

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*

February 12, 2007

Version 1.2

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.

Contents

1	How To Read This Manual	5
1.1	Module <code>fiPy.__version__</code>	8
2	Package <code>fiPy.boundaryConditions</code>	9
2.1	Module <code>fiPy.boundaryConditions.boundaryCondition</code>	9
2.2	Module <code>fiPy.boundaryConditions.fixedFlux</code>	11
2.3	Module <code>fiPy.boundaryConditions.fixedValue</code>	12
2.4	Module <code>fiPy.boundaryConditions.nthOrderBoundaryCondition</code>	13
3	Package <code>fiPy.meshes</code>	15
3.1	Module <code>fiPy.meshes.common.mesh</code>	15
3.2	Module <code>fiPy.meshes.grid1D</code>	20
3.3	Module <code>fiPy.meshes.grid2D</code>	21
3.4	Module <code>fiPy.meshes.grid3D</code>	22
3.5	Module <code>fiPy.meshes.meshIterator</code>	23
3.6	Module <code>fiPy.meshes.numMesh.cell</code>	25
3.7	Module <code>fiPy.meshes.numMesh.face</code>	26
3.8	Module <code>fiPy.meshes.numMesh.gmshExport</code>	27
3.9	Module <code>fiPy.meshes.numMesh.gmshImport</code>	28
3.10	Module <code>fiPy.meshes.numMesh.grid1D</code>	34
3.11	Module <code>fiPy.meshes.numMesh.grid2D</code>	36
3.12	Module <code>fiPy.meshes.numMesh.grid3D</code>	38
3.13	Module <code>fiPy.meshes.numMesh.mesh</code>	40
3.14	Module <code>fiPy.meshes.numMesh.mesh1D</code>	44
3.15	Module <code>fiPy.meshes.numMesh.mesh2D</code>	46
3.16	Module <code>fiPy.meshes.numMesh.periodicGrid1D</code>	48
3.17	Module <code>fiPy.meshes.numMesh.periodicGrid2D</code>	50
3.18	Module <code>fiPy.meshes.numMesh.skewedGrid2D</code>	54
3.19	Module <code>fiPy.meshes.numMesh.tri2D</code>	56
3.20	Module <code>fiPy.meshes.numMesh.uniformGrid1D</code>	58
3.21	Module <code>fiPy.meshes.numMesh.uniformGrid2D</code>	60
3.22	Module <code>fiPy.meshes.numMesh.uniformGrid3D</code>	62
3.23	Module <code>fiPy.meshes.pyMesh.cell</code>	64
3.24	Module <code>fiPy.meshes.pyMesh.face</code>	65
3.25	Module <code>fiPy.meshes.pyMesh.face2D</code>	67
3.26	Module <code>fiPy.meshes.pyMesh.grid2D</code>	68
3.27	Module <code>fiPy.meshes.pyMesh.mesh</code>	71
3.28	Module <code>fiPy.meshes.pyMesh.vertex</code>	72

4	Package <code>fiPy.models</code>	73
4.1	Module <code>fiPy.models.levelSet.advection.advectionEquation</code>	73
4.2	Module <code>fiPy.models.levelSet.advection.higherOrderAdvectionEquation</code>	74
4.3	Module <code>fiPy.models.levelSet.distanceFunction.distanceVariable</code>	75
4.4	Module <code>fiPy.models.levelSet.electroChem.gapFillMesh</code>	80
4.5	Module <code>fiPy.models.levelSet.electroChem.metalIonDiffusionEquation</code>	84
4.6	Module <code>fiPy.models.levelSet.surfactant.adsorbingSurfactantEquation</code>	86
4.7	Module <code>fiPy.models.levelSet.surfactant.surfactantBulkDiffusionEquation</code>	91
4.8	Module <code>fiPy.models.levelSet.surfactant.surfactantEquation</code>	92
4.9	Module <code>fiPy.models.levelSet.surfactant.surfactantVariable</code>	94
5	Package <code>fiPy.solvers</code>	97
5.1	Module <code>fiPy.solvers.linearCGSSolver</code>	97
5.2	Module <code>fiPy.solvers.linearGMRESSolver</code>	98
5.3	Module <code>fiPy.solvers.linearJORSolver</code>	99
5.4	Module <code>fiPy.solvers.linearLUSolver</code>	100
5.5	Module <code>fiPy.solvers.linearPCGSolver</code>	101
5.6	Module <code>fiPy.solvers.linearScipyCGSolver</code>	102
5.7	Module <code>fiPy.solvers.linearScipyGMRESSolver</code>	103
5.8	Module <code>fiPy.solvers.linearScipyLUSolver</code>	104
5.9	Module <code>fiPy.solvers.solver</code>	105
6	Package <code>fiPy.steppers</code>	111
6.1	Functions	111
6.2	Module <code>fiPy.steppers.pidStepper</code>	114
6.3	Module <code>fiPy.steppers.pseudoRKQSStepper</code>	115
6.4	Module <code>fiPy.steppers.stepper</code>	116
7	Package <code>fiPy.terms</code>	117
7.1	Module <code>fiPy.terms.cellTerm</code>	117
7.2	Module <code>fiPy.terms.centralDiffConvectionTerm</code>	118
7.3	Module <code>fiPy.terms.convectionTerm</code>	119
7.4	Module <code>fiPy.terms.diffusionTerm</code>	121
7.5	Module <code>fiPy.terms.explicitDiffusionTerm</code>	122
7.6	Module <code>fiPy.terms.explicitUpwindConvectionTerm</code>	123
7.7	Module <code>fiPy.terms.exponentialConvectionTerm</code>	124
7.8	Module <code>fiPy.terms.faceTerm</code>	125
7.9	Module <code>fiPy.terms.hybridConvectionTerm</code>	126
7.10	Module <code>fiPy.terms.implicitDiffusionTerm</code>	127
7.11	Module <code>fiPy.terms.implicitSourceTerm</code>	128
7.12	Module <code>fiPy.terms.nthOrderDiffusionTerm</code>	129
7.13	Module <code>fiPy.terms.powerLawConvectionTerm</code>	131
7.14	Module <code>fiPy.terms.sourceTerm</code>	132
7.15	Module <code>fiPy.terms.term</code>	133
7.16	Module <code>fiPy.terms.transientTerm</code>	137
7.17	Module <code>fiPy.terms.upwindConvectionTerm</code>	139
7.18	Module <code>fiPy.terms.vanLeerConvectionTerm</code>	140

8	Package fipy.tools	141
8.1	Module fipy.tools.dimensions.physicalField	141
8.2	Module fipy.tools.dump	163
8.3	Module fipy.tools.memoryLogger	164
8.4	Module fipy.tools.numerix	166
8.5	Module fipy.tools.parser	178
8.6	Module fipy.tools.vector	179
9	Package fipy.variables	181
9.1	Module fipy.variables.betaNoiseVariable	181
9.2	Module fipy.variables.cellVariable	184
9.3	Module fipy.variables.exponentialNoiseVariable	190
9.4	Module fipy.variables.faceVariable	193
9.5	Module fipy.variables.gammaNoiseVariable	194
9.6	Module fipy.variables.gaussianNoiseVariable	197
9.7	Module fipy.variables.histogramVariable	200
9.8	Module fipy.variables.modularVariable	203
9.9	Module fipy.variables.noiseVariable	206
9.10	Module fipy.variables.uniformNoiseVariable	208
9.11	Module fipy.variables.variable	211
9.12	Module fipy.variables.vectorCellVariable	223
9.13	Module fipy.variables.vectorFaceVariable	226
10	Package fipy.viewers	229
10.1	Functions	229
10.2	Class MeshDimensionError	230
10.3	Module fipy.viewers.multiViewer	231
10.4	Module fipy.viewers.tsvViewer	232
10.5	Module fipy.viewers.viewer	234
	Bibliography	235
	Index	236

Chapter 1

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: `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`

```
fipy.package.base.Base └─
                        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 `Base: method1`

1.1 Module `fipy.__version__`

Variables

Name	Description
<code>__version__</code>	Value: <code>'1.2'</code> (<i>type=</i> <code>str</code>)

Chapter 2

Package `fipy.boundaryConditions`

2.1 Module `fipy.boundaryConditions.boundaryCondition`

Class `BoundaryCondition`

Known Subclasses: `FixedFlux`, `FixedValue`, `NthOrderBoundaryCondition`

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 not be invoked directly.

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.getFaces(), 0)
Traceback (most recent call last):
...
IndexError: Face list has interior faces
>>> bc = BoundaryCondition(mesh.getFaces()[0] + mesh.getFaces()[0], 0)
Traceback (most recent call last):
...
IndexError: Face list has repeated entries
```

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`

`fipy.boundaryConditions.boundaryCondition.BoundaryCondition`  `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 `BoundaryCondition`: `__repr__`

2.3 Module `fipy.boundaryConditions.fixedValue`

Class `FixedValue`

`fipy.boundaryConditions.boundaryCondition.BoundaryCondition`  `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 $-\text{value} * G_{\text{face}}$ 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 `BoundaryCondition`: `__init__`, `__repr__`

2.4 Module `fipy.boundaryConditions.nthOrderBoundaryCondition`

Class `NthOrderBoundaryCondition`

`fipy.boundaryConditions.boundaryCondition.BoundaryCondition`

`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 `BoundaryCondition`: `__repr__`

Chapter 3

Package `fipy.meshes`

3.1 Module `fipy.meshes.common.mesh`

Class `Mesh`

Known Subclasses: [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, 0.5,]
 [ 1.5, 0.5,]
 [ 0.5, 1.5,]
 [ 1.5, 1.5,]] 1
```

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

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

```
[ 5.5, 11.5,]
[ 6.5, 11.5,]]
```

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

```
>>> addedMesh = baseMesh + (baseMesh + (2, 0))
>>> addedMesh.getCellCenters()
[[ 0.5, 0.5,]
 [ 1.5, 0.5,]
 [ 0.5, 1.5,]
 [ 1.5, 1.5,]
 [ 2.5, 0.5,]
 [ 3.5, 0.5,]
 [ 2.5, 1.5,]
 [ 3.5, 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))
>>> addedMesh.getCellCenters()
[[ 0.5, 0.5,]
 [ 1.5, 0.5,]
 [ 0.5, 1.5,]
 [ 1.5, 1.5,]
 [ 1.5, 0.5,]
 [ 2.5, 0.5,]
 [ 1.5, 1.5,]
 [ 2.5, 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
>>> triAddedMesh.getCellCenters()
[[ 0.5      , 0.5      ,]
 [ 1.5      , 0.5      ,]
 [ 0.5      , 1.5      ,]
 [ 1.5      , 1.5      ,]
 [ 2.83333333, 0.5      ,]
 [ 3.83333333, 0.5      ,]
 [ 2.5      , 0.83333333,]
 [ 3.5      , 0.83333333,]
```

```

[ 2.16666667, 0.5      ,]
[ 3.16666667, 0.5      ,]
[ 2.5        , 0.16666667,]
[ 3.5        , 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))
>>> threeDAddedMesh.getCellCenters()
[[ 0.5, 0.5, 0.5,]
 [ 1.5, 0.5, 0.5,]
 [ 0.5, 1.5, 0.5,]
 [ 1.5, 1.5, 0.5,]
 [ 0.5, 0.5, 1.5,]
 [ 1.5, 0.5, 1.5,]
 [ 0.5, 1.5, 1.5,]
 [ 1.5, 1.5, 1.5,]
 [ 2.5, 0.5, 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, 0.5,]
 [ 1.5, 0.5,]
 [ 0.5, 1.5,]
 [ 1.5, 1.5,]] 1

```

The factor can be a scalar

```

>>> dilatedMesh = baseMesh * 3
>>> dilatedMesh.getCellCenters()

```

```
[[ 1.5, 1.5,]
 [ 4.5, 1.5,]
 [ 1.5, 4.5,]
 [ 4.5, 4.5,]]
```

or a vector

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

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

```
>>> dilatedMesh = baseMesh * (3, 2, 1)
Traceback (most recent call last):
...
ValueError: frames are not aligned
```

```
__repr__(self)
```

```
getCellCenters(self)
```

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

Return Cell objects of `Mesh`.

```
getCellVolumes(self)
```

```
getDim(self)
```

```
getExteriorFaces(self)
```

```
getFaces(self, filter=None, **args)
```

Return Face objects of `Mesh`.

```
getInteriorFaces(self)
```

```
getNearestCell(self, point)
```

`getNumberOfCells(self)`

`setScale(self, value=1.0)`

3.2 Module `fipy.meshes.grid1D`

Functions

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

3.3 Module `fipy.meshes.grid2D`

Functions

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

3.4 Module `fipy.meshes.grid3D`


Functions

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

3.5 Module `fipy.meshes.meshIterator`

Class `FaceIterator`

```
fipy.meshes.meshIterator.MeshIterator
```



```
FaceIterator
```

Methods

```
getAreas(self)
```

```
getCenters(self)
```

Inherited from `MeshIterator`: `__init__`, `__add__`, `__array__`, `__getitem__`, `__iter__`, `__len__`, `__repr__`, `__str__`, `getIDs`, `getMesh`, `where`

Class `MeshIterator`

Known Subclasses: `FaceIterator`

Methods

```
__init__(self, mesh, ids=(), checkIDs=False)
```

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

```
__array__(self, t=None)
```

```
__getitem__(self, index)
```

```
__iter__(self)
```

```
__len__(self)
```

`__repr__(self)`

`__str__(self)`

`getIDs(self)`

`getMesh(self)`

`where(self, condition)`

3.6 Module *fipy.meshes.numMesh.cell*

Class *Cell*

Methods

__init__(self, mesh, id)

__cmp__(self, cell)

getCenter(self)

getID(self)

getMesh(self)

getNormal(self, index)

3.7 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

`getArea(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.
`getID(self)`

`getMesh(self)`

3.8 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`

