CellVariable(mesh=mesh, name="x", value=x)
>>> k = Variable(name="k", value=0.)
>>> viewer = Gist1DViewer(vars=(sin(k * xVar), cos(k * xVar / pi)),
...                       limits={'xmin': 10, 'xmax': 90},
...                       datamin=-0.9, datamax=2.0,
```

```
...                 title="Gist1DViewer test")
>>> for kval in numerix.arange(0,0.3,0.03):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()
```

Methods

__init__(*self*, *vars*, *title*=None, *xlog*=0, *ylog*=0, *style*='work.gs', *limits*={}, ***kwlimits*)

Creates a `Gist1DViewer`.

Parameters

vars: a `CellVariable` or tuple of `CellVariable` objects to plot

title: displayed at the top of the `Viewer` window

xlog: log scaling of x axis if `True`

ylog: log scaling of y axis if `True`

stye: the Gist stylefile to use.

limits: a (deprecated) alternative to limit keyword arguments *(type=dict)*

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*).

Overrides: fipy.viewers.viewer._Viewer.__init__()

plot(*self*, *filename*=None)

Update the display of the viewed variables.

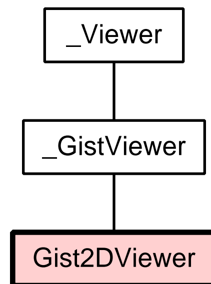
Parameters

filename: If not `None`, the name of a file to save the image into.

Overrides: fipy.viewers.viewer._Viewer.plot() *(inherited documentation)*

Inherited from fipy.viewers.viewer._Viewer: getVars(), plotMesh(), setLimits()

Class Gist2DViewer



Displays a contour plot of a 2D `CellVariable` object.

```
>>> from fipy import *
>>> mesh = Grid2D(nx=50, ny=100, dx=0.1, dy=0.01)
>>> x, y = mesh.getCellCenters()
>>> xyVar = CellVariable(mesh=mesh, name="x y", value=x * y)
>>> k = Variable(name="k", value=0.)
>>> viewer = Gist2DViewer(vars=sin(k * xyVar),
...                      limits={'ymin': 0.1, 'ymax': 0.9},
...                      datamin=-0.9, datamax=2.0,
...                      title="Gist2DViewer test")
>>> for kval in range(10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()
```

```
>>> from fipy import *
>>> mesh = (Grid2D(nx=5, ny=10, dx=0.1, dy=0.1)
...         + (Tri2D(nx=5, ny=5, dx=0.1, dy=0.1)
...          + ((0.5,), (0.2,))))
>>> x, y = mesh.getCellCenters()
>>> xyVar = CellVariable(mesh=mesh, name="x y", value=x * y)
>>> k = Variable(name="k", value=0.)
>>> viewer = Gist2DViewer(vars=sin(k * xyVar),
...                      limits={'ymin': 0.1, 'ymax': 0.9},
...                      datamin=-0.9, datamax=2.0,
...                      title="Gist2DViewer test")
>>> for kval in range(10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()
```

Methods

`__init__`(*self*, *vars*, *title*=`None`, *palette*=`'heat.gp'`, *grid*=`True`, *dpi*=`75`, *limits*={}, ***kwlimits*)

Creates a `Gist2DViewer`.

Parameters

vars: a `CellVariable` object.

title: displayed at the top of the `Viewer` window

palette: the color scheme to use for the image plot. Default is `heat.gp`. Another choice would be `rainbow.gp`.

grid: whether to show the grid lines in the plot.

limits: a (deprecated) alternative to limit keyword arguments *(type=dict)*

xmin, xmax, ymin, ymax, datamin, datamax: displayed range of data. Any limit set to a (default) value of `None` will autoscale.

Overrides: fipy.viewers.viewer._Viewer.__init__()

`plot`(*self*, *filename*=`None`)

Plot the `CellVariable` as a contour plot.

Parameters

filename: If not `None`, the name of a file to save the image into.

Overrides: fipy.viewers.viewer._Viewer.plot()

`plotMesh`(*self*, *filename*=`None`)

Display a representation of the mesh

Parameters

filename: If not `None`, the name of a file to save the image into.

Overrides: fipy.viewers.viewer._Viewer.plotMesh() *(inherited documentation)*

Inherited from fipy.viewers.viewer._Viewer: getVars(), setLimits()

Class GistVectorViewer

_Viewer
|
_GistViewer
|
GistVectorViewer

Displays a vector plot of a 2D rank-1 `CellVariable` or `FaceVariable` object using gist.

```
>>> from fipy import *
>>> mesh = Grid2D(nx=50, ny=100, dx=0.1, dy=0.01)
>>> x, y = mesh.getCellCenters()
>>> xyVar = CellVariable(mesh=mesh, name="x y", value=x * y)
>>> k = Variable(name="k", value=0.)
>>> viewer = GistVectorViewer(vars=sin(k * xyVar).getGrad(),
...                           title="GistVectorViewer test")
>>> for kval in range(10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()

>>> viewer = GistVectorViewer(vars=sin(k * xyVar).getFaceGrad(),
...                           title="GistVectorViewer test")
>>> for kval in range(10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()

>>> from fipy import *
>>> mesh = (Grid2D(nx=5, ny=10, dx=0.1, dy=0.1)
...         + (Tri2D(nx=5, ny=5, dx=0.1, dy=0.1)
...          + ((0.5,), (0.2,))))
>>> x, y = mesh.getCellCenters()
>>> xyVar = CellVariable(mesh=mesh, name="x y", value=x * y)
>>> k = Variable(name="k", value=0.)
>>> viewer = GistVectorViewer(vars=sin(k * xyVar).getGrad(),
...                           title="GistVectorViewer test")
```

```
>>> for kval in range(10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()
```

```
>>> viewer = GistVectorViewer(vars=sin(k * xyVar).getFaceGrad(),
...                   title="GistVectorViewer test")
>>> for kval in range(10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()
```

Methods

`__init__`(*self*, *vars*, *title*=`None`, *limits*={}, ***kwlimits*)

Creates a `GistVectorViewer`.

Parameters

vars: a rank-1 `CellVariable` or `FaceVariable` object.

title: displayed at the top of the `Viewer` window

limits: a (deprecated) alternative to limit keyword arguments *(type=dict)*

xmin, xmax, ymin, ymax, datamin, datamax: displayed range of data. Any limit set to a (default) value of `None` will autoscale.

Overrides: fipy.viewers.viewer._Viewer.__init__()

`plot`(*self*, *filename*=`None`)

Update the display of the viewed variables.

Parameters

filename: If not `None`, the name of a file to save the image into.

Overrides: fipy.viewers.viewer._Viewer.plot() *(inherited documentation)*

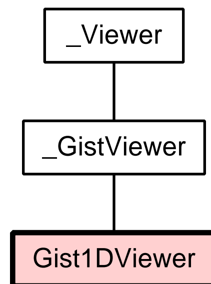
`getArray`(*self*)

Inherited from fipy.viewers.viewer._Viewer: getVars(), plotMesh(), setLimits()

10.3 Module fipy.viewers.gistViewer.colorbar

10.4 Module fipy.viewers.gistViewer.gist1DViewer

Class Gist1DViewer



Displays a y vs. x plot of one or more 1D `CellVariable` objects.

```
>>> from fipy import *
>>> mesh = Grid1D(nx=100)
>>> x, = mesh.getCellCenters()
>>> xVar = CellVariable(mesh=mesh, name="x", value=x)
>>> k = Variable(name="k", value=0.)
>>> viewer = Gist1DViewer(vars=(sin(k * xVar), cos(k * xVar / pi)),
...                  limits={'xmin': 10, 'xmax': 90},
...                  datamin=-0.9, datamax=2.0,
...                  title="Gist1DViewer test")
>>> for kval in numerix.arange(0,0.3,0.03):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()
```

Methods

__init__(*self*, *vars*, *title*=None, *xlog*=0, *ylog*=0, *style*='work.gs', *limits*={}, ***kwlimits*)

Creates a `Gist1DViewer`.

Parameters

vars: a `CellVariable` or tuple of `CellVariable` objects to plot

title: displayed at the top of the `Viewer` window

xlog: log scaling of x axis if `True`

ylog: log scaling of y axis if `True`

stye: the Gist stylefile to use.

limits: a (deprecated) alternative to limit keyword arguments *(type=dict)*

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*).

Overrides: fipy.viewers.viewer._Viewer.__init__()

`plot`(*self*, *filename*=`None`)

Update the display of the viewed variables.

Parameters

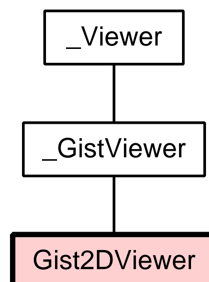
filename: If not `None`, the name of a file to save the image into.

Overrides: fipy.viewers.viewer._Viewer.plot() *(inherited documentation)*

Inherited from fipy.viewers.viewer._Viewer: getVars(), plotMesh(), setLimits()

10.5 Module fipy.viewers.gistViewer.gist2DViewer

Class Gist2DViewer



Displays a contour plot of a 2D `CellVariable` object.