```
exceptions.Exception └─  
                        MeshExportError
```

Methods

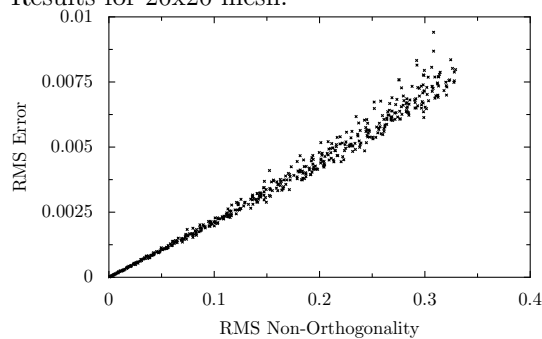
Inherited from `Exception`: `__init__`, `__getitem__`, `__str__`

3.9 Module `fipy.meshes.numMesh.gmshImport`

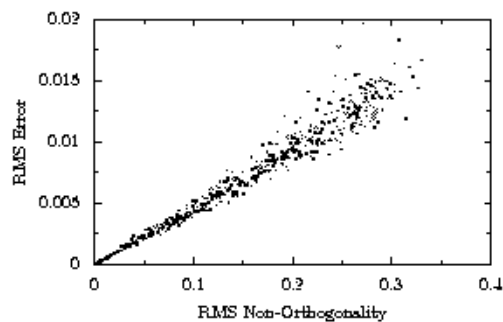
This module takes a Gmsh output file (`.msh`) and converts it into a FiPy mesh. This currently supports triangular and tetrahedral meshes only.

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 create 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.  0. ]
 [ 0.5 0.5 1. ]
 [ 1.  0.  0. ]
 [ 0.5 1.  0. ]
 [ 0.5 0.5 0.5]]

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

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

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

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

>>> print mesh._getFaceVertexIDs()
[[2 0]
```

```

[0 1]
[1 2]
[0 3]
[3 2]
[1 4]
[4 2]
[4 3]
[3 5]
[5 4]
[3 6]
[6 5]
[5 7]
[7 4]
[7 6]]

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

```

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

```

>>> cellSize = 0.7
>>> radius = 1.
>>> lines = ['cellSize = ' + str(cellSize) + ';\n',
...          'radius = ' + str(radius) + ';\n',
...          'Point(1) = {0, 0, 0, cellSize};\n',
...          'Point(2) = {-radius, 0, 0, cellSize};\n',
...          'Point(3) = {0, radius, 0, cellSize};\n',
...          'Point(4) = {radius, 0, 0, cellSize};\n',
...          'Point(5) = {0, -radius, 0, cellSize};\n',
...          'Circle(6) = {2, 1, 3};\n',
...          'Circle(7) = {3, 1, 4};\n',
...          'Circle(8) = {4, 1, 5};\n',
...          'Circle(9) = {5, 1, 2};\n',
...          'Line Loop(10) = {6, 7, 8, 9} ;\n',
...          'Plane Surface(11) = {10};\n']

>>> def buildMesh(lines):
...     import tempfile
...     (f, geomName) = tempfile.mkstemp('.geo')
...     file = open(geomName, 'w')
...     file.writelines(lines)

```



```

...     file.close()
...     import os
...     os.close(f)
...     import sys
...     if sys.platform == 'win32':
...         meshName = 'tmp.msh'
...     else:
...         (f, meshName) = tempfile.mkstemp('.msh')
...     os.system('gmsh ' + geomName + ' -2 -v 0 -format msh -o ' + meshName)
...     if sys.platform != 'win32':
...         os.close(f)
...     os.remove(geomName)
...     mesh = GmshImporter2D(meshName)
...     os.remove(meshName)
...     return mesh

```

Check that the sign of the mesh volumes is correct

```

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

```

Class *GmshImporter2D*

```

fipy.meshes.common.mesh.Mesh ┌
                               │
fipy.meshes.numMesh.mesh.Mesh ┌
                               │
fipy.meshes.numMesh.mesh2D.Mesh2D ┌
                                   │
                                   GmshImporter2D

```

Known Subclasses: [GmshImporter2DIn3DSpace](#), [_AdaptiveMesh2D](#)

Methods

`__init__(self, filename, coordDimensions=2)`

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

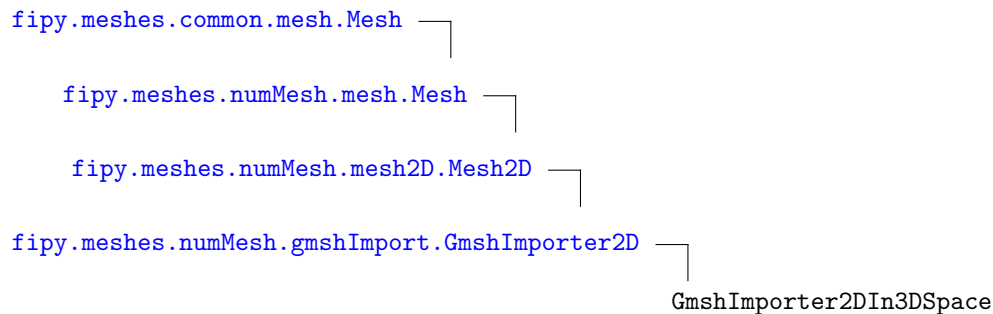
`getCellVolumes(self)`

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

Inherited from `Mesh`: `__repr__`, `getCellCenters`, `getCells`, `getDim`, `getFaces`, `getNearestCell`, `getNumberOfCells`, `setScale`

Inherited from `Mesh`: `__add__`, `__getstate__`, `__setstate__`, `getExteriorFaces`, `getFaceCellIDs`, `getFaceCenters`, `getInteriorFaces`, `getVertexCoords`

Inherited from `Mesh2D`: `__mul__`

Class `GmshImporter2DIn3DSpace`**Methods**

`__init__(self, filename)`

Overrides: `fipy.meshes.numMesh.gmshImport.GmshImporter2D.__init__()`

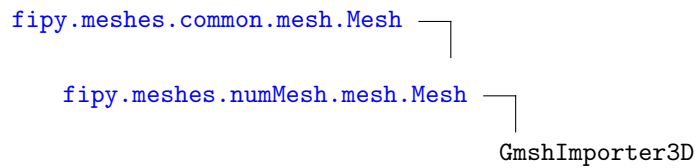
Inherited from `Mesh`: `__repr__`, `getCellCenters`, `getCells`, `getDim`, `getFaces`, `getNearestCell`, `getNumberOfCells`, `setScale`

Inherited from `GmshImporter2D`: `getCellVolumes`

Inherited from `Mesh`: `__add__`, `__getstate__`, `__setstate__`, `getExteriorFaces`, `getFaceCellIDs`, `getFaceCenters`, `getInteriorFaces`, `getVertexCoords`

Inherited from `Mesh2D`: `__mul__`

Class GmshImporter3D



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

Methods

```
__init__(self, filename)
```

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

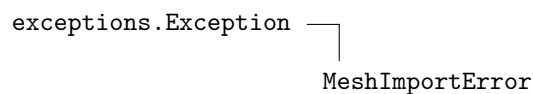
```
getCellVolumes(self)
```

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

Inherited from `Mesh`: `__repr__`, `getCellCenters`, `getCells`, `getDim`, `getFaces`, `getNearestCell`, `getNumberOfCells`, `setScale`

Inherited from `Mesh`: `__add__`, `__getstate__`, `__mul__`, `__setstate__`, `getExteriorFaces`, `getFaceCellIDs`, `getFaceCenters`, `getInteriorFaces`, `getVertexCoords`

Class MeshImportError



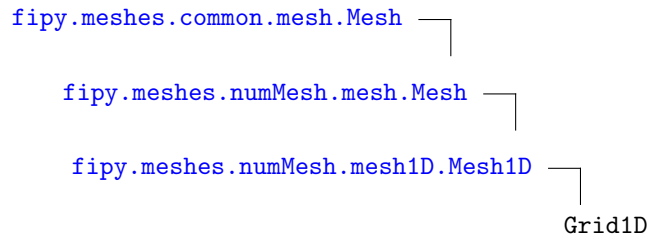
Methods

Inherited from `Exception`: `__init__`, `__getitem__`, `__str__`

3.10 Module `fipy.meshes.numMesh.grid1D`

1D Mesh

Class `Grid1D`



Known Subclasses: `PeriodicGrid1D`, `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)`

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

`__getstate__(self)`

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

`__repr__(self)`

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

`__setstate__(self, dict)`

Overrides: [fipy.meshes.numMesh.mesh.Mesh.__setstate__\(\)](#)

[getDim\(self\)](#)

Overrides: [fipy.meshes.common.mesh.Mesh.getDim\(\)](#)

[getFacesLeft\(self\)](#)

Return face on left boundary of *Grid1D* as list.

[getFacesRight\(self\)](#)

Return face on right boundary of *Grid1D* as list.

[getPhysicalShape\(self\)](#)

Return physical dimensions of *Grid1D*.

[getScale\(self\)](#)

[getShape\(self\)](#)

Inherited from [Mesh](#): [getCellCenters](#), [getCells](#), [getCellVolumes](#), [getFaces](#), [getNearestCell](#),
[getNumberOfCells](#), [setScale](#)

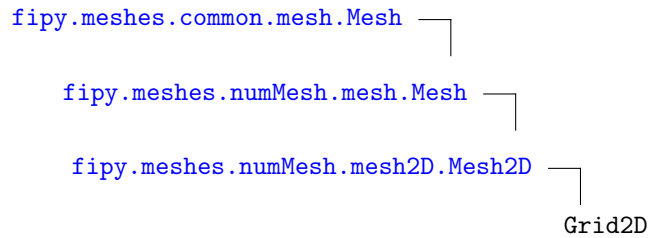
Inherited from [Mesh](#): [__add__](#), [getExteriorFaces](#), [getFaceCellIDs](#), [getFaceCenters](#),
[getInteriorFaces](#), [getVertexCoords](#)

Inherited from [Mesh1D](#): [__mul__](#)

3.11 Module `fipy.meshes.numMesh.grid2D`

2D rectangular Mesh

Class `Grid2D`



Known Subclasses: `PeriodicGrid2D`, `UniformGrid2D`

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)
```

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

```
__getstate__(self)
```

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

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

```
__repr__(self)
```

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

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

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

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

```
getFacesBottom(self)
```

Return list of faces on bottom boundary of `Grid2D`.

```

>>> mesh = Grid2D(nx = 3, ny = 2, dx = 0.5, dy = 2.)
>>> numerix.allequal((0, 1, 2), mesh.getFacesBottom())
1
  
```

```
getFacesLeft(self)
```

Return list of faces on left boundary of `Grid2D`.

```
>>> mesh = Grid2D(nx = 3, ny = 2, dx = 0.5, dy = 2.)
>>> numerix.allequal((9, 13), mesh.getFacesLeft())
1
```

`getFacesRight(self)`

Return list of faces on right boundary of Grid2D.

```
>>> mesh = Grid2D(nx = 3, ny = 2, dx = 0.5, dy = 2.)
>>> numerix.allequal((12, 16), mesh.getFacesRight())
1
```

`getFacesTop(self)`

Return list of faces on top boundary of Grid2D.

```
>>> mesh = Grid2D(nx = 3, ny = 2, dx = 0.5, dy = 2.)
>>> numerix.allequal((6, 7, 8), mesh.getFacesTop())
1
```

`getPhysicalShape(self)`

Return physical dimensions of Grid2D.

`getScale(self)`

`getShape(self)`

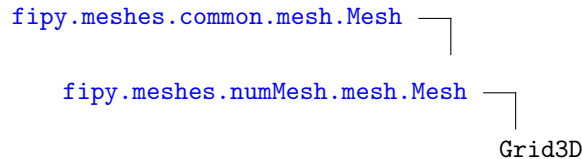
Inherited from `Mesh`: `getCellCenters`, `getCells`, `getCellVolumes`, `getDim`, `getFaces`,
`getNearestCell`, `getNumberOfCells`, `setScale`

Inherited from `Mesh`: `__add__`, `getExteriorFaces`, `getFaceCellIDs`, `getFaceCenters`,
`getInteriorFaces`, `getVertexCoords`

Inherited from `Mesh2D`: `__mul__`

3.12 Module `fipy.meshes.numMesh.grid3D`

Class `Grid3D`



Known Subclasses: `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)
```

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

```
__getstate__(self)
```

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

```
__repr__(self)
```

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

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

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

```
getFacesBack(self)
```

Return list of faces on back boundary of `Grid3D`.

```

>>> mesh = Grid3D(nx = 3, ny = 2, nz = 1, dx = 0.5, dy = 2., dz = 4.)
>>> numerix.allequal((6, 7, 8, 9, 10, 11), mesh.getFacesBack())
1
  
```

```
getFacesBottom(self)
```

Return list of faces on bottom boundary of `Grid3D`.

```

>>> mesh = Grid3D(nx = 3, ny = 2, nz = 1, dx = 0.5, dy = 2., dz = 4.)
>>> numerix.allequal((12, 13, 14), mesh.getFacesBottom())
1
  
```

getFacesFront(*self*)

Return list of faces on front boundary of Grid3D.

```
>>> mesh = Grid3D(nx = 3, ny = 2, nz = 1, dx = 0.5, dy = 2., dz = 4.)
>>> numerix.allequal((0, 1, 2, 3, 4, 5), mesh.getFacesFront())
1
```

getFacesLeft(*self*)

Return list of faces on left boundary of Grid3D.

```
>>> mesh = Grid3D(nx = 3, ny = 2, nz = 1, dx = 0.5, dy = 2., dz = 4.)
>>> numerix.allequal((21, 25), mesh.getFacesLeft())
1
```

getFacesRight(*self*)

Return list of faces on right boundary of Grid3D.

```
>>> mesh = Grid3D(nx = 3, ny = 2, nz = 1, dx = 0.5, dy = 2., dz = 4.)
>>> numerix.allequal((24, 28), mesh.getFacesRight())
1
```

getFacesTop(*self*)

Return list of faces on top boundary of Grid3D.

```
>>> mesh = Grid3D(nx = 3, ny = 2, nz = 1, dx = 0.5, dy = 2., dz = 4.)
>>> numerix.allequal((18, 19, 20), mesh.getFacesTop())
1
```

getPhysicalShape(*self*)

Return physical dimensions of Grid3D.

getScale(*self*)

getShape(*self*)

Inherited from Mesh: [getCellCenters](#), [getCells](#), [getCellVolumes](#), [getDim](#), [getFaces](#), [getNearestCell](#), [getNumberOfCells](#), [setScale](#)

Inherited from Mesh: [__add__](#), [__mul__](#), [getExteriorFaces](#), [getFaceCellIDs](#), [getFaceCenters](#), [getInteriorFaces](#), [getVertexCoords](#)

3.13 Module `fipy.meshes.numMesh.mesh`

Class `Mesh`

```
fipy.meshes.common.mesh.Mesh ┌
                               │
                               └ Mesh
```

Known Subclasses: `GmshImporter3D`, `Grid3D`, `Mesh1D`, `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.

Methods

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

`faceVertexIDs` and `cellFaceIDs` 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, 0.5,]
 [ 1.5, 0.5,]
 [ 0.5, 1.5,]
 [ 1.5, 1.5,]] 1
```

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

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

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

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

```
[ 3.5, 0.5,]
[ 2.5, 1.5,]
[ 3.5, 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))
>>> addedMesh.getCellCenters()
[[ 0.5, 0.5,]
 [ 1.5, 0.5,]
 [ 0.5, 1.5,]
 [ 1.5, 1.5,]
 [ 1.5, 0.5,]
 [ 2.5, 0.5,]
 [ 1.5, 1.5,]
 [ 2.5, 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
>>> triAddedMesh.getCellCenters()
[[ 0.5      , 0.5      ,]
 [ 1.5      , 0.5      ,]
 [ 0.5      , 1.5      ,]
 [ 1.5      , 1.5      ,]
 [ 2.83333333, 0.5      ,]
 [ 3.83333333, 0.5      ,]
 [ 2.5      , 0.83333333,]
 [ 3.5      , 0.83333333,]
 [ 2.16666667, 0.5      ,]
 [ 3.16666667, 0.5      ,]
 [ 2.5      , 0.16666667,]
 [ 3.5      , 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))
>>> threeDAddedMesh.getCellCenters()
[[ 0.5, 0.5, 0.5,]
 [ 1.5, 0.5, 0.5,]
 [ 0.5, 1.5, 0.5,]
 [ 1.5, 1.5, 0.5,]
 [ 0.5, 0.5, 1.5,]
 [ 1.5, 0.5, 1.5,]
 [ 0.5, 1.5, 1.5,]
 [ 1.5, 1.5, 1.5,]
 [ 2.5, 0.5, 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*)

`__getstate__(self)`

`__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, 0.5,]
 [ 1.5, 0.5,]
 [ 0.5, 1.5,]
 [ 1.5, 1.5,]] 1
```

The factor can be a scalar

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

or a vector

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

but the vector must have the same dimensionality as the Mesh

```
>>> dilatedMesh = baseMesh * (3, 2, 1)
Traceback (most recent call last):
...
ValueError: frames are not aligned
```

Overrides: [fipy.meshes.common.mesh.Mesh.__mul__\(\)](#) (*inherited documentation*)

`__setstate__(self, dict)`

`getExteriorFaces(self)`

Overrides: [fipy.meshes.common.mesh.Mesh.getExteriorFaces\(\)](#)

`getFaceCellIDs(self)`

`getFaceCenters(self)`

`getInteriorFaces(self)`

Overrides: [fipy.meshes.common.mesh.Mesh.getInteriorFaces\(\)](#)

`getVertexCoords(self)`

Inherited from Mesh: [__repr__](#), [getCellCenters](#), [getCells](#), [getCellVolumes](#), [getDim](#), [getFaces](#), [getNearestCell](#), [getNumberOfCells](#), [setScale](#)

Class MeshAdditionError

```
exceptions.Exception └─
                        MeshAdditionError
```

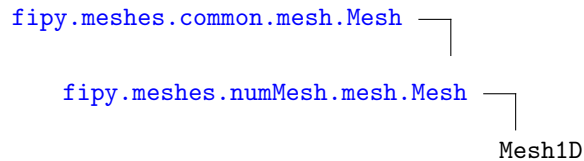
Methods

Inherited from Exception: [__init__](#), [__getitem__](#), [__str__](#)

3.14 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: `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, 0.5,]
 [ 1.5, 0.5,]
 [ 0.5, 1.5,]
 [ 1.5, 1.5,]] 1
  
```

The factor can be a scalar

```

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

or a vector

```

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

but the vector must have the same dimensionality as the Mesh

```

>>> dilatedMesh = baseMesh * (3, 2, 1)
Traceback (most recent call last):
...
ValueError: frames are not aligned
  
```

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

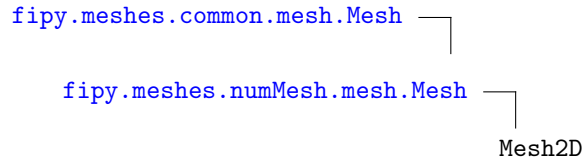
Inherited from `Mesh`: `__repr__`, `getCellCenters`, `getCells`, `getCellVolumes`, `getDim`, `getFaces`, `getNearestCell`, `getNumberOfCells`, `setScale`

Inherited from `Mesh`: `__init__`, `__add__`, `__getstate__`, `__setstate__`, `getExteriorFaces`, `getFaceCellIDs`, `getFaceCenters`, `getInteriorFaces`, `getVertexCoords`

3.15 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: `GapFillMesh`, `GmshImporter2D`, `Grid2D`, `SkewedGrid2D`, `Tri2D`, `_RefinedMesh2D`

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, 0.5,]
 [ 1.5, 0.5,]
 [ 0.5, 1.5,]
 [ 1.5, 1.5,]] 1
  
```

The factor can be a scalar

```

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

or a vector

```

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

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

```

>>> dilatedMesh = baseMesh * (3, 2, 1)
Traceback (most recent call last):
...
ValueError: frames are not aligned
  
```


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

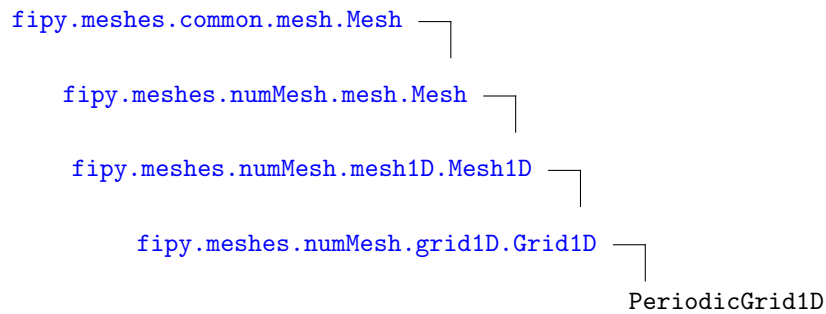
Inherited from `Mesh`: `__repr__`, `getCellCenters`, `getCells`, `getCellVolumes`, `getDim`, `getFaces`, `getNearestCell`, `getNumberOfCells`, `setScale`

Inherited from `Mesh`: `__init__`, `__add__`, `__getstate__`, `__setstate__`, `getExteriorFaces`, `getFaceCellIDs`, `getFaceCenters`, `getInteriorFaces`, `getVertexCoords`

3.16 Module `fipy.meshes.numMesh.periodicGrid1D`

Periodic 1D Mesh

Class `PeriodicGrid1D`



Creates a Periodic grid mesh.

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

```
>>> print mesh.getExteriorFaces()
[3]
```

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

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

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

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

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

Methods

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

Overrides: `fiipy.meshes.numMesh.grid1D.Grid1D.__init__()`

Inherited from `Mesh`: `getCellCenters`, `getCells`, `getCellVolumes`, `getFaces`, `getNearestCell`,
`getNumberOfCells`, `setScale`

Inherited from `Grid1D`: `__getstate__`, `__repr__`, `__setstate__`, `getDim`, `getFacesLeft`, `getFacesRight`,
`getPhysicalShape`, `getScale`, `getShape`

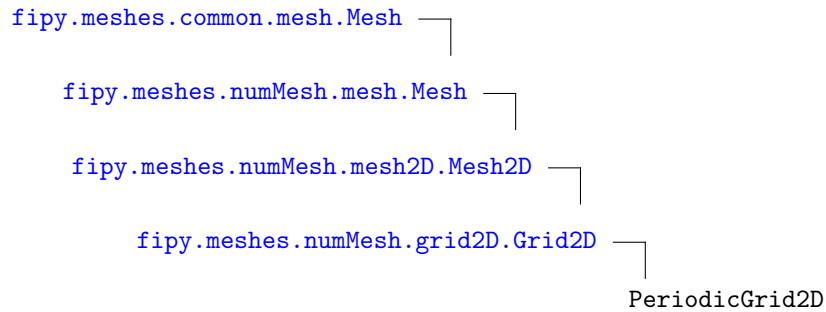
Inherited from `Mesh`: `__add__`, `getExteriorFaces`, `getFaceCellIDs`, `getFaceCenters`,
`getInteriorFaces`, `getVertexCoords`

Inherited from `Mesh1D`: `__mul__`

3.17 Module `fipy.meshes.numMesh.periodicGrid2D`

2D periodic rectangular Mesh

Class `PeriodicGrid2D`



Known Subclasses: `PeriodicGrid2DLeftRight`, `PeriodicGrid2DTopBottom`

Creates a periodic2D grid mesh with horizontal faces numbered first and then vertical faces. Vertices and cells are numbered in the usual way.

```

>>> mesh = PeriodicGrid2D(dx = 1., dy = 0.5, nx = 2, ny = 2)

>>> print mesh.getExteriorFaces()
[ 4  5  8 11]

>>> print mesh.getFaceCellIDs()
[[2 0]
 [3 1]
 [0 2]
 [1 3]
 [2 --]
 [3 --]
 [1 0]
 [0 1]
 [1 --]
 [3 2]
 [2 3]
 [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 7 2 6]
 [1 6 3 7]
 [2 10 0 9]
 [3 9 1 10]]

```

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

>>> normals = [[0, 1],
...           [0, 1],
...           [0, 1],
...           [0, 1],
...           [0, 1],
...           [0, 1],
...           [1, 0],
...           [1, 0],
...           [1, 0],
...           [1, 0],
...           [1, 0],
...           [1, 0]]

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

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

Methods

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

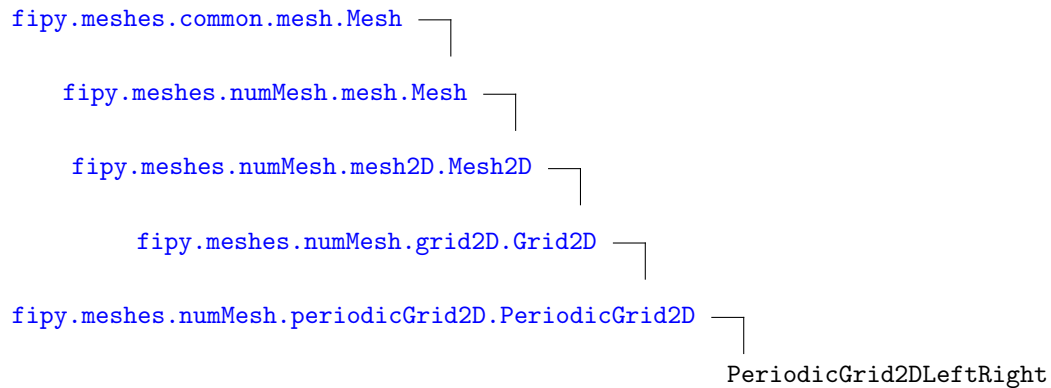
Overrides: [fipy.meshes.numMesh.grid2D.Grid2D.__init__\(\)](#)

Inherited from [Mesh](#): [getCellCenters](#), [getCells](#), [getCellVolumes](#), [getDim](#), [getFaces](#), [getNearestCell](#), [getNumberOfCells](#), [setScale](#)

Inherited from [Grid2D](#): [__getstate__](#), [__repr__](#), [__setstate__](#), [getFacesBottom](#), [getFacesLeft](#), [getFacesRight](#), [getFacesTop](#), [getPhysicalShape](#), [getScale](#), [getShape](#)

Inherited from [Mesh](#): [__add__](#), [getExteriorFaces](#), [getFaceCellIDs](#), [getFaceCenters](#), [getInteriorFaces](#), [getVertexCoords](#)

Inherited from [Mesh2D](#): [__mul__](#)

Class PeriodicGrid2DLeftRight**Methods**

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

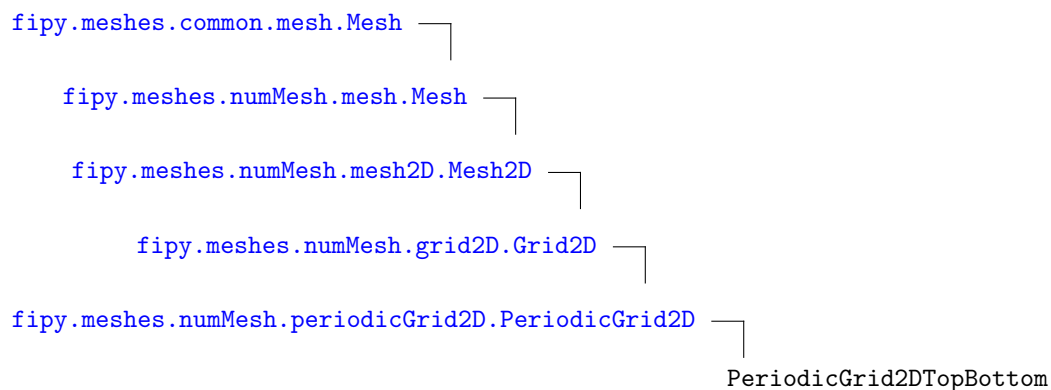
Overrides: `fipy.meshes.numMesh.periodicGrid2D.PeriodicGrid2D.__init__()`

Inherited from `Mesh`: `getCellCenters`, `getCells`, `getCellVolumes`, `getDim`, `getFaces`, `getNearestCell`, `getNumberOfCells`, `setScale`

Inherited from `Grid2D`: `__getstate__`, `__repr__`, `__setstate__`, `getFacesBottom`, `getFacesLeft`, `getFacesRight`, `getFacesTop`, `getPhysicalShape`, `getScale`, `getShape`

Inherited from `Mesh`: `__add__`, `getExteriorFaces`, `getFaceCellIDs`, `getFaceCenters`, `getInteriorFaces`, `getVertexCoords`

Inherited from `Mesh2D`: `__mul__`

Class PeriodicGrid2DTopBottom

Methods

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

Overrides: `fipy.meshes.numMesh.periodicGrid2D.PeriodicGrid2D.__init__()`

Inherited from `Mesh`: `getCellCenters`, `getCells`, `getCellVolumes`, `getDim`, `getFaces`,
`getNearestCell`, `getNumberOfCells`, `setScale`

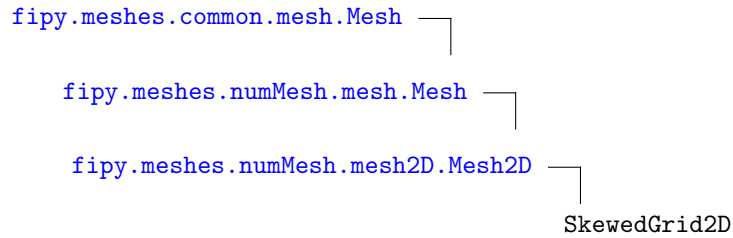
Inherited from `Grid2D`: `__getstate__`, `__repr__`, `__setstate__`, `getFacesBottom`, `getFacesLeft`,
`getFacesRight`, `getFacesTop`, `getPhysicalShape`, `getScale`, `getShape`

Inherited from `Mesh`: `__add__`, `getExteriorFaces`, `getFaceCellIDs`, `getFaceCenters`,
`getInteriorFaces`, `getVertexCoords`

Inherited from `Mesh2D`: `__mul__`

3.18 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)
```

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

```
__getstate__(self)
```

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

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

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

```
getFacesBottom(self)
```

Return list of faces on bottom boundary of Grid2D.

```
getFacesLeft(self)
```

Return list of faces on left boundary of Grid2D.

```
getFacesRight(self)
```

Return list of faces on right boundary of Grid2D.

```
getFacesTop(self)
```

Return list of faces on top boundary of Grid2D.

```
getPhysicalShape(self)
```

Return physical dimensions of Grid2D.

```
getScale(self)
```

`getShape(self)`

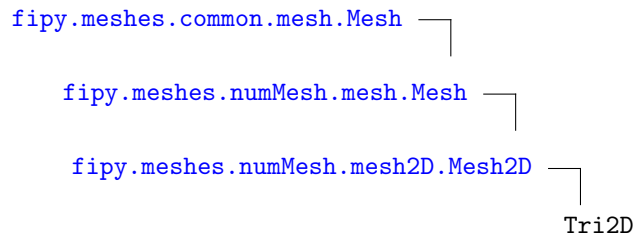
Inherited from `Mesh`: `__repr__`, `getCellCenters`, `getCells`, `getCellVolumes`, `getDim`, `getFaces`,
`getNearestCell`, `getNumberOfCells`, `setScale`

Inherited from `Mesh`: `__add__`, `getExteriorFaces`, `getFaceCellIDs`, `getFaceCenters`,
`getInteriorFaces`, `getVertexCoords`

Inherited from `Mesh2D`: `__mul__`

3.19 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 $dx \neq dy$, the line segments connecting the cell centers will not be orthogonal to the faces.

nx, ny: The number of boxes in the X direction and the Y direction. The total number of boxes will be equal to $nx * ny$, and the total number of cells will be equal to $4 * nx * ny$.