```
>>> from fipy import *
>>> mesh = Grid2D(nx=50, ny=100, dx=0.1, dy=0.01)
>>> x, y = mesh.getCellCenters()
>>> xyVar = CellVariable(mesh=mesh, name="x y", value=x * y)
>>> k = Variable(name="k", value=0.)
>>> viewer = Gist2DViewer(vars=sin(k * xyVar),
...                  limits={'ymin': 0.1, 'ymax': 0.9},
...                  datamin=-0.9, datamax=2.0,
...                  title="Gist2DViewer test")
>>> for kval in range(10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()
```

```
>>> from fipy import *
>>> mesh = (Grid2D(nx=5, ny=10, dx=0.1, dy=0.1)
...         + (Tri2D(nx=5, ny=5, dx=0.1, dy=0.1)
...          + ((0.5,), (0.2,))))
>>> x, y = mesh.getCellCenters()
>>> xyVar = CellVariable(mesh=mesh, name="x y", value=x * y)
>>> k = Variable(name="k", value=0.)
>>> viewer = Gist2DViewer(vars=sin(k * xyVar),
...                  limits={'ymin': 0.1, 'ymax': 0.9},
...                  datamin=-0.9, datamax=2.0,
...                  title="Gist2DViewer test")
>>> for kval in range(10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()
```

Methods

`__init__`(*self*, *vars*, *title*=`None`, *palette*=`'heat.gp'`, *grid*=`True`, *dpi*=`75`, *limits*=`{}`, `**`*kwlimits*)

Creates a `Gist2DViewer`.

Parameters

vars: a `CellVariable` object.

title: displayed at the top of the `Viewer` window

palette: the color scheme to use for the image plot. Default is `heat.gp`. Another choice would be `rainbow.gp`.

grid: whether to show the grid lines in the plot.

limits: a (deprecated) alternative to limit keyword arguments *(type=dict)*

xmin, xmax, ymin, ymax, datamin, datamax: displayed range of data. Any limit set to a (default) value of `None` will autoscale.

Overrides: fipy.viewers.viewer._Viewer.__init__()

`plot`(*self*, *filename*=`None`)

Plot the `CellVariable` as a contour plot.

Parameters

filename: If not `None`, the name of a file to save the image into.

Overrides: fipy.viewers.viewer._Viewer.plot()

`plotMesh`(*self*, *filename*=`None`)

Display a representation of the mesh

Parameters

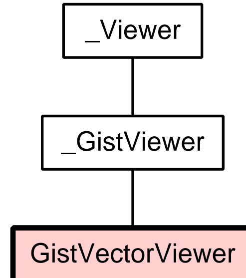
filename: If not `None`, the name of a file to save the image into.

Overrides: fipy.viewers.viewer._Viewer.plotMesh() *(inherited documentation)*

Inherited from fipy.viewers.viewer._Viewer: getVars(), setLimits()

10.6 Module fipy.viewers.gistViewer.gistVectorViewer

Class GistVectorViewer



Displays a vector plot of a 2D rank-1 `CellVariable` or `FaceVariable` object using gist.

```
>>> from fipy import *
>>> mesh = Grid2D(nx=50, ny=100, dx=0.1, dy=0.01)
>>> x, y = mesh.getCellCenters()
>>> xyVar = CellVariable(mesh=mesh, name="x y", value=x * y)
>>> k = Variable(name="k", value=0.)
>>> viewer = GistVectorViewer(vars=sin(k * xyVar).getGrad(),
...                   title="GistVectorViewer test")
>>> for kval in range(10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()

>>> viewer = GistVectorViewer(vars=sin(k * xyVar).getFaceGrad(),
...                   title="GistVectorViewer test")
>>> for kval in range(10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()

>>> from fipy import *
>>> mesh = (Grid2D(nx=5, ny=10, dx=0.1, dy=0.1)
...         + (Tri2D(nx=5, ny=5, dx=0.1, dy=0.1)
...          + ((0.5,), (0.2,))))
>>> x, y = mesh.getCellCenters()
>>> xyVar = CellVariable(mesh=mesh, name="x y", value=x * y)
>>> k = Variable(name="k", value=0.)
>>> viewer = GistVectorViewer(vars=sin(k * xyVar).getGrad(),
...                   title="GistVectorViewer test")
>>> for kval in range(10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()
```

```
>>> viewer = GistVectorViewer(vars=sin(k * xyVar).getFaceGrad(),
...                           title="GistVectorViewer test")
>>> for kval in range(10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()
```

Methods

`__init__`(*self*, *vars*, *title*=`None`, *limits*={}, ***kwlimits*)

Creates a `GistVectorViewer`.

Parameters

vars: a rank-1 `CellVariable` or `FaceVariable` object.

title: displayed at the top of the `Viewer` window

limits: a (deprecated) alternative to limit keyword arguments *(type=dict)*

xmin, xmax, ymin, ymax, datamin, datamax: displayed range of data. Any limit set to a (default) value of `None` will autoscale.

Overrides: fipy.viewers.viewer._Viewer.__init__()

`plot`(*self*, *filename*=`None`)

Update the display of the viewed variables.

Parameters

filename: If not `None`, the name of a file to save the image into.

Overrides: fipy.viewers.viewer._Viewer.plot() *(inherited documentation)*

`getArray`(*self*)

Inherited from fipy.viewers.viewer._Viewer: getVars(), plotMesh(), setLimits()

10.7 Module fipy.viewers.gistViewer.gistViewer

10.8 Module fipy.viewers.gistViewer.test

Test numeric implementation of the mesh

10.9 Package fipy.viewers.gnuplotViewer

Functions

`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

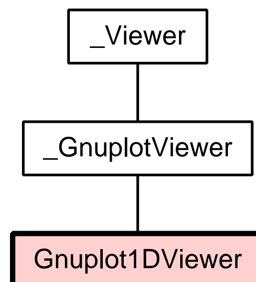
vars: a `CellVariable` or tuple of `CellVariable` objects to plot

title: displayed at the top of the Viewer window

limits: a (deprecated) alternative to limit keyword arguments *(type=dict)*

xmin, xmax, ymin, ymax, datamin, datamax: displayed range of data. A 1D `Viewer` will only use `xmin` and `xmax`, a 2D viewer will also use `ymin` and `ymax`. All viewers will use `datamin` and `datamax`. Any limit set to a (default) value of `None` will autoscale.

Class Gnuplot1DViewer



Displays a y vs. x plot of one or more 1D `CellVariable` objects.

The `Gnuplot1DViewer` plots a 1D `CellVariable` using a front end python wrapper available to download (Gnuplot.py).

```
>>> from fipy import *
>>> mesh = Grid1D(nx=100)
>>> x, = mesh.getCellCenters()
>>> xVar = CellVariable(mesh=mesh, name="x", value=x)
>>> k = Variable(name="k", value=0.)
```

```
>>> viewer = Gnuplot1DViewer(vars=(sin(k * xVar), cos(k * xVar / pi)),
...                 limits={'xmin': 10, 'xmax': 90},
...                 datamin=-0.9, datamax=2.0,
...                 title="Gnuplot1DViewer test")
>>> for kval in numerix.arange(0,0.3,0.03):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()
```

Different style script demos are available at the Gnuplot site.

Note

Gnuplot1DViewer requires Gnuplot version 4.0.

Methods

Inherited from fipy.viewers.gnuplotViewer.gnuplotViewer._GnuplotViewer: __init__(), plot()

Inherited from fipy.viewers.viewer._Viewer: getVars(), plotMesh(), setLimits()

Class Gnuplot2DViewer

_Viewer

_GnuplotViewer

Gnuplot2DViewer

Displays a contour plot of a 2D CellVariable object.

The Gnuplot2DViewer plots a 2D CellVariable using a front end python wrapper available to download (Gnuplot.py).

```
>>> from fipy import *
>>> mesh = Grid2D(nx=50, ny=100, dx=0.1, dy=0.01)
>>> x, y = mesh.getCellCenters()
>>> xyVar = CellVariable(mesh=mesh, name="x y", value=x * y)
>>> k = Variable(name="k", value=0.)
>>> viewer = Gnuplot2DViewer(vars=sin(k * xyVar),
...                 limits={'ymin': 0.1, 'ymax': 0.9},
...                 datamin=-0.9, datamax=2.0,
```

```
...                  title="Gnuplot2DViewer test")
>>> for kval in range(10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()
```

```
>>> from fipy import *
>>> mesh = (Grid2D(nx=5, ny=10, dx=0.1, dy=0.1)
...         + (Tri2D(nx=5, ny=5, dx=0.1, dy=0.1)
...          + ((0.5,), (0.2,))))
>>> x, y = mesh.getCellCenters()
>>> xyVar = CellVariable(mesh=mesh, name="x y", value=x * y)
>>> k = Variable(name="k", value=0.)
>>> viewer = Gnuplot2DViewer(vars=sin(k * xyVar),
...                  limits={'ymin': 0.1, 'ymax': 0.9},
...                  datamin=-0.9, datamax=2.0,
...                  title="Gnuplot2DViewer test")
>>> for kval in range(10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()
```

Different style script demos are available at the Gnuplot site.

Note

`Gnuplot2DViewer` requires Gnuplot version 4.0.

Methods

`__init__`(*self*, *vars*, *title*=`None`, *limits*={}, ***kwlimits*)

Creates a `Gnuplot2DViewer`.

Parameters

vars: a `CellVariable` object.

title: displayed at the top of the `Viewer` window

limits: a (deprecated) alternative to limit keyword arguments *(type=dict)*

xmin, xmax, ymin, ymax, datamin, datamax: displayed range of data. Any limit set to a (default) value of `None` will autoscale.

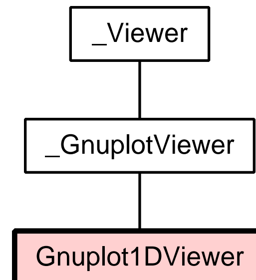
Overrides: fipy.viewers.viewer._Viewer.__init__()

Inherited from fipy.viewers.gnuplotViewer.gnuplotViewer._GnuplotViewer: plot()

Inherited from fipy.viewers.viewer._Viewer: getVars(), plotMesh(), setLimits()

10.10 Module fipy.viewers.gnuplotViewer.gnuplot1DViewer

Class Gnuplot1DViewer



Displays a y vs. x plot of one or more 1D `CellVariable` objects.

The `Gnuplot1DViewer` plots a 1D `CellVariable` using a front end python wrapper available to download (Gnuplot.py).