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

```
__getstate__(self)
```

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

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

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

```
getFacesBottom(self)
```

Return list of faces on bottom boundary of `Grid2D`.

```
getFacesLeft(self)
```

Return list of faces on left boundary of Grid2D.

`getFacesRight(self)`

Return list of faces on right boundary of Grid2D.

`getFacesTop(self)`

Return list of faces on top boundary of Grid2D.

`getPhysicalShape(self)`

Return physical dimensions of Grid2D.

`getScale(self)`

`getShape(self)`

Inherited from `Mesh`: `__repr__`, `getCellCenters`, `getCells`, `getCellVolumes`, `getDim`, `getFaces`,
`getNearestCell`, `getNumberOfCells`, `setScale`

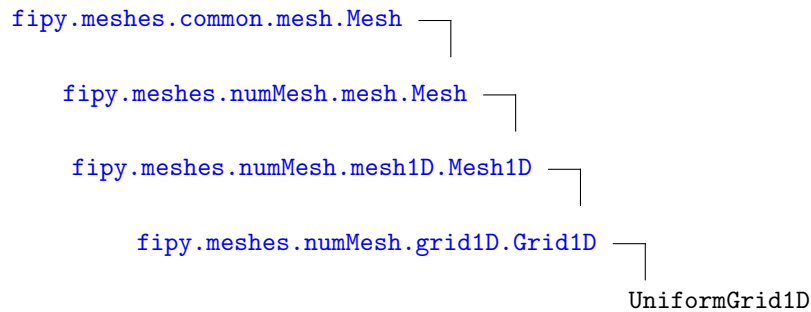
Inherited from `Mesh`: `__add__`, `getExteriorFaces`, `getFaceCellIDs`, `getFaceCenters`,
`getInteriorFaces`, `getVertexCoords`

Inherited from `Mesh2D`: `__mul__`

3.20 Module `fipy.meshes.numMesh.uniformGrid1D`

1D Mesh

Class `UniformGrid1D`



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,))
```

Overrides: `fipy.meshes.numMesh.grid1D.Grid1D.__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, 0.5,]
 [ 1.5, 0.5,]
 [ 0.5, 1.5,]
 [ 1.5, 1.5,]] 1

```

The factor can be a scalar

```

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

```

or a vector

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

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

```
>>> dilatedMesh = baseMesh * (3, 2, 1)
Traceback (most recent call last):
...
ValueError: frames are not aligned
```

Overrides: [fipy.meshes.numMesh.mesh1D.Mesh1D.__mul__\(\)](#) (*inherited documentation*)

`getCellCenters(self)`

Overrides: [fipy.meshes.common.mesh.Mesh.getCellCenters\(\)](#)

`getCellVolumes(self)`

Overrides: [fipy.meshes.common.mesh.Mesh.getCellVolumes\(\)](#)

`getFaceCellIDs(self)`

Overrides: [fipy.meshes.numMesh.mesh.Mesh.getFaceCellIDs\(\)](#)

`getFaceCenters(self)`

Overrides: [fipy.meshes.numMesh.mesh.Mesh.getFaceCenters\(\)](#)

`getInteriorFaces(self)`

Overrides: [fipy.meshes.numMesh.mesh.Mesh.getInteriorFaces\(\)](#)

`getVertexCoords(self)`

Overrides: [fipy.meshes.numMesh.mesh.Mesh.getVertexCoords\(\)](#)

Inherited from `Mesh`: `getCells`, `getFaces`, `getNearestCell`, `getNumberOfCells`, `setScale`

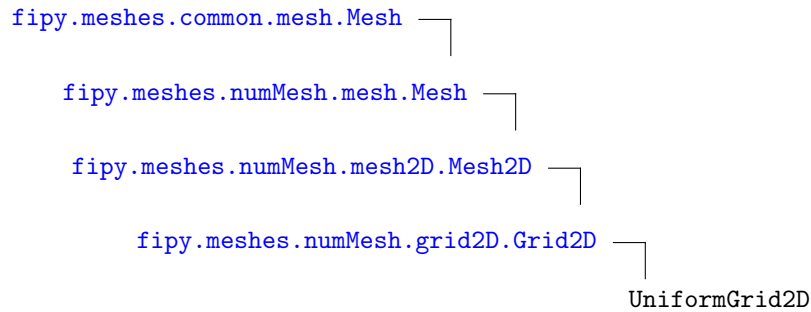
Inherited from `Grid1D`: `__getstate__`, `__repr__`, `__setstate__`, `getDim`, `getFacesLeft`, `getFacesRight`, `getPhysicalShape`, `getScale`, `getShape`

Inherited from `Mesh`: `__add__`, `getExteriorFaces`

3.21 Module `fipy.meshes.numMesh.uniformGrid2D`

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

Class `UniformGrid2D`



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))
```

Overrides: `fipy.meshes.numMesh.grid2D.Grid2D.__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, 0.5,]
 [ 1.5, 0.5,]
 [ 0.5, 1.5,]
 [ 1.5, 1.5,]] 1

```

The factor can be a scalar

```

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

```

or a vector

```

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

```

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

```
>>> dilatedMesh = baseMesh * (3, 2, 1)
Traceback (most recent call last):
...
ValueError: frames are not aligned
```

Overrides: `fiipy.meshes.numMesh.mesh2D.Mesh2D.__mul__()` (*inherited documentation*)

`getCellCenters(self)`

Overrides: `fiipy.meshes.common.mesh.Mesh.getCellCenters()`

`getCellVolumes(self)`

Overrides: `fiipy.meshes.common.mesh.Mesh.getCellVolumes()`

`getExteriorFaces(self)`

Overrides: `fiipy.meshes.numMesh.mesh.Mesh.getExteriorFaces()`

`getFaceCellIDs(self)`

Overrides: `fiipy.meshes.numMesh.mesh.Mesh.getFaceCellIDs()`

`getFaceCenters(self)`

Overrides: `fiipy.meshes.numMesh.mesh.Mesh.getFaceCenters()`

`getInteriorFaces(self)`

Overrides: `fiipy.meshes.numMesh.mesh.Mesh.getInteriorFaces()`

`getVertexCoords(self)`

Overrides: `fiipy.meshes.numMesh.mesh.Mesh.getVertexCoords()`

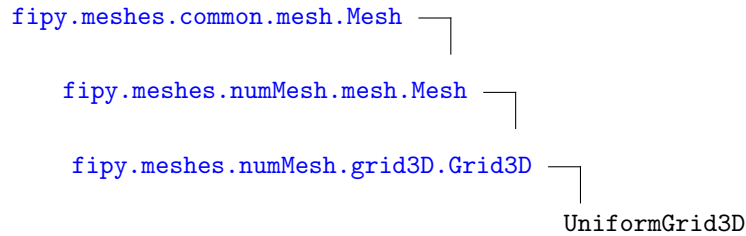
Inherited from `Mesh`: `getCells`, `getDim`, `getFaces`, `getNearestCell`, `getNumberOfCells`, `setScale`

Inherited from `Grid2D`: `__getstate__`, `__repr__`, `__setstate__`, `getFacesBottom`, `getFacesLeft`, `getFacesRight`, `getFacesTop`, `getPhysicalShape`, `getScale`, `getShape`

Inherited from `Mesh`: `__add__`

3.22 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))
```

Overrides: `fipy.meshes.numMesh.grid3D.Grid3D.__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, 0.5,]
 [ 1.5, 0.5,]
 [ 0.5, 1.5,]
 [ 1.5, 1.5,]] 1

```

The `factor` can be a scalar

```

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

```

or a vector


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

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

```
>>> dilatedMesh = baseMesh * (3, 2, 1)
Traceback (most recent call last):
...
ValueError: frames are not aligned
```

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

`getCellCenters(self)`

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

`getCellVolumes(self)`

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

`getExteriorFaces(self)`

Overrides: `fipy.meshes.numMesh.mesh.Mesh.getExteriorFaces()`

`getFaceCellIDs(self)`

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

`getFaceCenters(self)`

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

`getInteriorFaces(self)`

Overrides: `fipy.meshes.numMesh.mesh.Mesh.getInteriorFaces()`

`getVertexCoords(self)`

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

Inherited from `Mesh`: `getCells`, `getDim`, `getFaces`, `getNearestCell`, `getNumberOfCells`, `setScale`

Inherited from `Grid3D`: `__getstate__`, `__repr__`, `__setstate__`, `getFacesBack`, `getFacesBottom`, `getFacesFront`, `getFacesLeft`, `getFacesRight`, `getFacesTop`, `getPhysicalShape`, `getScale`, `getShape`

Inherited from `Mesh`: `__add__`

3.23 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

`__repr__(self)`

Textual representation of `Cell`.

`getBoundingCells(self)`

`getCenter(self)`

Return the coordinates of the `Cell` center.

`getFaceIDs(self)`

`getFaceOrientations(self)`

`getFaces(self)`

Return the faces bounding the `Cell`.

`getID(self)`

Return the id of this `Cell`.

`getVolume(self)`

Return the volume of the `Cell`.

3.24 Module `fipy.meshes.pyMesh.face`

Face within a Mesh

Class Face

Known Subclasses: [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

`__repr__(self)`

Textual representation of Face.

`addBoundingCell(self, cell, orientation)`

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

`getArea(self)`

Return the area of the Face.

`getCellDistance(self)`

Return the distance between adjacent `Cell` centers.

`getCellID(self, index=0)`

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

`getCells(self)`

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

`getCenter(self)`

Return the coordinates of the Face center.

`getID(self)`


`getNormal(self)`

Return the unit normal vector

3.25 Module `fipy.meshes.pyMesh.face2D`

1D (edge) Face in a 2D Mesh

Class `Face2D`

`fipy.meshes.pyMesh.face.Face` 
Face2D

1D (edge) Face in a 2D Mesh
Face2D is bounded by two Vertices.

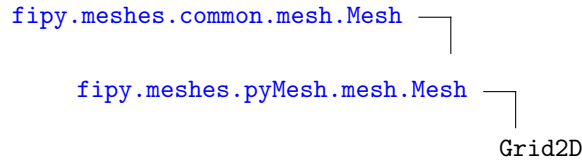
Methods

Inherited from `Face`: `__init__`, `__repr__`, `addBoundingCell`, `getArea`, `getCellDistance`, `getCellID`,
`getCells`, `getCenter`, `getID`, `getNormal`

3.26 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.pyMesh.mesh.Mesh.__init__\(\)](#)

`getCellCenters(self)`

Overrides: [fipy.meshes.common.mesh.Mesh.getCellCenters\(\)](#)

`getCellVolumes(self)`

Overrides: [fipy.meshes.common.mesh.Mesh.getCellVolumes\(\)](#)

`getFacesBottom(self)`

Return list of faces on bottom boundary of Grid2D.

`getFacesLeft(self)`

Return list of faces on left boundary of Grid2D.

`getFacesRight(self)`

Return list of faces on right boundary of Grid2D.

`getFacesTop(self)`

Return list of faces on top boundary of Grid2D.

`getPhysicalShape(self)`

Return physical dimensions of Grid2D.

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

`getShape(self)`

Return cell dimensions Grid2D.


Inherited from Mesh: `__add__`, `__mul__`, `__repr__`, `getCells`, `getDim`, `getFaces`, `getInteriorFaces`,
`getNearestCell`, `getNumberOfCells`

Inherited from Mesh: `getExteriorFaces`, `getFaceOrientations`, `getScale`, `setScale`

3.27 Module `fipy.meshes.pyMesh.mesh`

Generic mesh class
Meshes contain cells, faces, and vertices.

Class Mesh

`fipy.meshes.common.mesh.Mesh`  Mesh

Known Subclasses: `Grid2D`

Methods

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

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

Return all Faces of Mesh that are on a Mesh boundary
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 `Mesh`: `__add__`, `__mul__`, `__repr__`, `getCellCenters`, `getCells`, `getCellVolumes`, `getDim`, `getFaces`, `getInteriorFaces`, `getNearestCell`, `getNumberOfCells`

3.28 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.

`__repr__(self)`

Textual representation of Vertex.

`getCoordinates(self)`

Return coordinates of Vertex.

Chapter 4

Package `fipy.models`

4.1 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.

4.2 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,

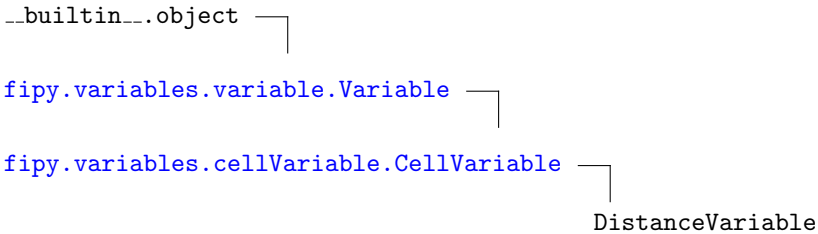
$$\frac{\partial \phi}{\partial t} + u|\nabla \phi| = 0.$$

Parameters

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

4.3 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.