```
>>> from fipy import *
>>> mesh = Grid1D(nx=100)
>>> x, = mesh.getCellCenters()
>>> xVar = CellVariable(mesh=mesh, name="x", value=x)
>>> k = Variable(name="k", value=0.)
>>> viewer = Gnuplot1DViewer(vars=(sin(k * xVar), cos(k * xVar / pi)),
...                 limits={'xmin': 10, 'xmax': 90},
...                 datamin=-0.9, datamax=2.0,
...                 title="Gnuplot1DViewer test")
>>> for kval in numerix.arange(0,0.3,0.03):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()
```

Different style script demos are available at the Gnuplot site.

Note

`Gnuplot1DViewer` requires Gnuplot version 4.0.

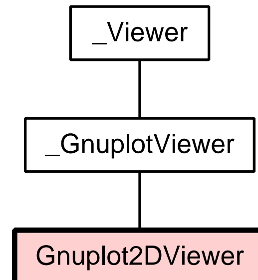
Methods

Inherited from fipy.viewers.gnuplotViewer.gnuplotViewer._GnuplotViewer: __init__(), plot()

Inherited from fipy.viewers.viewer._Viewer: getVars(), plotMesh(), setLimits()

10.11 Module fipy.viewers.gnuplotViewer.gnuplot2DViewer

Class Gnuplot2DViewer



Displays a contour plot of a 2D `CellVariable` object.

The `Gnuplot2DViewer` plots a 2D `CellVariable` using a front end python wrapper available to download (Gnuplot.py).

```
>>> from fipy import *
>>> mesh = Grid2D(nx=50, ny=100, dx=0.1, dy=0.01)
>>> x, y = mesh.getCellCenters()
>>> xyVar = CellVariable(mesh=mesh, name="x y", value=x * y)
>>> k = Variable(name="k", value=0.)
>>> viewer = Gnuplot2DViewer(vars=sin(k * xyVar),
...                 limits={'ymin': 0.1, 'ymax': 0.9},
...                 datamin=-0.9, datamax=2.0,
...                 title="Gnuplot2DViewer test")
>>> for kval in range(10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()
```

```
>>> from fipy import *
>>> mesh = (Grid2D(nx=5, ny=10, dx=0.1, dy=0.1)
...         + (Tri2D(nx=5, ny=5, dx=0.1, dy=0.1)
...          + ((0.5,), (0.2,))))
>>> x, y = mesh.getCellCenters()
>>> xyVar = CellVariable(mesh=mesh, name="x y", value=x * y)
>>> k = Variable(name="k", value=0.)
>>> viewer = Gnuplot2DViewer(vars=sin(k * xyVar),
...                 limits={'ymin': 0.1, 'ymax': 0.9},
...                 datamin=-0.9, datamax=2.0,
...                 title="Gnuplot2DViewer test")
>>> for kval in range(10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()
```

Different style script demos are available at the Gnuplot site.

Note

`Gnuplot2DViewer` requires Gnuplot version 4.0.

Methods

__init__(*self*, *vars*, *title*=None, *limits*={}, ***kwlimits*)

Creates a `Gnuplot2DViewer`.

Parameters

vars: a `CellVariable` object.

title: displayed at the top of the `Viewer` window

limits: a (deprecated) alternative to limit keyword arguments *(type=dict)*

xmin, xmax, ymin, ymax, datamin, datamax: displayed range of data. Any limit set to a (default) value of `None` will autoscale.

Overrides: fipy.viewers.viewer._Viewer.__init__()

Inherited from fipy.viewers.gnuplotViewer.gnuplotViewer._GnuplotViewer: plot()

Inherited from fipy.viewers.viewer._Viewer: getVars(), plotMesh(), setLimits()

10.12 Module fipy.viewers.gnuplotViewer.gnuplotViewer

10.13 Module fipy.viewers.gnuplotViewer.test

Test numeric implementation of the mesh

10.14 Package fipy.viewers.matplotlibViewer

Functions

`MatplotlibViewer(`*vars*, *title*=`None`, *limits*={}, ***kwlimits*`)`

Generic function for creating a `MatplotlibViewer`.

The `MatplotlibViewer` factory will search the module tree and return an instance of the first `MatplotlibViewer` it finds of the correct dimension and rank.

Parameters

vars: a `CellVariable` or tuple of `CellVariable` objects to plot

title: displayed at the top of the `Viewer` window

limits: a (deprecated) alternative to limit keyword arguments *(type=dict)*

xmin, xmax, ymin, ymax, datamin, datamax: displayed range of data. A 1D `Viewer` will only use `xmin` and `xmax`, a 2D viewer will also use `ymin` and `ymax`. All viewers will use `datamin` and `datamax`. Any limit set to a (default) value of `None` will autoscale.

Class Matplotlib1DViewer

_Viewer → _MatplotlibViewer → Matplotlib1DViewer

Displays a y vs. x plot of one or more 1D `CellVariable` objects using Matplotlib.

```
>>> from fipy import *
>>> mesh = Grid1D(nx=100)
>>> x, = mesh.getCellCenters()
>>> xVar = CellVariable(mesh=mesh, name="x", value=x)
>>> k = Variable(name="k", value=0.)
>>> viewer = Matplotlib1DViewer(vars=(sin(k * xVar), cos(k * xVar / pi)),
...                             limits={'xmin': 10, 'xmax': 90},
```

```
...                  datamin=-0.9, datamax=2.0,
...                  title="Matplotlib1DViewer test")
>>> for kval in numerix.arange(0,0.3,0.03):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()
```

Methods

`__init__`(*self*, *vars*, *title*=`None`, *xlog*=`False`, *ylog*=`False`, *limits*={}, ***kwlimits*)

Create a `_MatplotlibViewer`.

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: a (deprecated) alternative to limit keyword arguments *(type=dict)*

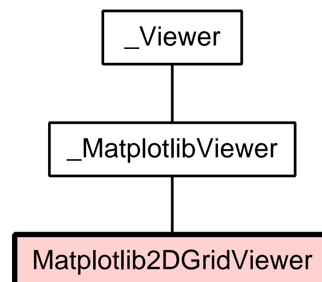
xmin, xmax, datamin, datamax: displayed range of data. Any limit set to a (default) value of `None` will autoscale. (*ymin* and *ymax* are synonyms for *datamin* and *datamax*).

Overrides: fipy.viewers.viewer._Viewer.__init__()

Inherited from fipy.viewers.matplotlibViewer.matplotlibViewer._MatplotlibViewer: plot()

Inherited from fipy.viewers.viewer._Viewer: getVars(), plotMesh(), setLimits()

Class Matplotlib2DGridViewer



Displays an image plot of a 2D `CellVariable` object using Matplotlib.

```
>>> from fipy import *
>>> mesh = Grid2D(nx=50, ny=100, dx=0.1, dy=0.01)
>>> x, y = mesh.getCellCenters()
>>> xyVar = CellVariable(mesh=mesh, name="x y", value=x * y)
>>> k = Variable(name="k", value=0.)
>>> viewer = Matplotlib2DGridViewer(vars=sin(k * xyVar),
...                 limits={'ymin': 0.1, 'ymax': 0.9},
...                 datamin=-0.9, datamax=2.0,
...                 title="Matplotlib2DGridViewer test")
>>> for kval in range(10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()
```

Methods

`__init__`(*self*, *vars*, *title*=`None`, *limits*={}, *cmap*=`None`, `**`*kwlimits*)

Creates a `Matplotlib2DGridViewer`.

Parameters

vars: A `CellVariable` object.

title: displayed at the top of the `Viewer` window

limits: a (deprecated) alternative to limit keyword arguments *(type=dict)*

cmap: The colormap. Defaults to `pylab.cm.jet`

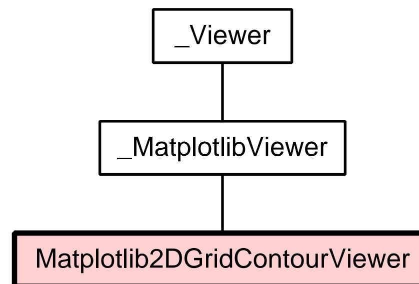
xmin, xmax, ymin, ymax, datamin, datamax: displayed range of data. Any limit set to a (default) value of `None` will autoscale.

Overrides: fipy.viewers.viewer._Viewer.__init__()

Inherited from fipy.viewers.matplotlibViewer.matplotlibViewer._MatplotlibViewer: plot()

Inherited from fipy.viewers.viewer._Viewer: getVars(), plotMesh(), setLimits()

Class Matplotlib2DGridContourViewer



Displays a contour plot of a 2D `CellVariable` object.

The `Matplotlib2DGridContourViewer` plots a 2D `CellVariable` using Matplotlib.

```
>>> from fipy import *
>>> mesh = Grid2D(nx=50, ny=100, dx=0.1, dy=0.01)
>>> x, y = mesh.getCellCenters()
>>> xyVar = CellVariable(mesh=mesh, name="x y", value=x * y)
>>> k = Variable(name="k", value=0.)
>>> viewer = Matplotlib2DGridContourViewer(vars=sin(k * xyVar),
...                 limits={'ymin': 0.1, 'ymax': 0.9},
...                 datamin=-0.9, datamax=2.0,
...                 title="Matplotlib2DGridContourViewer test")
>>> for kval in range(10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()
```

Methods

`__init__`(*self*, *vars*, *title*=`None`, *limits*={}, ***kwlimits*)

Creates a `Matplotlib2DViewer`.

Parameters

vars: a `CellVariable` object.

title: displayed at the top of the `Viewer` window

limits: a (deprecated) alternative to limit keyword arguments *(type=dict)*

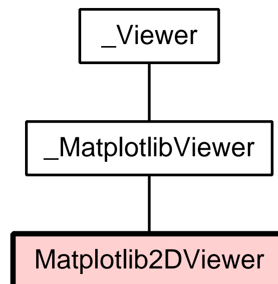
xmin, xmax, ymin, ymax, datamin, datamax: displayed range of data. Any limit set to a (default) value of `None` will autoscale.

Overrides: fipy.viewers.viewer._Viewer.__init__()

Inherited from fipy.viewers.matplotlibViewer.matplotlibViewer._MatplotlibViewer: plot()

Inherited from fipy.viewers.viewer._Viewer: getVars(), plotMesh(), setLimits()

Class Matplotlib2DViewer



Displays a contour plot of a 2D `CellVariable` object.

The `Matplotlib2DViewer` plots a 2D `CellVariable` using Matplotlib.