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

and in 3D,

$$X_{\min} = \frac{1}{3!} |\vec{s} \cdot (\vec{t} \times \vec{u})|$$

where the vectors \vec{s} , \vec{t} and \vec{u} represent the vectors from the cell of interest to the neighboring cell.

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.cellVariable.CellVariable.__init__()`

```
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.

`extendVariable(self, extensionVariable, deleteIslands=False)`

Takes a `cellVariable` and extends the variable from the zero to the region encapsulated by the `narrowBandWidth`.

Parameters

extensionVariable: The variable to extend from the zero level set.

deleteIslands: Sets the temporary level set value to zero in isolated cells.

`getCellInterfaceAreas(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 circumference of a circle is, in fact, $2\pi r$.

```
>>> mesh = Grid2D(dx = 0.05, dy = 0.05, nx = 20, ny = 20)
>>> r = 0.25
>>> rad = numerix.sqrt((mesh.getCellCenters()[:,0] - .5)**2
...                   + (mesh.getCellCenters()[:,1] - .5)**2) - r
>>> distanceVariable = DistanceVariable(mesh = mesh, value = rad)
>>> print numerix.sum(distanceVariable.getCellInterfaceAreas())
1.57984690073
```


Inherited from object: `__delattr__`, `__getattr__`, `__hash__`, `__reduce__`, `__reduce_ex__`, `__setattr__`

Inherited from `CellVariable`: `__call__`, `__getstate__`, `__setstate__`, `copy`, `getArithmeticFaceValue`, `getCellVolumeAverage`, `getFaceGrad`, `getFaceGradAverage`, `getFaceValue`, `getGaussGrad`, `getGrad`, `getHarmonicFaceValue`, `getLeastSquaresGrad`, `getOld`, `getValue`, `setValue`, `updateOld`

Inherited from `Variable`: `__abs__`, `__add__`, `__and__`, `__array__`, `__array_wrap__`, `__div__`, `__eq__`, `__float__`, `__ge__`, `__getitem__`, `__gt__`, `__le__`, `__len__`, `__lt__`, `__mod__`, `__mul__`, `__ne__`, `__neg__`, `__or__`, `__pos__`, `__pow__`, `__radd__`, `__rdiv__`, `__repr__`, `__rmul__`, `__rpow__`, `__rsub__`, `__setitem__`, `__str__`, `__sub__`, `allclose`, `allequal`, `arccos`, `arccosh`, `arcsin`, `arcsinh`, `arctan`, `arctan2`, `arctanh`, `cacheMe`, `ceil`, `conjugate`, `cos`, `cosh`, `dontCacheMe`, `dot`, `exp`, `floor`, `getMag`, `getMesh`, `getName`, `getNumericValue`, `getShape`, `getSubscribedVariables`, `getTypecode`, `getUnit`, `inBaseUnits`, `inUnitsOf`, `log`, `log10`, `put`, `reshape`, `setName`, `sin`, `sinh`, `sqrt`, `sum`, `take`, `tan`, `tanh`, `tostring`, `transpose`

Static Methods

Inherited from `Variable`: `__new__`

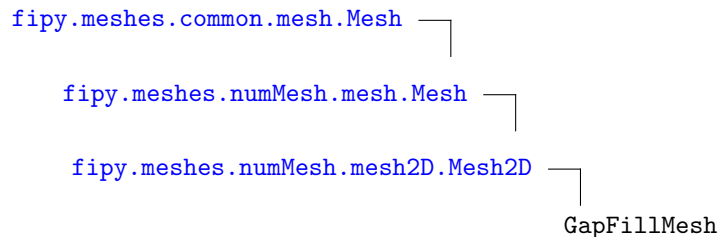
Class Variables

Name	Description
Inherited from <code>Variable</code> : <code>__array_priority__</code>	

4.4 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: [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.getBottomFaces(), 0.),
...                                   FixedValue(mesh.getTopFaces(), domainHeight)))

```

Evaluate the result:

```

>>> centers = mesh.getCellCenters()[:,1]
>>> localErrors = (centers - var)**2 / centers**2
>>> globalError = numerix.sqrt(numerix.sum(localErrors) / mesh.getNumberOfCells())
>>> argmax = numerix.argmax(localErrors)

```

```
>>> print numerix.sqrt(localErrors[argmax]) < 0.05
1
>>> print globalError < 0.02
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.numMesh.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)
```

```
getBottomFaces(self)
```

```
getCellIDsAboveFineRegion(self)
```

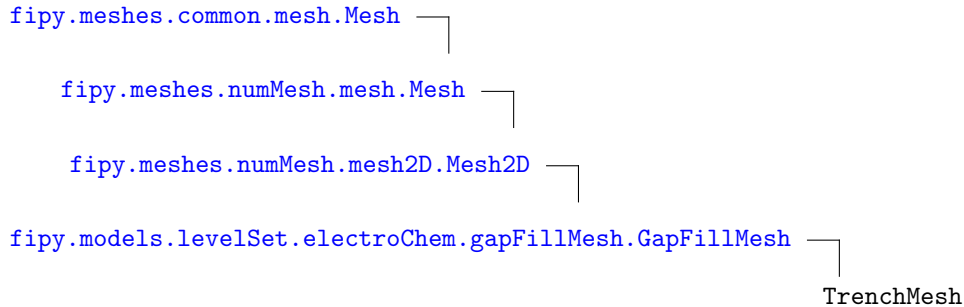
```
getFineMesh(self)
```

```
getTopFaces(self)
```

Inherited from [Mesh](#): [__repr__](#), [getCellCenters](#), [getCells](#), [getCellVolumes](#), [getDim](#), [getFaces](#), [getNearestCell](#), [getNumberOfCells](#), [setScale](#)

Inherited from [Mesh](#): [__add__](#), [getExteriorFaces](#), [getFaceCellIDs](#), [getFaceCenters](#), [getInteriorFaces](#), [getVertexCoords](#)

Inherited from [Mesh2D](#): [_mul__](#)

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()))
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.getBottomFaces(), 0.),
...                                   FixedValue(mesh.getTopFaces(), domainHeight)))

```

Evaluate the result:

```

>>> centers = mesh.getCellCenters()[:,1]
>>> 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.models.levelSet.electroChem.gapFillMesh.GapFillMesh.__init__\(\)](#)

```
__getstate__(self)
```

Overrides: [fipy.models.levelSet.electroChem.gapFillMesh.GapFillMesh.__getstate__\(\)](#)

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

Overrides: [fipy.models.levelSet.electroChem.gapFillMesh.GapFillMesh.__setstate__\(\)](#)

```
getElectrolyteMask(self)
```

Inherited from [Mesh](#): [__repr__](#), [getCellCenters](#), [getCells](#), [getCellVolumes](#), [getDim](#), [getFaces](#),
[getNearestCell](#), [getNumberOfCells](#), [setScale](#)

Inherited from [Mesh](#): [__add__](#), [getExteriorFaces](#), [getFaceCellIDs](#), [getFaceCenters](#),
[getInteriorFaces](#), [getVertexCoords](#)

Inherited from [Mesh2D](#): [_mul__](#)

Inherited from [GapFillMesh](#): [buildTransitionMesh](#), [getBottomFaces](#), [getCellIDsAboveFineRegion](#),
[getFineMesh](#), [getTopFaces](#)

4.5 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

$$v(c) = cV_0$$

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

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

This is the test case,

```
>>> from fipy.tools import numerix
>>> from fipy.meshes.grid1D import Grid1D
>>> nx = 11
>>> dx = 1.
>>> mesh = Grid1D(nx = nx, dx = dx)
>>> 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 = numerix.arange(11) - 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 * (mesh.getCellCenters()[...,0] - L - dx * 3 / 2) \
...     + cinf
>>> answer[0] = 1
>>> numerix.allclose(answer, numerix.array(ionVar))
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.

4.6 Module `fipy.models.levelSet.surfactant.adsorbingSurfactantEquation`

Class `AdsorbingSurfactantEquation`

`fipy.models.levelSet.surfactant.surfactantEquation.SurfactantEquation`

`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 adsorption coefficient to zero leads to the solver not converging and an eventual failure.

```

>>> var0 = SurfactantVariable(value = (0, 0, theta0, 0, 0),
...                           distanceVar = distanceVar)
>>> bulkVar0 = CellVariable(mesh = mesh, value = (c0, c0, c0, c0, c0))

>>> eqn0 = AdsorbingSurfactantEquation(surfactantVar = var0,
...                                   distanceVar = distanceVar,
...                                   bulkVar = bulkVar0,
...                                   rateConstant = 0)

>>> eqn0.solve(var0, dt = dt)
>>> eqn0.solve(var0, dt = dt)
>>> 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.

boundaryConditions: A tuple of `boundaryConditions`.

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

dt: The time step size.

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

```
sweep(self, var, solver=LinearLUSolver(tolerance=1e-10, iterations=1000),
      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. Defaults to `LinearPCGSolver`.

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()`

4.7 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.

4.8 Module `fipy.models.levelSet.surfactant.surfactantEquation`

Class `SurfactantEquation`

Known Subclasses: [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=LinearCGSSolver(tolerance=1e-10, iterations=1000), 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.

boundaryConditions: A tuple of `boundaryConditions`.

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

dt: The time step size.

`sweep(self, var, solver=LinearCGSSolver(tolerance=1e-10, iterations=1000),
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`.

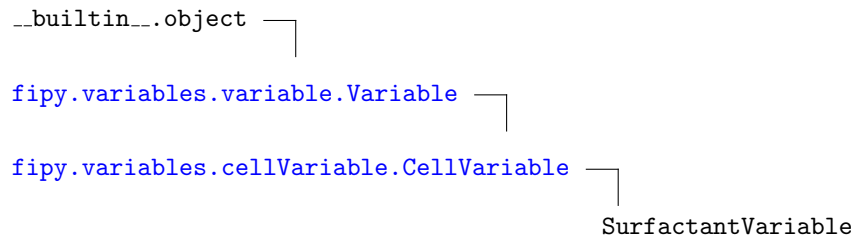
boundaryConditions: A tuple of boundaryConditions.

dt: The time step size.

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

4.9 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.cellVariable.CellVariable.__init__\(\)](#)

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.cellVariable.CellVariable.copy\(\)](#) (*inherited documentation*)

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.

Inherited from object: [__delattr__](#), [__getattr__](#), [__hash__](#), [__reduce__](#), [__reduce_ex__](#), [__setattr__](#)

Inherited from *CellVariable*: [__call__](#), [__getstate__](#), [__setstate__](#), [getArithmeticFaceValue](#), [getCellVolumeAverage](#), [getFaceGrad](#), [getFaceGradAverage](#), [getFaceValue](#), [getGaussGrad](#), [getGrad](#), [getHarmonicFaceValue](#), [getLeastSquaresGrad](#), [getOld](#), [getValue](#), [setValue](#), [updateOld](#)

Inherited from Variable: `__abs__`, `__add__`, `__and__`, `__array__`, `__array_wrap__`, `__div__`, `__eq__`, `__float__`, `__ge__`, `__getitem__`, `__gt__`, `__le__`, `__len__`, `__lt__`, `__mod__`, `__mul__`, `__ne__`, `__neg__`, `__or__`, `__pos__`, `__pow__`, `__radd__`, `__rdiv__`, `__repr__`, `__rmul__`, `__rpow__`, `__rsub__`, `__setitem__`, `__str__`, `__sub__`, `allclose`, `allequal`, `arccos`, `arccosh`, `arcsin`, `arcsinh`, `arctan`, `arctan2`, `arctanh`, `cacheMe`, `ceil`, `conjugate`, `cos`, `cosh`, `dontCacheMe`, `dot`, `exp`, `floor`, `getMag`, `getMesh`, `getName`, `getNumericValue`, `getShape`, `getSubscribedVariables`, `getTypecode`, `getUnit`, `inBaseUnits`, `inUnitsOf`, `log`, `log10`, `put`, `reshape`, `setName`, `sin`, `sinh`, `sqrt`, `sum`, `take`, `tan`, `tanh`, `tostring`, `transpose`

Static Methods

Inherited from Variable: `__new__`

Class Variables

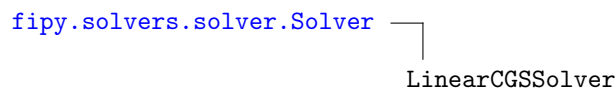
Name	Description
Inherited from Variable: <code>__array_priority__</code>	

Chapter 5

Package `fipy.solvers`

5.1 Module `fipy.solvers.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

Inherited from `Solver`: `__init__`, `__repr__`

5.2 Module `fipy.solvers.linearGMRESSolver`

Class `LinearGMRESSolver`

`fipy.solvers.solver.Solver` 
`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.

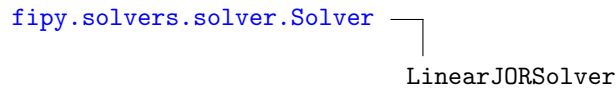
The `LinearGMRESSolver` is a wrapper class for the the [PySparse](#) `itsolvers.gmres()` and `precon.jacobi()` methods.

Methods

Inherited from `Solver`: `__init__`, `__repr__`

5.3 Module `fipy.solvers.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)`

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 `Solver`: `__repr__`

5.4 Module `fipy.solvers.linearLUSolver`

Class `LinearLUSolver`

`fipy.solvers.solver.Solver` 
`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)`

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 steps is generally required.

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

Inherited from `Solver`: `__repr__`

5.5 Module `fipy.solvers.linearPCGSolver`

Class `LinearPCGSolver`

`fipy.solvers.solver.Solver` 
`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.

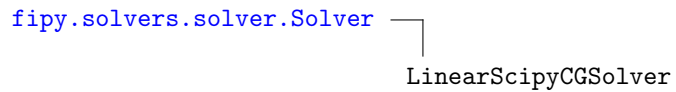
The `LinearPCGSolver` is a wrapper class for the the [PySparse](#) `itsolvers.pcg()` and `precon.ssor()` methods.

Methods

Inherited from `Solver`: `__init__`, `__repr__`

5.6 Module `fipy.solvers.linearScipyCGSolver`

Class `LinearScipyCGSolver`



The `LinearScipyCGSolver` solves a linear system of equations using the conjugent gradient. It solves a system with a symmetric coefficient matrix.

The `LinearScipyCGSolver` is a wrapper class for the the [Scipy](#) `scipy.linalg.iterative.cg()` method.

Warning


Currently the solvers that use [Scipy](#) are only useful for small systems due to the whole sparse matrix having to be turned into an array of size $N * N$.

Methods

Inherited from `Solver`: `__init__`, `__repr__`

5.7 Module `fipy.solvers.linearScipyGMRESSolver`

Class `LinearScipyGMRESSolver`

`fipy.solvers.solver.Solver` 
`LinearScipyGMRESSolver`

The `LinearScipyGMRESSolver` solves a linear system of equations using the generalised minimal residual method (GMRES) with no GMRES solves systems with a general non-symmetric coefficient matrix. The `LinearScipyGMRESSolver` is a wrapper class for the the [Scipy](#) `linalg.iterative.gmres()` method.

Warning

Currently the solvers that use [Scipy](#) are only useful for small systems due to the whole sparse matrix having to be turned into an array of size $N * N$.

Methods

Inherited from `Solver`: `__init__`, `__repr__`

5.8 Module `fipy.solvers.linearScipyLUSolver`

Class `LinearScipyLUSolver`

```

fipy.solvers.solver.Solver ┌
                             |
                             └ LinearScipyLUSolver

```

The `LinearScipyLUSolver` solves a linear system of equations using LU-factorisation. This method solves systems of general non-symmetric matrices with partial pivoting.

The `LinearScipyLUSolver` is a wrapper class for the the [SciPy](#) `scipy.linalg.lu_solve()` method.

Warning

Currently the solvers that use [Scipy](#) are only useful for small systems due to the whole sparse matrix having to be turned into an array of size $N * N$.

Methods

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

Creates a `LinearScipyLUSolver`.

Parameters

tolerance: The required error tolerance.

iterations: The number of LU decompositions to perform. For large systems a number of steps is generally required.

steps: A deprecated name for `iterations`.

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

Inherited from `Solver`: `__repr__`

5.9 Module `fipy.solvers.solver`

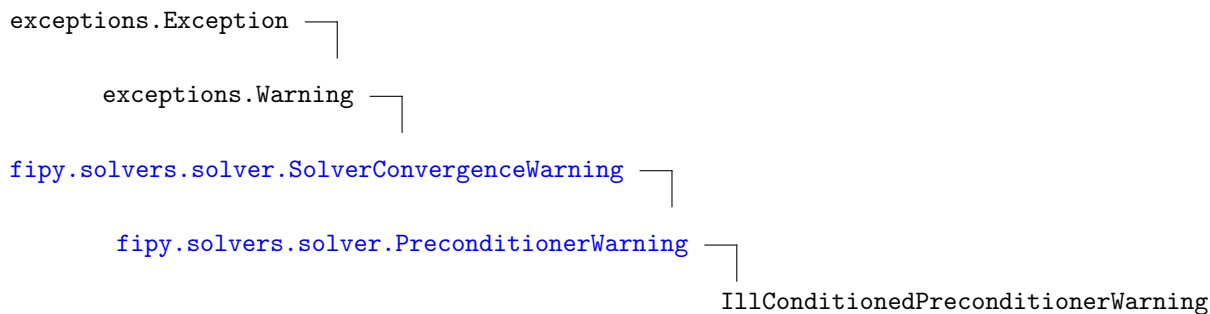
The iterative solvers may output warnings if the solution is considered unsatisfactory. If you are not interested in these warnings, you can invoke python with a warning filter such as:

```
$ 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 `IllConditionedPreconditionerWarning`



Methods

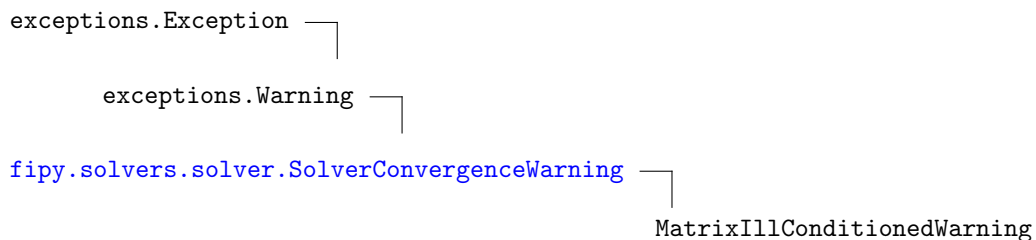
```
__str__(self)
```

Overrides: `fipy.solvers.solver.SolverConvergenceWarning.__str__()`

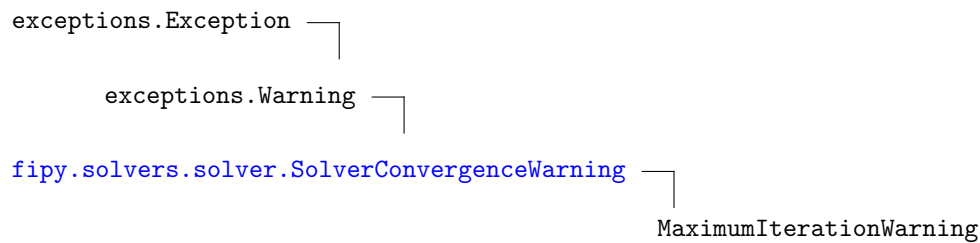
Inherited from `Exception`: `__getitem__`

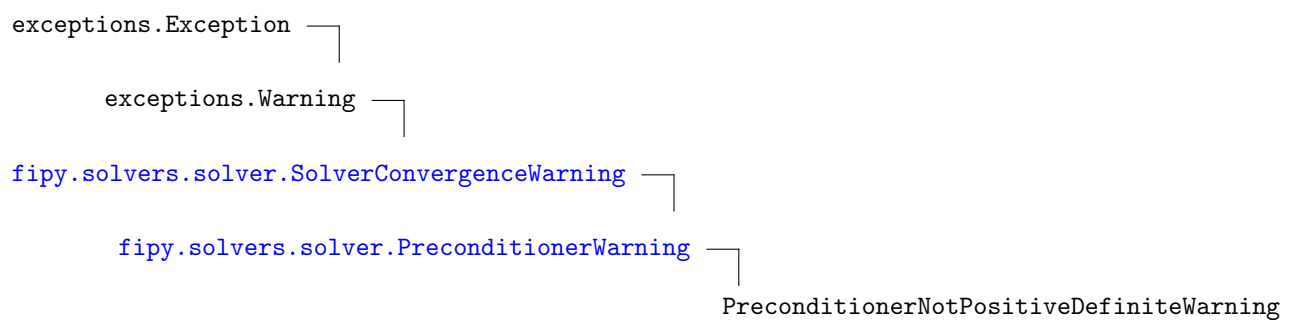
Inherited from `SolverConvergenceWarning`: `__init__`

Class `MatrixIllConditionedWarning`



Methods`__str__(self)`

Overrides: `fipy.solvers.solver.SolverConvergenceWarning.__str__()`Inherited from Exception: `__getitem__`Inherited from `SolverConvergenceWarning`: `__init__`**Class `MaximumIterationWarning`****Methods**`__str__(self)`




Overrides: `fipy.solvers.solver.SolverConvergenceWarning.__str__()`Inherited from Exception: `__getitem__`Inherited from `SolverConvergenceWarning`: `__init__`**Class `PreconditionerNotPositiveDefiniteWarning`**

Methods`__str__(self)`

Overrides: `fipy.solvers.solver.SolverConvergenceWarning.__str__()`

Inherited from `Exception`: `__getitem__`

Inherited from `SolverConvergenceWarning`: `__init__`




Class `PreconditionerWarning``exceptions.Exception` `exceptions.Warning` `fipy.solvers.solver.SolverConvergenceWarning` `PreconditionerWarning`

Known Subclasses: `IllConditionedPreconditionerWarning`,
`PreconditionerNotPositiveDefiniteWarning`

Methods

Inherited from `Exception`: `__getitem__`

Inherited from `SolverConvergenceWarning`: `__init__`, `__str__`

Class `ScalarQuantityOutOfRangeWarning``exceptions.Exception` `exceptions.Warning` `fipy.solvers.solver.SolverConvergenceWarning` `ScalarQuantityOutOfRangeWarning`**Methods**`__str__(self)`

Overrides: `fipy.solvers.solver.SolverConvergenceWarning.__str__()`

Inherited from `Exception`: `__getitem__`

Inherited from `SolverConvergenceWarning`: `__init__`

Class Solver

Known Subclasses: [LinearCGSSolver](#), [LinearGMRESSolver](#), [LinearJORSolver](#), [LinearLUSolver](#), [LinearPCGSolver](#), [LinearScipyCGSolver](#), [LinearScipyGMRESSolver](#), [LinearScipyLUSolver](#)

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)`

Create a `Solver` object.

Parameters

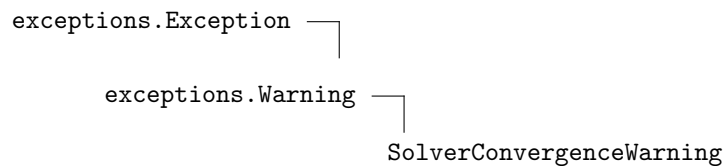
tolerance: The required error tolerance.

iterations: The maximum number of iterative steps to perform.

steps: A deprecated name for `iterations`.

`__repr__(self)`

Class SolverConvergenceWarning



Known Subclasses: [MatrixIllConditionedWarning](#), [MaximumIterationWarning](#), [PreconditionerWarning](#), [ScalarQuantityOutOfRangeWarning](#), [StagnatedSolverWarning](#)

Methods

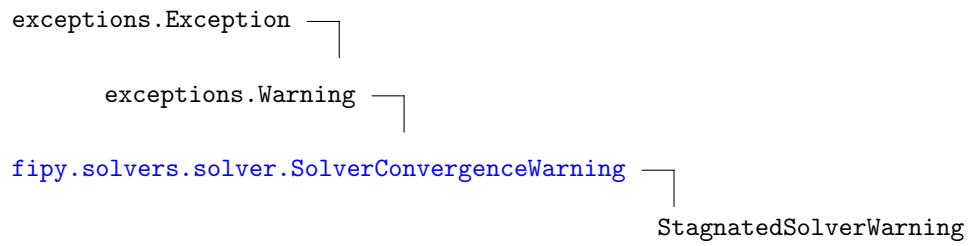
`__init__(self, solver, iter, relres)`

Overrides: `exceptions.Exception.__init__()`

`__str__(self)`

Overrides: `exceptions.Exception.__str__()`

Inherited from Exception: `__getitem__`

Class StagnatedSolverWarning**Methods**`__str__(self)`

Overrides: `fipy.solvers.solver.SolverConvergenceWarning.__str__()`

Inherited from `Exception`: `__getitem__`

Inherited from `SolverConvergenceWarning`: `__init__`

Chapter 6

Package `fipy.steps`

6.1 Functions

`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

Return Value

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

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)

Return Value

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

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

`L2error(var, matrix, RHSvector)`

Parameters

var: The `CellVariable` in question.

matrix: (ignored)

RHSvector: (ignored)

Return Value

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

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

`LINFerror(var, matrix, RHSvector)`

Parameters

var: The `CellVariable` in question.

matrix: (ignored)

RHSvector: (ignored)

Return Value

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

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

`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

Return Value

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

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

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

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

Parameters

fn: The function to call

args: The unnamed function argument list


kwargs: The named function argument dict

Return Value

the final residual

6.2 Module `fipy.steppers.pidStepper`

Class `PIDStepper`

`fipy.steppers.stepper.Stepper`  `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=0.074999999999999997,
  integral=0.17499999999999999, derivative=0.01)
```

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


Inherited from `Stepper`: `step`

Static Methods

Inherited from `Stepper`: `failFn`, `successFn`, `sweepFn`

6.3 Module `fipy.steppers.pseudoRKQSStepper`

Class `PseudoRKQSStepper`

`fipy.steppers.stepper.Stepper`  `PseudoRKQSStepper`

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

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

Methods

```
__init__(self, vardata=(), safety=0.90000000000000002,  
         pgrow=-0.20000000000000001, pshrink=-0.25,  
         errcon=0.0001890000000000001)
```

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

Inherited from `Stepper`: `step`

Static Methods

Inherited from `Stepper`: `failFn`, `successFn`, `sweepFn`

6.4 Module `fipy.steppers.stepper`

Class `Stepper`

Known Subclasses: [PIDStepper](#), [PseudoRKQSStepper](#)

Methods

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

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

Static Methods

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

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

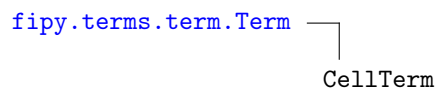
```
sweepFn(vardata, dt, *args, **kwargs)
```

Chapter 7

Package `fipy.terms`

7.1 Module `fipy.terms.cellTerm`

Class `CellTerm`



Known Subclasses: `SourceTerm`, `TransientTerm`

Attention!

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

Methods

`__init__(self, coeff=1.0)`

Create a `Term`.

Parameters

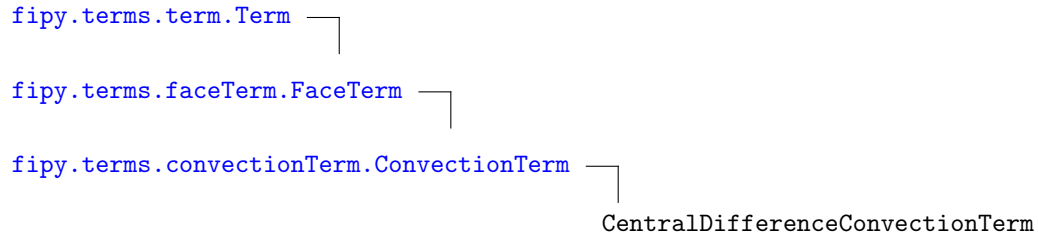
coeff: The coefficient for the term. A `CellVariable` or number. `FaceVariable` objects are also acceptable for diffusion or convection terms.

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

Inherited from `Term`: `__add__`, `__eq__`, `__neg__`, `__pos__`, `__radd__`, `__repr__`, `__rsub__`, `__sub__`, `cacheMatrix`, `cacheRHSvector`, `getMatrix`, `getRHSvector`, `justResidualVector`, `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].

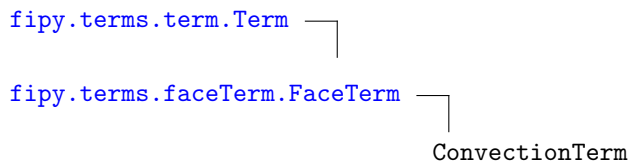
Methods

Inherited from `ConvectionTerm`: `__init__`, `__neg__`

Inherited from `Term`: `__add__`, `__eq__`, `__pos__`, `__radd__`, `__repr__`, `__rsub__`, `__sub__`, `cacheMatrix`, `cacheRHSvector`, `getMatrix`, `getRHSvector`, `justResidualVector`, `solve`, `sweep`

7.3 Module `fipy.terms.convectionTerm`

Class `ConvectionTerm`



Known Subclasses: `CentralDifferenceConvectionTerm`, `ExponentialConvectionTerm`, `HybridConvectionTerm`, `PowerLawConvectionTerm`, `UpwindConvectionTerm`

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
>>> from fipy.variables.vectorCellVariable import VectorCellVariable
>>> from fipy.variables.vectorFaceVariable import VectorFaceVariable
>>> m = Grid1D(nx = 2)
>>> cv = CellVariable(mesh = m)
>>> fv = FaceVariable(mesh = m)
>>> vcv = VectorCellVariable(mesh = m)
>>> vfv = VectorFaceVariable(mesh = m)
>>> ConvectionTerm(coeff = cv)
Traceback (most recent call last):
...
TypeError: The coefficient must be a VectorFaceVariable, VectorCellVariable, or a vector value.
>>> ConvectionTerm(coeff = fv)
Traceback (most recent call last):
...
TypeError: The coefficient must be a VectorFaceVariable, VectorCellVariable, or a vector value.
>>> ConvectionTerm(coeff = vcv)
ConvectionTerm(coeff = [[ 0.]
 [ 0.]
 [ 0.]])
>>> ConvectionTerm(coeff = vfv)
ConvectionTerm(coeff = [[ 0.]
 [ 0.]