```
>>> from fipy import *
>>> mesh = (Grid2D(nx=5, ny=10, dx=0.1, dy=0.1)
...         + (Tri2D(nx=5, ny=5, dx=0.1, dy=0.1)
...          + ((0.5,), (0.2,))))
>>> x, y = mesh.getCellCenters()
>>> xyVar = CellVariable(mesh=mesh, name="x y", value=x * y)
>>> k = Variable(name="k", value=0.)
>>> viewer = Matplotlib2DViewer(vars=sin(k * xyVar),
...                 limits={'ymin': 0.1, 'ymax': 0.9},
...                 datamin=-0.9, datamax=2.0,
...                 title="Matplotlib2DViewer test")
>>> for kval in range(10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()
```

Methods

__init__(*self*, *vars*, *title*=None, *limits*={}, *cmap*=None, ***kwlimits*)

Creates a `Matplotlib2DViewer`.

Parameters

vars: a `CellVariable` object.

title: displayed at the top of the `Viewer` window

limits: a (deprecated) alternative to limit keyword arguments *(type=dict)*

cmap: the colormap. Defaults to `pylab.cm.jet`

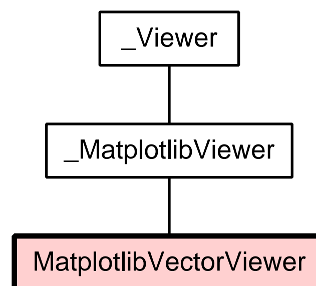
xmin, xmax, ymin, ymax, datamin, datamax: displayed range of data. Any limit set to a (default) value of `None` will autoscale.

Overrides: fipy.viewers.viewer._Viewer.__init__()

Inherited from fipy.viewers.matplotlibViewer.matplotlibViewer._MatplotlibViewer: plot()

Inherited from fipy.viewers.viewer._Viewer: getVars(), plotMesh(), setLimits()

Class MatplotlibVectorViewer



Displays a vector plot of a 2D rank-1 `CellVariable` or `FaceVariable` object using Matplotlib

```
>>> from fipy import *
>>> mesh = Grid2D(nx=50, ny=100, dx=0.1, dy=0.01)
>>> x, y = mesh.getCellCenters()
>>> xyVar = CellVariable(mesh=mesh, name="x y", value=x * y)
>>> k = Variable(name="k", value=0.)
>>> viewer = MatplotlibVectorViewer(vars=sin(k * xyVar).getGrad(),
...                                 title="MatplotlibVectorViewer test")
>>> for kval in range(10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()

>>> viewer = MatplotlibVectorViewer(vars=sin(k * xyVar).getFaceGrad(),
...                                 title="MatplotlibVectorViewer test")
```

```
>>> for kval in range(10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()
```

```
>>> for sparsity in arange(5000, 0, -500):
...     viewer.quiver(sparsity=sparsity)
...     viewer.plot()
>>> viewer._promptForOpinion()
```

```
>>> from fipy import *
>>> mesh = (Grid2D(nx=5, ny=10, dx=0.1, dy=0.1)
...         + (Tri2D(nx=5, ny=5, dx=0.1, dy=0.1)
...          + ((0.5,), (0.2,))))
>>> x, y = mesh.getCellCenters()
>>> xyVar = CellVariable(mesh=mesh, name="x y", value=x * y)
>>> k = Variable(name="k", value=0.)
>>> viewer = MatplotlibVectorViewer(vars=sin(k * xyVar).getGrad(),
...                     title="MatplotlibVectorViewer test")
>>> for kval in range(10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()
```

```
>>> viewer = MatplotlibVectorViewer(vars=sin(k * xyVar).getFaceGrad(),
...                     title="MatplotlibVectorViewer test")
>>> for kval in range(10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()
```

Methods

`__init__`(*self*, *vars*, *title*=None, *scale*=None, *sparsity*=None, *limits*={}, ***kwlimits*)

Creates a `Matplotlib2DViewer`.

Parameters

vars: a rank-1 `CellVariable` or `FaceVariable` object.

title: displayed at the top of the `Viewer` window

scale: if not `None`, scale all arrow lengths by this value

sparsity: if not `None`, then this number of arrows will be randomly chosen (weighted by the cell volume or face area)

limits: a (deprecated) alternative to limit keyword arguments *(type=dict)*

xmin, xmax, ymin, ymax, datamin, datamax: displayed range of data. Any limit set to a (default) value of `None` will autoscale.

Overrides: fipy.viewers.viewer._Viewer.__init__()

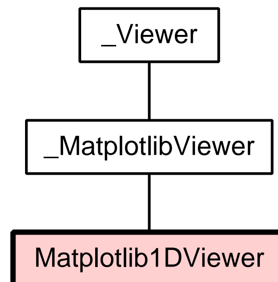
`quiver`(*self*, *sparsity*=`None`, *scale*=`None`)

Inherited from fipy.viewers.matplotlibViewer.matplotlibViewer._MatplotlibViewer: plot()

Inherited from fipy.viewers.viewer._Viewer: getVars(), plotMesh(), setLimits()

10.15 Module fipy.viewers.matplotlibViewer.matplotlib1DViewer

Class Matplotlib1DViewer



Displays a y vs. x plot of one or more 1D `CellVariable` objects using Matplotlib.

```
>>> from fipy import *
>>> mesh = Grid1D(nx=100)
>>> x, = mesh.getCellCenters()
>>> xVar = CellVariable(mesh=mesh, name="x", value=x)
>>> k = Variable(name="k", value=0.)
>>> viewer = Matplotlib1DViewer(vars=(sin(k * xVar), cos(k * xVar / pi)),
...                 limits={'xmin': 10, 'xmax': 90},
...                 datamin=-0.9, datamax=2.0,
...                 title="Matplotlib1DViewer test")
>>> for kval in numerix.arange(0,0.3,0.03):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()
```

Methods

`__init__`(*self*, *vars*, *title*=`None`, *xlog*=`False`, *ylog*=`False`, *limits*={}, ***kwlimits*)

Create a `_MatplotlibViewer`.

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: a (deprecated) alternative to limit keyword arguments *(type=dict)*

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*).

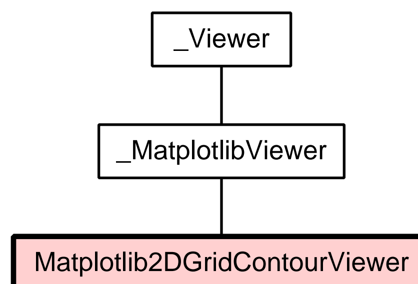
Overrides: fipy.viewers.viewer._Viewer.__init__()

Inherited from fipy.viewers.matplotlibViewer.matplotlibViewer._MatplotlibViewer: plot()

Inherited from fipy.viewers.viewer._Viewer: getVars(), plotMesh(), setLimits()

10.16 Module fipy.viewers.matplotlibViewer.matplotlib2DGridContourViewer

Class Matplotlib2DGridContourViewer



Displays a contour plot of a 2D `CellVariable` object.

The `Matplotlib2DGridContourViewer` plots a 2D `CellVariable` using Matplotlib.

```
>>> from fipy import *
>>> mesh = Grid2D(nx=50, ny=100, dx=0.1, dy=0.01)
>>> x, y = mesh.getCellCenters()
>>> xyVar = CellVariable(mesh=mesh, name="x y", value=x * y)
>>> k = Variable(name="k", value=0.)
```

```
>>> viewer = Matplotlib2DGridContourViewer(vars=sin(k * xyVar),
...                     limits={'ymin': 0.1, 'ymax': 0.9},
...                     datamin=-0.9, datamax=2.0,
...                     title="Matplotlib2DGridContourViewer test")
>>> for kval in range(10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()
```

Methods

__init__(*self*, *vars*, *title*=None, *limits*={}, ***kwlimits*)

Creates a `Matplotlib2DViewer`.

Parameters

vars: a `CellVariable` object.

title: displayed at the top of the `Viewer` window

limits: a (deprecated) alternative to limit keyword arguments *(type=dict)*

xmin, xmax, ymin, ymax, datamin, datamax: displayed range of data. Any limit set to a (default) value of `None` will autoscale.

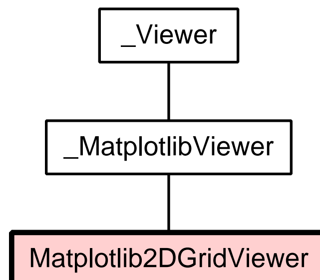
Overrides: fipy.viewers.viewer._Viewer.__init__()

Inherited from fipy.viewers.matplotlibViewer.matplotlibViewer._MatplotlibViewer: plot()

Inherited from fipy.viewers.viewer._Viewer: getVars(), plotMesh(), setLimits()

10.17 Module fipy.viewers.matplotlibViewer.matplotlib2DGridViewer

Class Matplotlib2DGridViewer



Displays an image plot of a 2D `CellVariable` object using Matplotlib.

```
>>> from fipy import *
>>> mesh = Grid2D(nx=50, ny=100, dx=0.1, dy=0.01)
>>> x, y = mesh.getCellCenters()
>>> xyVar = CellVariable(mesh=mesh, name="x y", value=x * y)
>>> k = Variable(name="k", value=0.)
>>> viewer = Matplotlib2DGridViewer(vars=sin(k * xyVar),
...                  limits={'ymin': 0.1, 'ymax': 0.9},
...                  datamin=-0.9, datamax=2.0,
...                  title="Matplotlib2DGridViewer test")
>>> for kval in range(10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()
```

Methods

`__init__`(*self*, *vars*, *title*=`None`, *limits*={}, *cmap*=`None`, ***kwlimits*)

Creates a `Matplotlib2DGridViewer`.

Parameters

vars: A `CellVariable` object.

title: displayed at the top of the `Viewer` window

limits: a (deprecated) alternative to limit keyword arguments *(type=dict)*

cmap: The colormap. Defaults to `pylab.cm.jet`

xmin, xmax, ymin, ymax, datamin, datamax: displayed range of data. Any limit set to a (default) value of `None` will autoscale.

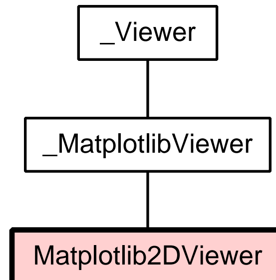
Overrides: fipy.viewers.viewer._Viewer.__init__()

Inherited from fipy.viewers.matplotlibViewer.matplotlibViewer._MatplotlibViewer: plot()

Inherited from fipy.viewers.viewer._Viewer: getVars(), plotMesh(), setLimits()

10.18 Module fipy.viewers.matplotlibViewer.matplotlib2DViewer

Class Matplotlib2DViewer



Displays a contour plot of a 2D `CellVariable` object.

The `Matplotlib2DViewer` plots a 2D `CellVariable` using Matplotlib.

```
>>> from fipy import *
>>> mesh = (Grid2D(nx=5, ny=10, dx=0.1, dy=0.1)
...         + (Tri2D(nx=5, ny=5, dx=0.1, dy=0.1)
...          + ((0.5,), (0.2,))))
>>> x, y = mesh.getCellCenters()
>>> xyVar = CellVariable(mesh=mesh, name="x y", value=x * y)
>>> k = Variable(name="k", value=0.)
>>> viewer = Matplotlib2DViewer(vars=sin(k * xyVar),
...                  limits={'ymin': 0.1, 'ymax': 0.9},
...                  datamin=-0.9, datamax=2.0,
...                  title="Matplotlib2DViewer test")
>>> for kval in range(10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()
```

Methods

`__init__`(*self*, *vars*, *title*=None, *limits*={}, *cmap*=None, ***kwlimits*)

Creates a `Matplotlib2DViewer`.

Parameters

vars: a `CellVariable` object.

title: displayed at the top of the `Viewer` window

limits: a (deprecated) alternative to limit keyword arguments *(type=dict)*

cmap: the colormap. Defaults to `pylab.cm.jet`

xmin, xmax, ymin, ymax, datamin, datamax: displayed range of data. Any limit set to a (default) value of `None` will autoscale.

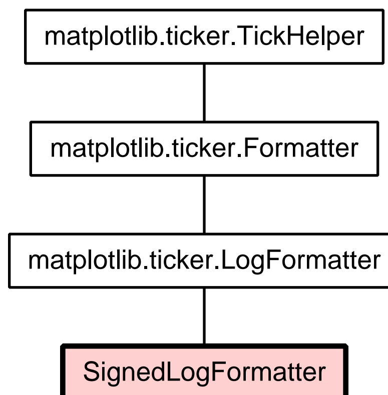
Overrides: fipy.viewers.viewer._Viewer.__init__()

Inherited from fipy.viewers.matplotlibViewer.matplotlibViewer._MatplotlibViewer: plot()

Inherited from fipy.viewers.viewer._Viewer: getVars(), plotMesh(), setLimits()

10.19 Module fipy.viewers.matplotlibViewer.matplotlibSparseMatrixViewer

Class SignedLogFormatter



Format values for log axis;

if attribute decadeOnly is True, only the decades will be labelled.

Methods

`__init__`(*self*, *base*=`10.0`, *labelOnlyBase*=`True`, *threshold*=`0.0`)

base is used to locate the decade tick, which will be the only one to be labeled if labelOnlyBase is False

Overrides: matplotlib.ticker.LogFormatter.__init__()

__call__(*self*, *x*, *pos*=None)

Return the format for tick val x at position pos

Overrides: matplotlib.ticker.Formatter.__call__()

pprint_val(*self*, *x*, *d*)

Overrides: matplotlib.ticker.LogFormatter.pprint_val()

Inherited from matplotlib.ticker.LogFormatter: base(), format_data(), format_data_short(), is_decade(), label_minor(), nearest_long()

Inherited from matplotlib.ticker.Formatter: fix_minus(), get_offset(), set_locs()

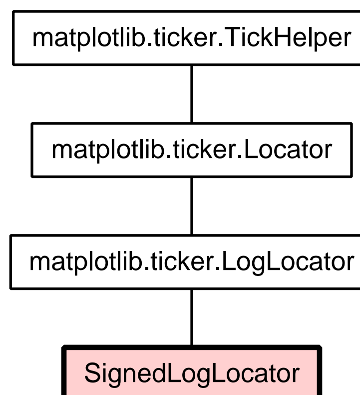
Inherited from matplotlib.ticker.TickHelper: create_dummy_axis(), set_axis(), set_bounds(), set_data_interval(), set_view_interval()

Class Variables

Inherited from matplotlib.ticker.Formatter: locs

Inherited from matplotlib.ticker.TickHelper: axis

Class SignedLogLocator



Determine the tick locations for "log" axes that express both positive and negative values

Methods

`__init__(self, base=10.0, subs=[1.0], threshold=0.0)`

---

place ticks on the location= base**i*subs[j]

Overrides: matplotlib.ticker.LogLocator.__init__()

`__call__(self)`

Return the locations of the ticks

Overrides: matplotlib.ticker.Locator.__call__()

`autoscale(self)`

Try to choose the view limits intelligently

Overrides: matplotlib.ticker.Locator.autoscale()

Inherited from matplotlib.ticker.LogLocator: base(), subs(), view_limits()

Inherited from matplotlib.ticker.Locator: pan(), refresh(), zoom()

Inherited from matplotlib.ticker.TickHelper: create_dummy_axis(), set_axis(), set_bounds(), set_data_interval(), set_view_interval()

Class Variables

Inherited from matplotlib.ticker.TickHelper: axis

Class MatplotlibSparseMatrixViewer

Methods

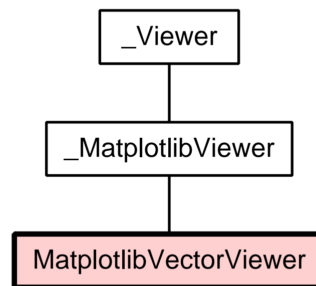
`__init__(self, title='Sparsity')`

---

plot(*self*, *matrix*, *RHSvector*, *log*='auto')

10.20 Module fipy.viewers.matplotlibViewer.matplotlibVectorViewer

Class MatplotlibVectorViewer



Displays a vector plot of a 2D rank-1 `CellVariable` or `FaceVariable` object using Matplotlib

```
>>> from fipy import *
>>> mesh = Grid2D(nx=50, ny=100, dx=0.1, dy=0.01)
>>> x, y = mesh.getCellCenters()
>>> xyVar = CellVariable(mesh=mesh, name="x y", value=x * y)
>>> k = Variable(name="k", value=0.)
>>> viewer = MatplotlibVectorViewer(vars=sin(k * xyVar).getGrad(),
...                    title="MatplotlibVectorViewer test")
>>> for kval in range(10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()

>>> viewer = MatplotlibVectorViewer(vars=sin(k * xyVar).getFaceGrad(),
...                    title="MatplotlibVectorViewer test")
>>> for kval in range(10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()

>>> for sparsity in arange(5000, 0, -500):
...     viewer.quiver(sparsity=sparsity)
...     viewer.plot()
>>> viewer._promptForOpinion()
```

```
>>> from fipy import *
>>> mesh = (Grid2D(nx=5, ny=10, dx=0.1, dy=0.1)
...         + (Tri2D(nx=5, ny=5, dx=0.1, dy=0.1)
...          + ((0.5,), (0.2,))))
>>> x, y = mesh.getCellCenters()
>>> xyVar = CellVariable(mesh=mesh, name="x y", value=x * y)
>>> k = Variable(name="k", value=0.)
>>> viewer = MatplotlibVectorViewer(vars=sin(k * xyVar).getGrad(),
...                  title="MatplotlibVectorViewer test")
>>> for kval in range(10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()

>>> viewer = MatplotlibVectorViewer(vars=sin(k * xyVar).getFaceGrad(),
...                  title="MatplotlibVectorViewer test")
>>> for kval in range(10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()
```

Methods

`__init__`(*self*, *vars*, *title*=`None`, *scale*=`None`, *sparsity*=`None`, *limits*={}, ***kwlimits*)

Creates a `Matplotlib2DViewer`.

Parameters

vars: a rank-1 `CellVariable` or `FaceVariable` object.

title: displayed at the top of the `Viewer` window

scale: if not `None`, scale all arrow lengths by this value

sparsity: if not `None`, then this number of arrows will be randomly chosen (weighted by the cell volume or face area)

limits: a (deprecated) alternative to limit keyword arguments *(type=dict)*

xmin, xmax, ymin, ymax, datamin, datamax: displayed range of data. Any limit set to a (default) value of `None` will autoscale.

Overrides: fipy.viewers.viewer._Viewer.__init__()

`quiver`(*self*, *sparsity*=`None`, *scale*=`None`)

Inherited from fipy.viewers.matplotlibViewer.matplotlibViewer._MatplotlibViewer: plot()

Inherited from fipy.viewers.viewer._Viewer: getVars(), plotMesh(), setLimits()

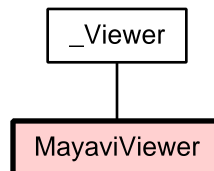
10.21 Module fipy.viewers.matplotlibViewer.matplotlibViewer

10.22 Module fipy.viewers.matplotlibViewer.test

Test numeric implementation of the mesh

10.23 Package fipy.viewers.mayaviViewer

Class MayaviViewer



The `MayaviViewer` creates viewers with the Mayavi python plotting package.