```

```
[ 0.]]
>>> 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 VectorFaceVariable, VectorCellVariable, or a vector value.
```

Parameters

coeff: : The Term's coefficient value.

diffusionTerm: : If a DiffusionTerm is given, the ConvectionTerm uses the diffusion coefficient to calculate the Peclet number.

Overrides: `fipy.terms.faceTerm.FaceTerm.__init__()`

`__neg__(self)`

Negate the term.

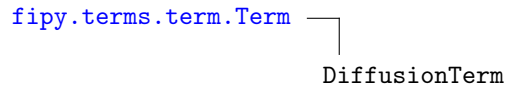
```
>>> -ConvectionTerm(coeff = 1.0)
ConvectionTerm(coeff = -1.0)
```

Overrides: `fipy.terms.term.Term.__neg__()`

Inherited from Term: `__add__`, `__eq__`, `__pos__`, `__radd__`, `__repr__`, `__rsub__`, `__sub__`, `cacheMatrix`, `cacheRHSvector`, `getMatrix`, `getRHSvector`, `justResidualVector`, `solve`, `sweep`

7.4 Module `fiy.terms.diffusionTerm`

Class `DiffusionTerm`



Known Subclasses: `ExplicitDiffusionTerm`, `ImplicitDiffusionTerm`

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

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

Create a `DiffusionTerm`.

Parameters

`coeff`: Tuple or list of `FaceVariables` or numbers.

Overrides: `fiy.terms.term.Term.__init__()`

```
__neg__(self)
```

Negate the term.

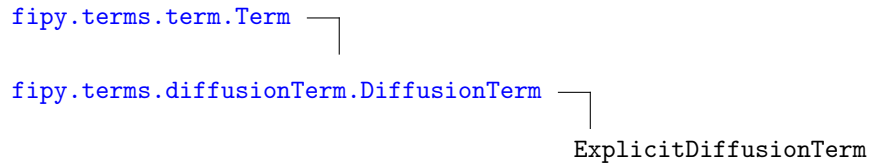
```
>>> -DiffusionTerm(coeff = [1.])
DiffusionTerm(coeff = [-1.0])
>>> -DiffusionTerm()
DiffusionTerm(coeff = [-1.0])
```

Overrides: `fiy.terms.term.Term.__neg__()`

Inherited from `Term`: `__add__`, `__eq__`, `__pos__`, `__radd__`, `__repr__`, `__rsub__`, `__sub__`, `cacheMatrix`, `cacheRHSvector`, `getMatrix`, `getRHSvector`, `justResidualVector`, `solve`, `sweep`

7.5 Module `fipy.terms.explicitDiffusionTerm`

Class `ExplicitDiffusionTerm`



Known Subclasses: `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.

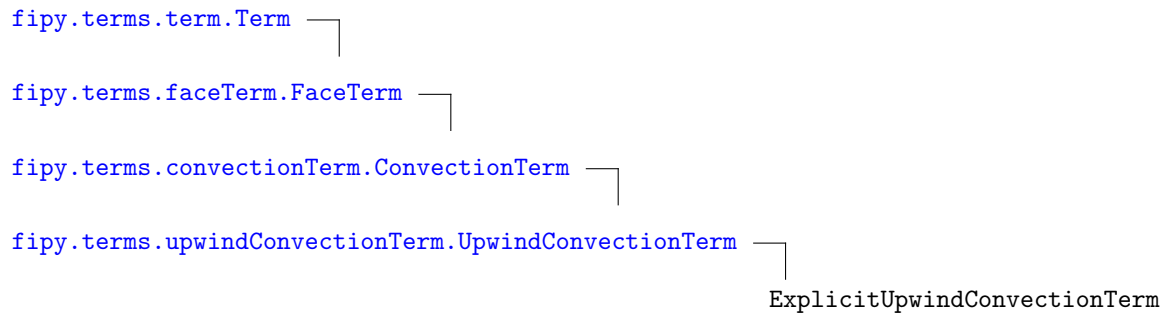
Methods

Inherited from `DiffusionTerm`: `__init__`, `__neg__`

Inherited from `Term`: `__add__`, `__eq__`, `__pos__`, `__radd__`, `__repr__`, `__rsub__`, `__sub__`, `cacheMatrix`, `cacheRHSvector`, `getMatrix`, `getRHSvector`, `justResidualVector`, `solve`, `sweep`

7.6 Module `fiPy.terms.explicitUpwindConvectionTerm`

Class `ExplicitUpwindConvectionTerm`



Known Subclasses: `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].

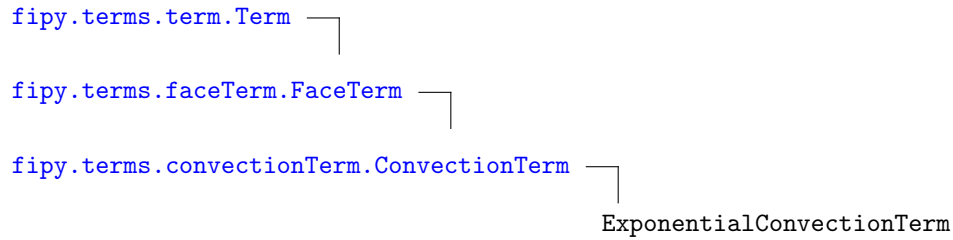
Methods

Inherited from `ConvectionTerm`: `__init__`, `__neg__`

Inherited from `Term`: `__add__`, `__eq__`, `__pos__`, `__radd__`, `__repr__`, `__rsub__`, `__sub__`, `cacheMatrix`, `cacheRHSvector`, `getMatrix`, `getRHSvector`, `justResidualVector`, `solve`, `sweep`

7.7 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 `ConvectionTerm`: `__init__`, `__neg__`

Inherited from `Term`: `__add__`, `__eq__`, `__pos__`, `__radd__`, `__repr__`, `__rsub__`, `__sub__`, `cacheMatrix`, `cacheRHSvector`, `getMatrix`, `getRHSvector`, `justResidualVector`, `solve`, `sweep`

7.8 Module `fiy.terms.faceTerm`

Class `FaceTerm`

`fiy.terms.term.Term` 
FaceTerm

Known Subclasses: `ConvectionTerm`

Attention!

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

Methods

`__init__(self, coeff=1.0)`

Create a Term.

Parameters

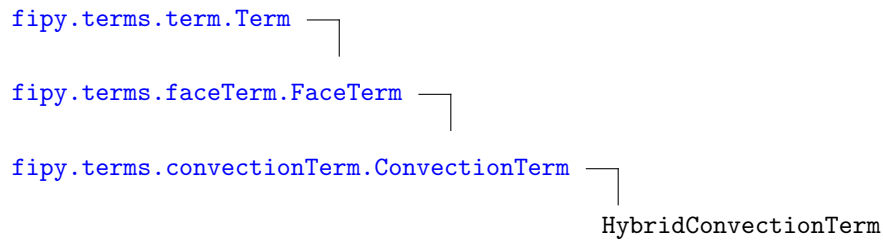
`coeff`: The coefficient for the term. A `CellVariable` or number. `FaceVariable` objects are also acceptable for diffusion or convection terms.

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

Inherited from Term: `__add__`, `__eq__`, `__neg__`, `__pos__`, `__radd__`, `__repr__`, `__rsub__`, `__sub__`, `cacheMatrix`, `cacheRHSvector`, `getMatrix`, `getRHSvector`, `justResidualVector`, `solve`, `sweep`

7.9 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].

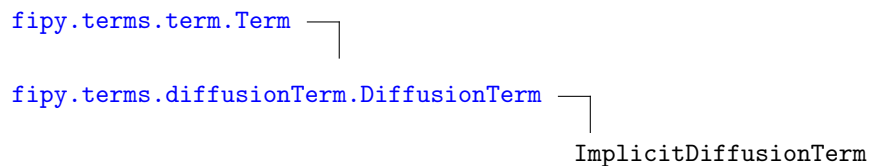
Methods

Inherited from `ConvectionTerm`: `__init__`, `__neg__`

Inherited from `Term`: `__add__`, `__eq__`, `__pos__`, `__radd__`, `__repr__`, `__rsub__`, `__sub__`, `cacheMatrix`, `cacheRHSvector`, `getMatrix`, `getRHSvector`, `justResidualVector`, `solve`, `sweep`

7.10 Module `fiPy.terms.implicitDiffusionTerm`

Class `ImplicitDiffusionTerm`



Known Subclasses: `NthOrderDiffusionTerm`

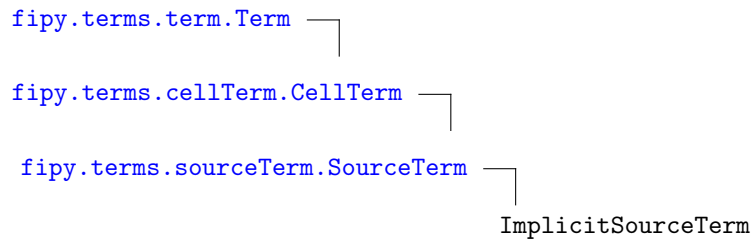
Methods

Inherited from `DiffusionTerm`: `__init__`, `__neg__`

Inherited from `Term`: `__add__`, `__eq__`, `__pos__`, `__radd__`, `__repr__`, `__rsub__`, `__sub__`, `cacheMatrix`, `cacheRHSvector`, `getMatrix`, `getRHSvector`, `justResidualVector`, `solve`, `sweep`

7.11 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 and in general should be negative to maintain stability.

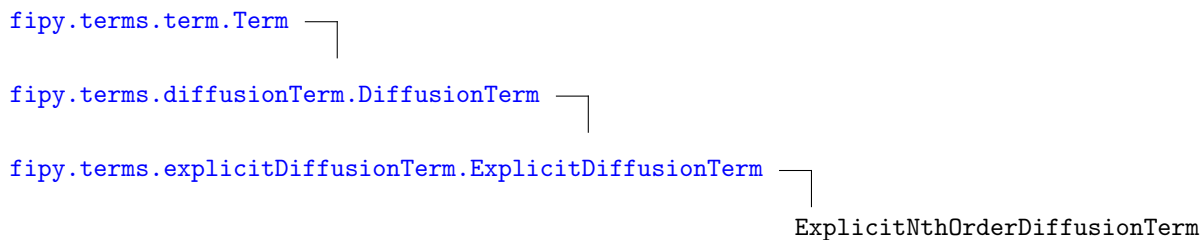
Methods

Inherited from `SourceTerm`: `__init__`

Inherited from `Term`: `__add__`, `__eq__`, `__neg__`, `__pos__`, `__radd__`, `__repr__`, `__rsub__`, `__sub__`, `cacheMatrix`, `cacheRHSvector`, `getMatrix`, `getRHSvector`, `justResidualVector`, `solve`, `sweep`

7.12 Module `fiy.terms.nthOrderDiffusionTerm`

Class `ExplicitNthOrderDiffusionTerm`



Methods

`__init__(self, coeff)`

Attention!

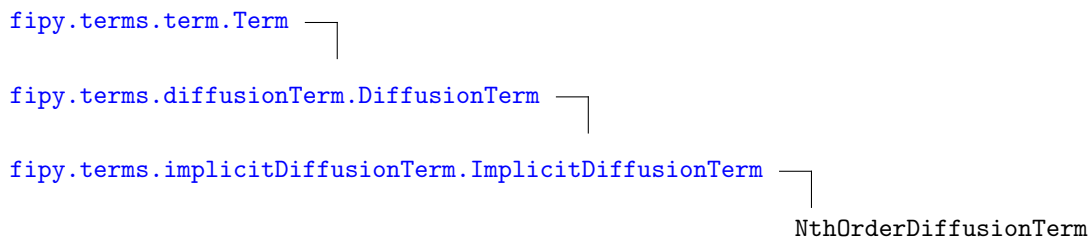
This class is deprecated. Use `ExplicitDiffusionTerm` instead.

Overrides: `fiy.terms.diffusionTerm.DiffusionTerm.__init__()`

Inherited from `DiffusionTerm`: `__neg__`

Inherited from `Term`: `__add__`, `__eq__`, `__pos__`, `__radd__`, `__repr__`, `__rsub__`, `__sub__`, `cacheMatrix`, `cacheRHSvector`, `getMatrix`, `getRHSvector`, `justResidualVector`, `solve`, `sweep`

Class `NthOrderDiffusionTerm`



Methods

`__init__(self, coeff)`

Attention!

This class is deprecated. Use `ImplicitDiffusionTerm` instead.

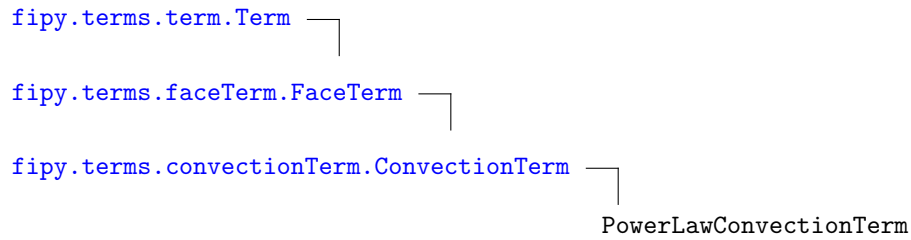
Overrides: `fiy.terms.diffusionTerm.DiffusionTerm.__init__()`

Inherited from `DiffusionTerm`: `__neg__`

Inherited from Term: `--add--`, `--eq--`, `--pos--`, `--radd--`, `--repr--`, `--rsub--`, `--sub--`, `cacheMatrix`,
`cacheRHSvector`, `getMatrix`, `getRHSvector`, `justResidualVector`, `solve`, `sweep`

7.13 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].