```
>>> from fipy import *
>>> mesh = Grid1D(nx=100)
>>> x, = mesh.getCellCenters()
>>> xVar = CellVariable(mesh=mesh, name="x", value=x)
>>> k = Variable(name="k", value=0.)
>>> viewer = MayaviViewer(vars=(sin(k * xVar), cos(k * xVar / pi)),
...                 limits={'xmin': 10, 'xmax': 90},
...                 datamin=-0.9, datamax=2.0,
...                 title="MayaviViewer 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.getCellCenters()
>>> xyVar = CellVariable(mesh=mesh, name="x y", value=x * y)
```

```
>>> k = Variable(name="k", value=0.)
>>> viewer = MayaviViewer(vars=sin(k * xyVar),
...                 limits={'ymin': 0.1, 'ymax': 0.9},
...                 datamin=-0.9, datamax=2.0,
...                 title="MayaviViewer test")
>>> for kval in range(10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()

>>> from fipy import *
>>> mesh = (Grid2D(nx=5, ny=10, dx=0.1, dy=0.1)
...         + (Tri2D(nx=5, ny=5, dx=0.1, dy=0.1)
...          + ((0.5,), (0.2,))))
>>> x, y = mesh.getCellCenters()
>>> xyVar = CellVariable(mesh=mesh, name="x y", value=x * y)
>>> k = Variable(name="k", value=0.)
>>> viewer = MayaviViewer(vars=sin(k * xyVar),
...                 limits={'ymin': 0.1, 'ymax': 0.9},
...                 datamin=-0.9, datamax=2.0,
...                 title="MayaviViewer 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.getCellCenters()
>>> xyzVar = CellVariable(mesh=mesh, name=r"x y z", value=x * y * z)
>>> k = Variable(name="k", value=0.)
>>> viewer = MayaviViewer(vars=sin(k * xyzVar),
...                     limits={'ymin': 0.1, 'ymax': 0.9},
...                     datamin=-0.9, datamax=2.0,
...                     title="MayaviViewer test")
>>> for kval in range(10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()
```

Issues with the `MayaviViewer` are

- `_getOrderedCellVertexIDs()` doesn't return the correct ordering for 3D meshes. This may be okay for tets and wedges but will break for hexahedrons.
- Different element types can not be displayed for 3D meshes. This is an ordering issue for the cell data. Could get round this either by implementing a method such as `var.getVertexVariable()` and use point data, or reordering the variable data via [tets, wedges, hexs] and keep using cell data. First option is cleaner. Second option is less work.

- Should this class be split into various dimensions? Is it useful to display data with different dimension is same viewer?

Methods

`__init__`(*self*, *vars*, *title*=`None`, *limits*={}, ***kwlimits*)

Create a `MayaviViewer`.

Parameters

vars: a `CellVariable` or tuple of `CellVariable` objects to plot

title: displayed at the top of the `Viewer` window

limits: a (deprecated) alternative to limit keyword arguments *(type=dict)*

xmin, xmax, ymin, ymax, zmin, zmax, datamin, datamax: displayed range of data. Any limit set to a (default) value of `None` will autoscale.

Overrides: fipy.viewers.viewer._Viewer.__init__()

`plot`(*self*, *filename*=`None`)

Update the display of the viewed variables.

Parameters

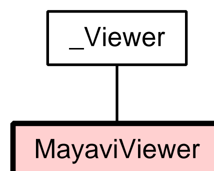
filename: If not `None`, the name of a file to save the image into.

Overrides: fipy.viewers.viewer._Viewer.plot() *(inherited documentation)*

Inherited from fipy.viewers.viewer._Viewer: getVars(), plotMesh(), setLimits()

10.24 Module fipy.viewers.mayaviViewer.mayaviViewer

Class MayaviViewer



The `MayaviViewer` creates viewers with the Mayavi python plotting package.

```
>>> from fipy import *
>>> mesh = Grid1D(nx=100)
>>> x, = mesh.getCellCenters()
>>> xVar = CellVariable(mesh=mesh, name="x", value=x)
>>> k = Variable(name="k", value=0.)
>>> viewer = MayaviViewer(vars=(sin(k * xVar), cos(k * xVar / pi)),
...                 limits={'xmin': 10, 'xmax': 90},
...                 datamin=-0.9, datamax=2.0,
...                 title="MayaviViewer 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.getCellCenters()
>>> xyVar = CellVariable(mesh=mesh, name="x y", value=x * y)
>>> k = Variable(name="k", value=0.)
>>> viewer = MayaviViewer(vars=sin(k * xyVar),
...                 limits={'ymin': 0.1, 'ymax': 0.9},
...                 datamin=-0.9, datamax=2.0,
...                 title="MayaviViewer test")
>>> for kval in range(10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()
```

```
>>> from fipy import *
>>> mesh = (Grid2D(nx=5, ny=10, dx=0.1, dy=0.1)
...         + (Tri2D(nx=5, ny=5, dx=0.1, dy=0.1)
...          + ((0.5,), (0.2,))))
>>> x, y = mesh.getCellCenters()
>>> xyVar = CellVariable(mesh=mesh, name="x y", value=x * y)
>>> k = Variable(name="k", value=0.)
>>> viewer = MayaviViewer(vars=sin(k * xyVar),
...                 limits={'ymin': 0.1, 'ymax': 0.9},
...                 datamin=-0.9, datamax=2.0,
...                 title="MayaviViewer 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.getCellCenters()
```

```
>>> xyzVar = CellVariable(mesh=mesh, name=r"x y z", value=x * y * z)
>>> k = Variable(name="k", value=0.)
>>> viewer = MayaviViewer(vars=sin(k * xyzVar),
...                       limits={'ymin': 0.1, 'ymax': 0.9},
...                       datamin=-0.9, datamax=2.0,
...                       title="MayaviViewer test")
>>> for kval in range(10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()
```

Issues with the `MayaviViewer` are

- `_getOrderedCellVertexIDs()` doesn't return the correct ordering for 3D meshes. This may be okay for tets and wedges but will break for hexahedrons.
- Different element types can not be displayed for 3D meshes. This is an ordering issue for the cell data. Could get round this either by implementing a method such as `var.getVertexVariable()` and use point data, or reordering the variable data via [tets, wedges, hexs] and keep using cell data. First option is cleaner. Second option is less work.
- Should this class be split into various dimensions? Is it useful to display data with different dimension is same viewer?

Methods

`__init__`(*self*, *vars*, *title*=`None`, *limits*={}, ***kwlimits*)

Create a `MayaviViewer`.

Parameters

vars: a `CellVariable` or tuple of `CellVariable` objects to plot

title: displayed at the top of the `Viewer` window

limits: a (deprecated) alternative to limit keyword arguments *(type=dict)*

xmin, xmax, ymin, ymax, zmin, zmax, datamin, datamax: displayed range of data. Any limit set to a (default) value of `None` will autoscale.

Overrides: fipy.viewers.viewer._Viewer.__init__()

`plot`(*self*, *filename*=`None`)

Update the display of the viewed variables.

Parameters

filename: If not `None`, the name of a file to save the image into.

Overrides: fipy.viewers.viewer._Viewer.plot() *(inherited documentation)*

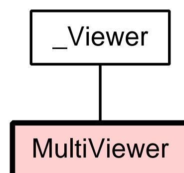
Inherited from fipy.viewers.viewer._Viewer: getVars(), plotMesh(), setLimits()

10.25 Module fipy.viewers.mayaviViewer.test

Test numeric implementation of the mesh

10.26 Module fipy.viewers.multiViewer

Class MultiViewer



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.

Methods

`__init__`(*self*, *viewers*)

```
:Parameters: viewers : list the viewers to bind together
```

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.

Overrides: fipy.viewers.viewer._Viewer.__init__()

setLimits(*self*, *limits*={}, ***kwlimits*)

Update the limits.

Parameters

limits: 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.

Overrides: fipy.viewers.viewer._Viewer.setLimits() *(inherited documentation)*

plot(*self*)

Update the display of the viewed variables.

Parameters

filename: If not `None`, the name of a file to save the image into.

Overrides: fipy.viewers.viewer._Viewer.plot() *(inherited documentation)*

getViewers(*self*)

Inherited from fipy.viewers.viewer._Viewer: getVars(), plotMesh()

10.27 Module fipy.viewers.test

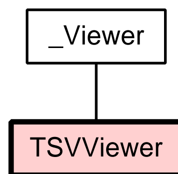
Test implementation of the viewers

10.28 Module fipy.viewers.testinteractive

Interactively test the viewers

10.29 Module fipy.viewers.tsvViewer

Class TSVViewer



"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   ...
:
:
```

Methods

`__init__`(*self*, *vars*, *title*=`None`, *limits*={}, ***kwlimits*)

Creates a `TSVViewer`.

Any cell centers that lie outside the limits provided will not be included. Any values that lie outside the *datamin* or *datamax* will be replaced with `nan`.

All variables must have the same mesh.

It tries to do something reasonable with rank-1 `CellVariable` and `FaceVariable` objects.

Parameters

vars: 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: a (deprecated) alternative to limit keyword arguments *(type=dict)*

xmin, xmax, ymin, ymax, zmin, zmax, datamin, datamax: displayed range of data. Any limit set to a (default) value of `None` will autoscale.

Overrides: fipy.viewers.viewer._Viewer.__init__()

plot(*self*, *filename*=None)

"plot" the coordinates and values of the variables to `filename`. If `filename` is not provided, "plots" to stdout.

```
>>> from fipy.meshes.grid1D import Grid1D
>>> m = Grid1D(nx = 3, dx = 0.4)
>>> from fipy.variables.cellVariable import CellVariable
>>> v = CellVariable(mesh = m, name = "var", value = (0, 2, 5))
>>> TSVViewer(vars = (v, v.getGrad())).plot() #doctest: +NORMALIZE_WHITESPACE
x var var_gauss_grad_x
0.2 0 2.5
0.6 2 6.25
1 5 3.75
```

```
>>> from fipy.meshes.grid2D import Grid2D
>>> m = Grid2D(nx = 2, dx = .1, ny = 2, dy = 0.3)
>>> v = CellVariable(mesh = m, name = "var", value = (0, 2, -2, 5))
>>> TSVViewer(vars = (v, v.getGrad())).plot() #doctest: +NORMALIZE_WHITESPACE
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.

Overrides: fipy.viewers.viewer._Viewer.plot()

Inherited from fipy.viewers.viewer._Viewer: getVars(), plotMesh(), setLimits()

10.30 Module fipy.viewers.viewer

Functions

`make`(*vars*, *title*=`None`, *limits*=`None`)

Bibliography

[1] Guido van Rossum, *Python Reference Manual*. URL http://docs.python.org/ref/. 14

[2] Daniel Wheeler, Jonathan E. Guyer, and James A. Warren, *A Finite Volume PDE Solver Using Python*. URL http://www.ctcms.nist.gov/fipy/download/fipy.pdf. 166, 173, 175, 179, 189

[3] Konrad Hinsen. URL http://starship.python.net/~hinsen/ScientificPython/ScientificPythonManual/Scientific_31.html. 198

Index

_TestProgram, 186
addOverFacesVariable, 249
AdsorbingSurfactantEquation, 109–114
adsorbingSurfactantEquation, 109–114
advectionEquation, 92
advectionTerm, 93
advection, 91
allclose(), 239
allequal(), 239
arccosh(), 232
arccos(), 231
arcsinh(), 233
arcsin(), 232
arctan2(), 233
arctanh(), 234
arctan(), 233
arithmeticCellToFaceVariable, 249
BetaNoiseVariable, 249–252
betaNoiseVariable, 249–252
binaryOperatorVariable, 252
boundaryConditions, 8–9
BoundaryCondition, 9–10
    __init__(), 9
    __repr__(), 10
boundaryCondition, 9–10
buildAdvectionEquation(), 92
buildHigherOrderAdvectionEquation(), 94
buildMetalIonDiffusionEquation(), 107
buildSurfactantBulkDiffusionEquation(), 119
ceil(), 237
CellTerm, 159–160
cellTerm, 159–160
cellToFaceVariable, 252
CellVariable, 252–260
    getArithmeticFaceValue(), 256, 257
    getCellVolumeAverage(), 255
    getFaceGradAverage(), 259
    getFaceGrad(), 258
    getGaussGrad(), 256
    getGrad(), 255
    getHarmonicFaceValue(), 258
    getLeastSquaresGrad(), 256
    getMinmodFaceValue(), 257
    getOld(), 259
    updateOld(), 259
cellVariable, 252–260
cellVolumeAverageVariable, 260–261
Cell, 27–28, 81–82
    __cmp__(), 27
    __init__(), 27, 81
    __repr__(), 27, 82
    getBoundingCells(), 82
    getCenter(), 27, 81
    getFaceIDs(), 82
    getFaceOrientations(), 81
    getFaces(), 82
    getID(), 27, 81
    getMesh(), 27
    getNormal(), 27
    getVolume(), 81
cell, 27–28, 80–82
centralDiffConvectionTerm, 160
CentralDifferenceConvectionTerm, 160
collectedDiffusionTerm, 160–161
colorbar, 312
common, 13
conjugate(), 238
constant, 261
convectionCoeff, 114
ConvectionTerm, 161–163
convectionTerm, 161–163
cosh(), 234
cos(), 234
CylindricalGrid1D, 28–29
CylindricalGrid1D(), 21
cylindricalGrid1D, 21, 28–29
CylindricalGrid2D, 29–31
CylindricalGrid2D(), 22

cylindricalGrid2D, 22, 29–31
CylindricalUniformGrid1D, 31–33
cylindricalUniformGrid1D, 31–33
CylindricalUniformGrid2D, 33–34
cylindricalUniformGrid2D, 33–34
DictWithDefault, 191
DiffusionTermNoCorrection, 164–165
DiffusionTerm, 163–164
diffusionTerm, 163–165
dimensions, 191
distanceFunction, 95
DistanceVariable, 95–100
    calcDistanceFunction(), 98
    extendVariable(), 98
    getCellInterfaceAreas(), 98
distanceVariable, 95–100
doctestPlus, 185
DomDecompPreconditioner, 145
domDecompPreconditioner, 145
dot(), 238
dump, 224
electroChem, 100
equation, 165
error(), 153
execButNoTest(), 185
ExplicitDiffusionTerm, 165–166
explicitDiffusionTerm, 165–166
ExplicitNthOrderDiffusionTerm, 171–172
explicitSourceTerm, 166
ExplicitUpwindConvectionTerm, 166–167
explicitUpwindConvectionTerm, 166–167
ExponentialConvectionTerm, 167
exponentialConvectionTerm, 167
ExponentialNoiseVariable, 261–263
exponentialNoiseVariable, 261–263
exportAsMesh(), 36
exp(), 237
Face2D, 84
face2D, 84
faceGradContributionsVariable, 263–264
faceGradVariable, 264
FaceTerm, 168
faceTerm, 167–168
FaceVariable, 264–265
    getDivergence(), 265
faceVariable, 264–265
Face, 34–35, 82–84
    __init__(), 35, 83
    __repr__(), 84
    addBoundingCell(), 83
    getArea(), 35, 83
    getCellDistance(), 84
    getCellID(), 35, 83
    getCells(), 83
    getCenter(), 35, 83
    getID(), 35, 83
    getMesh(), 35
    getNormal(), 84
face, 34–35, 82–84
fixedBCFaceGradVariable, 265–266
FixedFlux, 10
fixedFlux, 10
FixedValue, 11
fixedValue, 10–11
floor(), 237
GammaNoiseVariable, 266–269
gammaNoiseVariable, 266–269
GapFillMesh, 100–103
    buildTransitionMesh(), 102
    getCellIDsAboveFineRegion(), 102
    getFineMesh(), 103
gapFillMesh, 100–106
gaussCellGradVariable, 269
GaussianNoiseVariable, 269–272
gaussianNoiseVariable, 269–272
getShape(), 229
getUnit(), 228
Gist1DViewer, 305–306, 312–313
gist1DViewer, 312–313
Gist2DViewer, 306–309, 313–315
gist2DViewer, 313–315
GistVectorViewer, 309–311, 316–317
    getArray(), 310, 317
gistVectorViewer, 315–317
GistViewer(), 305
gistViewer, 317–318
gistViewer, 305–311
gmshExport, 23, 36
GmshImporter2DIn3DSpace, 42–43
GmshImporter2D, 40–42
GmshImporter3D, 43–44
gmshImport, 23, 37–44
Gnuplot1DViewer, 319–320, 323
gnuplot1DViewer, 323
Gnuplot2DViewer, 320–322, 324–325
gnuplot2DViewer, 323–325
GnuplotViewer(), 319
gnuplotViewer, 325

gnuplotViewer, 319–322
Grid1D, 44–46
  getPhysicalShape(), 45
  getScale(), 45
  getShape(), 45
Grid1D(), 24
grid1D, 24, 44–46
Grid2D, 46–48, 84–88
  getPhysicalShape(), 47
  getScale(), 47
  getShape(), 47, 87
Grid2D(), 25
grid2D, 25, 46–48, 84–88
Grid3D, 48–50
  getPhysicalShape(), 49
  getScale(), 49
  getShape(), 49
Grid3D(), 26
grid3D, 26, 48–50
harmonicCellToFaceVariable, 272–273
higherOrderAdvectionEquation, 94
higherOrderAdvectionTerm, 95
HistogramVariable, 273–274
histogramVariable, 273–274
HybridConvectionTerm, 169
hybridConvectionTerm, 168–169
ICPreconditioner, 145
icPreconditioner, 145
IllConditionedPreconditionerWarning, 133–134
implicitDiffusionTerm, 169–170
ImplicitSourceTerm, 170
implicitSourceTerm, 170
indices(), 240
inline, 225
isclose(), 239
isFloat(), 230
isInt(), 230
JacobiPreconditioner, 146
jacobiPreconditioner, 145–146
L1error(), 154
L1norm(), 240
L2error(), 154
L2norm(), 240
lateImportTest, 186
leastSquaresCellGradVariable, 274–275
levelSetDiffusionEquation, 100
levelSetDiffusionVariable, 100
levelSet, 91
LinearBicgstabSolver, 140
linearBicgstabSolver, 140
LinearCGSSolver, 125–126, 141
linearCGSSolver, 125–126, 140–141
LinearGMRESSolver, 126, 142
linearGMRESSolver, 126, 141–142
LinearJORSolver, 127
linearJORSolver, 126–127
LinearLUSolver, 128, 143
linearLUSolver, 127–128, 142–143
LinearPCGSolver, 129, 144
linearPCGSolver, 128–129, 143–144
lines, 114
LINFerror(), 154
LINFnorm(), 240
log10(), 235
log(), 238
make(), 303, 353
Matplotlib1DViewer, 326–327, 334–335
matplotlib1DViewer, 334–335
Matplotlib2DGridContourViewer, 328–330, 335–336
matplotlib2DGridContourViewer, 335–336
Matplotlib2DGridViewer, 327–328, 336–337
matplotlib2DGridViewer, 336–337
Matplotlib2DViewer, 330–331, 338–339
matplotlib2DViewer, 337–339
MatplotlibSparseMatrixViewer, 341–342
  __init__(), 341
  plot(), 342
matplotlibSparseMatrixViewer, 339–342
MatplotlibSurfactantViewer, 114–116
matplotlibSurfactantViewer, 114–116
MatplotlibVectorViewer, 331–333, 342–344
  quiver(), 333, 343
matplotlibVectorViewer, 342–344
MatplotlibViewer(), 326
matplotlibViewer, 344
matplotlibViewer, 326–333
MatrixIllConditionedWarning, 135–136
MaximumIterationWarning, 131–132
MayaviSurfactantViewer, 116–118
mayaviSurfactantViewer, 116–118
MayaviViewer, 344–349
mayaviViewer, 346–349
mayaviViewer, 344–346
MemoryHighWaterThread, 226
  stop(), 226
memoryLeak, 226
MemoryLogger, 226–227

- __del__(), 227
- __init__(), 227
- start(), 227
- stop(), 227

memoryLogger, 226–227
memoryUsage, 227
Mesh1D, 57–59
mesh1D, 57–59
Mesh2D, 59–61
- extrude(), 60

mesh2D, 59–61
MeshAdditionError, 50
MeshDimensionError, 304
meshes, 12–13
MeshExportError, 36
MeshImportError, 39–40
meshVariable, 275
Mesh, 13–20, 50–57, 88–89
- __add__(), 13, 53
- __getstate__(), 57
- __init__(), 13
- __mul__(), 15, 55
- __repr__(), 16
- __setstate__(), 57
- getCellCenters(), 19
- getCells(), 16
- getCellVolumes(), 19
- getDim(), 16
- getExteriorFaces(), 16
- getFaceCellIDs(), 56
- getFaceCenters(), 57
- getFaceOrientations(), 88
- getFacesBack(), 19
- getFacesBottom(), 18
- getFacesFront(), 19
- getFacesLeft(), 17
- getFacesRight(), 17
- getFacesTop(), 18, 19
- getFaces(), 17
- getInteriorFaces(), 16
- getNearestCell(), 20
- getNumberOfCells(), 16
- getPhysicalShape(), 89
- getScale(), 89
- getVertexCoords(), 56
- setScale(), 20

mesh, 13–20, 50–57, 88–89
metalIonDiffusionEquation, 107–108
metalIonSourceVariable, 109
minmodCellToFaceVariable, 275
modCellGradVariable, 275
modCellToFaceVariable, 275
models, 90–91
modFaceGradVariable, 275
modPhysicalField, 275
ModularVariable, 275–278
- getFaceGradNoMod(), 277

modularVariable, 275–278
MshFile, 40
- __init__(), 40
- getFilename(), 40
- remove(), 40

MultilevelDDPreconditioner, 146
multilevelDDPreconditioner, 146
MultilevelSGSPreconditioner, 147
multilevelSGSPreconditioner, 146–147
MultilevelSolverSmootherPreconditioner, 147–148
multilevelSolverSmootherPreconditioner, 147–148
MultiViewer, 349–350
- getViewers(), 350

multiViewer, 349–350
NoiseVariable, 278–280
- scramble(), 279

noiseVariable, 278–280
NthOrderBoundaryCondition, 11–12
nthOrderBoundaryCondition, 11–12
NthOrderDiffusionTerm, 171
nthOrderDiffusionTerm, 170–172
NumberDict, 191
numerix, 228–244
numMesh, 27
obj2sctype(), 240
ones(), 228
operatorVariable, 280
parser, 245
parse(), 245
PeriodicGrid1D, 61–63
periodicGrid1D, 61–63, 80
PeriodicGrid2DLeftRight, 65–67
PeriodicGrid2DTopBottom, 67–68
PeriodicGrid2D, 63–65
periodicGrid2D, 63–68, 80
PhysicalField, 194–216
- __abs__(), 201
- __add__(), 197, 198
- __array__(), 203
- __array_wrap__(), 203
- __div__(), 199, 200

__eq__(), 205
__float__(), 203
__ge__(), 205
__getitem__(), 202
__gt__(), 204
__le__(), 205
__len__(), 205
__lt__(), 205
__mod__(), 200
__mul__(), 198, 199
__ne__(), 205
__neg__(), 201
__nonzero__(), 201
__pos__(), 201
__pow__(), 200
__rdiv__(), 200
__rpow__(), 201
__rsub__(), 198
__setitem__(), 202
__sub__(), 198
allclose(), 215
allequal(), 215
arccosh(), 208
arccos(), 208
arcsin(), 209
arctan2(), 211
arctanh(), 212
arctan(), 212
ceil(), 213
conjugate(), 213
convertToUnit(), 205
copy(), 196
cosh(), 210
cos(), 210
dot(), 213
floor(), 213
getNumericValue(), 207
getsctype(), 207
getShape(), 214
getUnit(), 207
inBaseUnits(), 208
inDimensionless(), 206
inRadians(), 206
inSIUnits(), 208
inUnitsOf(), 206
isCompatible(), 208
itemset(), 202
log10(), 213
log(), 212
put(), 214
reshape(), 215
setUnit(), 207
sign(), 201
sinh(), 210
sin(), 209
sqrt(), 209
sum(), 215
take(), 214
tanh(), 211
tan(), 211
tostring(), 197

physicalField, 192–223

PhysicalUnit, 216–223
- __cmp__(), 217
- __div__(), 218
- __init__(), 216
- __mul__(), 217, 218
- __pow__(), 219
- __rdiv__(), 219
- __repr__(), 216, 217
- conversionFactorTo(), 220
- conversionTupleTo(), 221
- isAngle(), 221
- isCompatible(), 221
- isDimensionlessOrAngle(), 222
- isDimensionless(), 221
- isInverseAngle(), 222
- name(), 222
- setName(), 222

PIDStepper, 156
pidStepper, 156
PowerLawConvectionTerm, 173
powerLawConvectionTerm, 172–173
PreconditionerNotPositiveDefiniteWarning, 134–135
preconditioners, 144–145
PreconditionerWarning, 132–133
Preconditioner, 148
- __init__(), 148

preconditioner, 148
prune(), 247
PseudoRKQSStepper, 156–157
pseudoRKQSStepper, 156–157
putAdd(), 247
put(), 229
pyMesh, 80
pysparseMatrix, 246
PysparseSolver, 129–130

pysparseSolver, 129–130
pysparse, 125
rank(), 230
read(), 224
reshape(), 229
residual(), 153
ScalarQuantityOutOfRangeWarning, 137–138
ScharfetterGummelFaceVariable, 280–282
scharfetterGummelFaceVariable, 280–282
SignedLogFormatter, 339–340
SignedLogLocator, 340–341
sign(), 237
sinh(), 236
sin(), 236
SkewedGrid2D, 68–69
  getPhysicalShape(), 69
  getScale(), 68
  getShape(), 69
skewedGrid2D, 68–69, 90
SolverConvergenceWarning, 130–131
solvers, 124–125
Solver, 138–139
  __init__(), 139
  __repr__(), 139
solver, 130–139
SourceTerm, 174–175
sourceTerm, 173–175
sparseMatrix, 246
sqrtDot(), 239
sqrt(), 236
StagnatedSolverWarning, 136–137
steppers, 153–155
Stepper, 157–158
  __init__(), 157
  failFn *(static method)*, 158
  step(), 158
  successFn *(static method)*, 157
  sweepFn *(static method)*, 157
stepper, 157–158
sum(), 230
surfactantBulkDiffusionEquation, 119
SurfactantEquation, 120–121
  __init__(), 120
  solve(), 120
  sweep(), 121
surfactantEquation, 120–121
SurfactantVariable, 121–124
  getInterfaceVar(), 123
surfactantVariable, 121–124
surfactant, 109
sweepMonotonic(), 155
take(), 240
tanh(), 235
tan(), 235
terms, 158–159
Term, 175–180
  __add__(), 178
  __eq__(), 180
  __init__(), 175
  __neg__(), 179
  __pos__(), 179
  __radd__(), 179
  __repr__(), 180
  __rsub__(), 179
  __sub__(), 179
  cacheMatrix(), 178
  cacheRHSvector(), 178
  copy(), 175
  getMatrix(), 178
  getRHSvector(), 178
  justResidualVector(), 176
  residualVectorAndNorm(), 177
  solve(), 176
  sweep(), 176
term, 175–180
testBase, 186
testinteractive, 350
testProgram, 186
tests, 184
test, 12, 69, 89–90, 109, 124, 139–140, 180–181, 184, 246, 282, 318, 325, 344, 349–350
tools, 187–190
tostring(), 230
TransientTerm, 181–182
transientTerm, 181–182
TrenchMesh, 103–106
  getBottomFaces(), 105
  getElectrolyteMask(), 105
  getTopFaces(), 105
Tri2D, 70–71
  getPhysicalShape(), 71
  getScale(), 70
  getShape(), 71
tri2D, 69–71, 90
TrilinosAztecOOSolver, 149–150
trilinosAztecOOSolver, 148–150
trilinosMatrix, 246
TrilinosMLTest, 150–151

trilinosMLTest, 150–151
TrilinosSolver, 151–152
trilinosSolver, 151–152
trilinos, 140
TSVViewer, 351–352
tsvViewer, 350–352
unaryOperatorVariable, 282
UniformGrid1D, 71–74
uniformGrid1D, 71–74
UniformGrid2D, 74–77
uniformGrid2D, 74–77
UniformGrid3D, 78–80
uniformGrid3D, 77–80
UniformNoiseVariable, 282–284
uniformNoiseVariable, 282–284
UpwindConvectionTerm, 183
upwindConvectionTerm, 182–183
VanLeerConvectionTerm, 184
vanLeerConvectionTerm, 183–184
variables, 248–249
Variable, 285–301
    __abs__(), 292
    __add__(), 291
    __and__(), 295
    __array__(), 286
    __array_wrap__(), 286
    __call__(), 289
    __div__(), 292
    __eq__(), 293
    __float__(), 296
    __ge__(), 294
    __getitem__(), 300
    __getstate__(), 301
    __gt__(), 294
    __int__(), 296
    __iter__(), 296
    __le__(), 293
    __len__(), 296
    __lt__(), 293
    __mod__(), 292
    __mul__(), 291
    __ne__(), 294
    __neg__(), 292
    __nonzero__(), 296
    __or__(), 295
    __pos__(), 292
    __pow__(), 292
    __rdiv__(), 292
    __rpow__(), 292
    __rsub__(), 291
    __setitem__(), 288
    __setstate__(), 301
    __sub__(), 291
    allclose(), 300
    allequal(), 301
    all(), 296
    any(), 296
    arccosh(), 297
    arccos(), 297
    arcsinh(), 297
    arcsin(), 297
    arctan2(), 299
    arctanh(), 298
    arctan(), 298
    cacheMe(), 289
    ceil(), 299
    conjugate(), 299
    copy(), 286
    cosh(), 299
    cos(), 298
    dontCacheMe(), 289
    dot(), 299
    exp(), 298
    floor(), 299
    getMag(), 301
    getName(), 288
    getNumericValue(), 290
    getsctype(), 291
    getShape(), 290
    getSubscribedVariables(), 291
    getUnit(), 287
    getValue(), 289
    inBaseUnits(), 287
    inUnitsOf(), 287
    itemset(), 288
    log10(), 298
    log(), 298
    max(), 300
    min(), 300
    put(), 289
    reshape(), 299
    setName(), 288
    setUnit(), 287
    setValue(), 289
    sign(), 299
    sinh(), 298
    sin(), 298
    sqrt(), 297

  - sum(), 300
  - take(), 300
  - tanh(), 297
  - tan(), 297
  - tostring(), 288
  - transpose(), 299
- variable, 284–301
- vector, 247
- Vertex, 89–90
  - __init__(), 90
  - __repr__(), 90
  - getCoordinates(), 90
- vertex, 89–90
- viewers, 302–304
- Viewer(), 303
- viewer, 353
- write(), 224
- zeros(), 228