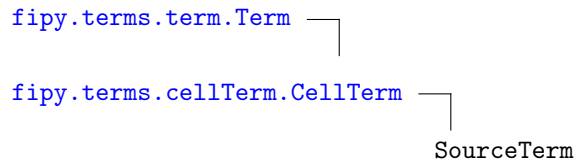
Methods

Inherited from `ConvectionTerm`: `__init__`, `__neg__`

Inherited from `Term`: `__add__`, `__eq__`, `__pos__`, `__radd__`, `__repr__`, `__rsub__`, `__sub__`, `cacheMatrix`, `cacheRHSvector`, `getMatrix`, `getRHSvector`, `justResidualVector`, `solve`, `sweep`

7.14 Module `fipy.terms.sourceTerm`

Class `SourceTerm`



Known Subclasses: `ImplicitSourceTerm`, `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.cellTerm.CellTerm.__init__()` (*inherited documentation*)

Inherited from Term: `__add__`, `__eq__`, `__neg__`, `__pos__`, `__radd__`, `__repr__`, `__rsub__`, `__sub__`, `cacheMatrix`, `cacheRHSvector`, `getMatrix`, `getRHSvector`, `justResidualVector`, `solve`, `sweep`

7.15 Module `fipy.terms.term`

Class `Term`

Known Subclasses: `CellTerm`, `DiffusionTerm`, `FaceTerm`, `_AdvectionTerm`, `_BinaryTerm`

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.

`__add__(self, other)`

Add a `Term` to another `Term`, number or variable.

```
>>> Term(coeff = 1.) + 10.
(Term(coeff = 1.0) + _ExplicitSourceTerm(coeff = 10.0))
>>> Term(coeff = 1.) + Term(coeff = 2.)
(Term(coeff = 1.0) + Term(coeff = 2.0))
```

`__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) == Term(coeff = 2.0))
```

it is equivalent to,

```
>>> Term(coeff = 1.) - Term(coeff = 2.)
(Term(coeff = 1.0) - Term(coeff = 2.0))
```

A `Term` should equate with a float.

Attention!

This does not work due to sign difficulties.

```
>>> Term(coeff = 1.) == 1.
False
```

Likewise for integers.

```

>>> Term(coeff = 1.) == 1
False
__neg__(self)

```

Negate a Term.

```

>>> -Term(coeff = 1.)
Term(coeff = -1.0)
__pos__(self)

```

Posate a Term.

```

>>> +Term(coeff = 1.)
Term(coeff = 1.0)
__radd__(self, other)

```

Add a Term to another Term, number or variable.

```

>>> Term(coeff = 1.) + 10.
(Term(coeff = 1.0) + _ExplicitSourceTerm(coeff = 10.0))
>>> Term(coeff = 1.) + Term(coeff = 2.)
(Term(coeff = 1.0) + Term(coeff = 2.0))
__repr__(self)

```

The representation of a Term object is given by,

```

>>> print Term(123.456)
Term(coeff = 123.456)
__rsub__(self, other)

```

Subtract a Term, number or variable from a Term.

```

>>> 10. - Term(coeff = 1.)
(_ExplicitSourceTerm(coeff = 10.0) - Term(coeff = 1.0))
__sub__(self, other)

```

Subtract a Term from a Term, number or variable.

```

>>> Term(coeff = 1.) - 10.
(Term(coeff = 1.0) - _ExplicitSourceTerm(coeff = 10.0))
>>> Term(coeff = 1.) - Term(coeff = 2.)
(Term(coeff = 1.0) - Term(coeff = 2.0))

```

`cacheMatrix(self)`

Informs `solve()` and `sweep()` to cache their matrix so that `getMatrix()` can return the matrix.

`cacheRHSvector(self)`

Informs `solve()` and `sweep()` to cache their right hand side vector so that `getRHSvector()` can return it.

`getMatrix(self)`

Return the matrix calculated in `solve()` or `sweep()`. The `cacheMatrix()` method should be called before `solve()` or `sweep()` to cache the matrix.

`getRHSvector(self)`

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

`justResidualVector(self, var, solver=None, boundaryConditions=(), dt=1.0, underRelaxation=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`.

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

`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`.

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`.

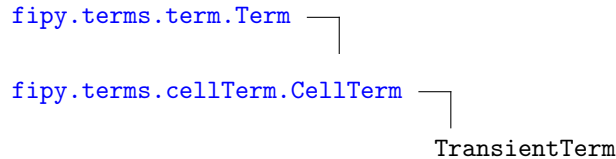
boundaryConditions: A tuple of boundaryConditions.

dt: The time step size.

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

7.16 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
```

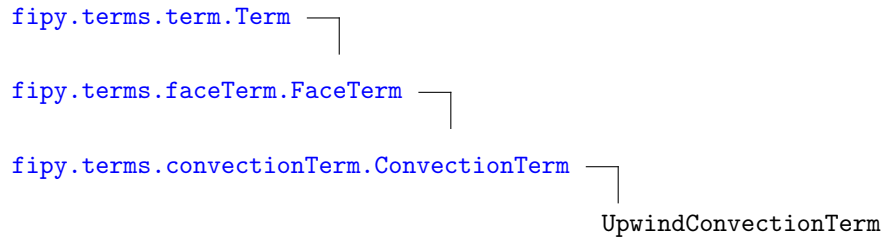
Methods

Inherited from `CellTerm`: `__init__`

Inherited from `Term`: `__add__`, `__eq__`, `__neg__`, `__pos__`, `__radd__`, `__repr__`, `__rsub__`, `__sub__`, `cacheMatrix`, `cacheRHSvector`, `getMatrix`, `getRHSvector`, `justResidualVector`, `solve`, `sweep`

7.17 Module `fiPy.terms.upwindConvectionTerm`

Class `UpwindConvectionTerm`



Known Subclasses: `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].

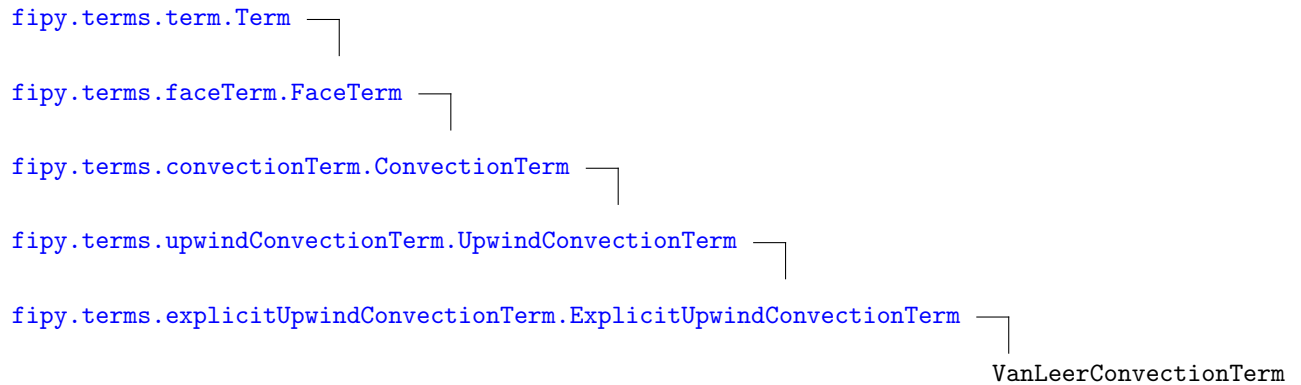
Methods

Inherited from `ConvectionTerm`: `__init__`, `__neg__`

Inherited from `Term`: `__add__`, `__eq__`, `__pos__`, `__radd__`, `__repr__`, `__rsub__`, `__sub__`, `cacheMatrix`, `cacheRHSvector`, `getMatrix`, `getRHSvector`, `justResidualVector`, `solve`, `sweep`

7.18 Module `fipy.terms.vanLeerConvectionTerm`

Class `VanLeerConvectionTerm`



Methods

Inherited from `ConvectionTerm`: `__init__`, `__neg__`

Inherited from `Term`: `__add__`, `__eq__`, `__pos__`, `__radd__`, `__repr__`, `__rsub__`, `__sub__`, `cacheMatrix`, `cacheRHSvector`, `getMatrix`, `getRHSvector`, `justResidualVector`, `solve`, `sweep`

Chapter 8

Package `fipy.tools`

8.1 Module `fipy.tools.dimensions.physicalField`

Physical quantities with units.

This module derives from [Konrad Hinsén'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
```

Class `PhysicalField`

Known Subclasses: `_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 Hinsén's](#) original `PhysicalQuantity`). The value can be a `Numeric` array:

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

or a tuple:

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

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

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

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

`__abs__(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
```

`__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
```

`__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 conversion fails.

```
>>> numerix.array(PhysicalField(((2.,3.), (4.,5.)), "m"))
Traceback (most recent call last):
...
TypeError: Numeric array value must be dimensionless
```

`__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 numerix.array(PhysicalField(1., 'inch')
...                       / PhysicalField(1., 'mm'))
25.4
>>> print numerix.array(PhysicalField(1., 'inch')
...                       / PhysicalField(1., 'kg'))
Traceback (most recent call last):
...
TypeError: Numeric array value must be dimensionless
```

`__eq__(self, other)`

`__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
```

`__ge__(self, other)`

`__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
```

`__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 a > PhysicalField("13 ft")
[[False True]
 [True True]]
```

Appropriately formatted dimensional quantity strings can also be compared.

```
>>> print a > "13 ft"
[[False True]
 [True True]]
```

Arrays are compared element to element

```
>>> print a > PhysicalField(((3.,13.), (17.,6.)), "ft")
[[True True]
 [False 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
```

```
__le__(self, other)
```

```
__len__(self)
```

```
__lt__(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
```

```
__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 numerix.array(PhysicalField(10., 's') * PhysicalField(2., 'Hz'))
20.0
```

```
>>> print numerix.array(PhysicalField(10., 's') * PhysicalField(2., 's'))
Traceback (most recent call last):
```

```
...
```

```
TypeError: Numeric array value must be dimensionless
```

```
__ne__(self, other)
```

`__neg__(self)`

Return the negative of the quantity. The unit is unchanged.

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

`__nonzero__(self)`

Test if the quantity is zero.

Should this only pass if the unit offset is zero?

`__pos__(self)`

`__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
```

`__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
```

`__rdiv__(self, other)`

`__repr__(self)`

Return representation of a physical quantity suitable for re-use

```
>>> PhysicalField(value = 3., unit = "eV")
PhysicalField(3.0,'eV')
```

`__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 numerix.array(PhysicalField(10., 's') * PhysicalField(2., 'Hz'))
20.0
>>> print numerix.array(PhysicalField(10., 's') * PhysicalField(2., 's'))
Traceback (most recent call last):
```

```
...
TypeError: Numeric array value must be dimensionless
```

```
__rpow__(self, other)
```

```
__rsub__(self, other)
```

```
__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
```

```
__str__(self)
```

Return human-readable form of a physical quantity

```
>>> print PhysicalField(value = 3., unit = "eV")
3.0 eV
```

```
__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
```

```
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:

$$| \text{self} - \text{other} | < \text{atol} + \text{rtol} * | \text{other} |$$

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

`allequal(self, other)`

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

`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: Numeric array value must be dimensionless
```

`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: Numeric array value must be dimensionless
```

`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: Numeric array value must be dimensionless
```

`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: Numeric array value must be dimensionless
```

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 dimensionless

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

```
...
```

```
TypeError: Numeric array value must be dimensionless
```

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()
(2.2+3j) ohm
```

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
```

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
```

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: Argument of cos must be an angle
```

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: Numeric array value must be dimensionless
```

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)
[ 17.  53.] m**2
```

floor(self)

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

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

getNumericValue(self)

Return the *PhysicalField* without units, after conversion to base SI units.

```
>>> print round(PhysicalField("1 inch").getNumericValue(), 6)
0.0254
```

getShape(self)

getUnit(self)

Return the unit object of *self*.

```
>>> PhysicalField("1 m").getUnit()
<PhysicalUnit m>
```

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
```

*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']
```

`isCompatible(self, unit)`

`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: Numeric array value must be dimensionless
```

`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: Numeric array value must be dimensionless
```

`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
```

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
```

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: Argument of sin must be an angle
```

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: Numeric array value must be dimensionless
```

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
```

`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
```

`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
```

`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: Argument of tan must be an angle
```

`tanh(self)`

Return the hyperbolic tangent of the `PhysicalField`

```
>>> PhysicalField(1.).tanh()
0.761594155956
```

The units of the `PhysicalField` must be dimensionless

```
>>> PhysicalField(60., "m").tanh()
Traceback (most recent call last):
...
TypeError: Numeric array value must be dimensionless
```

`tostring(self, max_line_width=None, precision=None, suppress_small=None, separator=' ')`

Return human-readable form of a physical quantity

```
>>> p = PhysicalField(value = (3., 3.14159), unit = "eV")
>>> print p.toString(precision = 3, separator = '|')
[ 3.   | 3.142] eV
```

Class `PhysicalUnit`

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.

`__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
```

`__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>
```

`__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>
```

`__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>
```

`__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>
```

`__repr__(self)`

Return representation of a physical unit

```
>>> PhysicalUnit('m', 1., [1,0,0,0,0,0,0,0,0])
<PhysicalUnit m>
```

`__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>

```

`__str__(self)`

Return representation of a physical unit

```

>>> PhysicalUnit('m', 1., [1,0,0,0,0,0,0,0,0])
<PhysicalUnit m>

```

`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']

```

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

```

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 a.getUnit().isCompatible(b.getUnit())
[True True True True True True True]
>>> c = PhysicalField("1. K")
>>> print a.getUnit().isCompatible(c.getUnit())
[False True True True False 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

```

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
```

`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'
```

`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>
```

8.2 Module `fiPy.tools.dump`

Functions

`read(filename, fileobject=None)`

Read a pickled object from a file. Returns the unpickled object. Wrapper for `cPickle.load()`.

Parameters

filename: The name of the file to unpickle the object from.

fileobject: Used to remove temporary files

`write(data, filename=None, extension='')`

Pickle an object and write it to a file. Wrapper for `cPickle.dump()`.

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

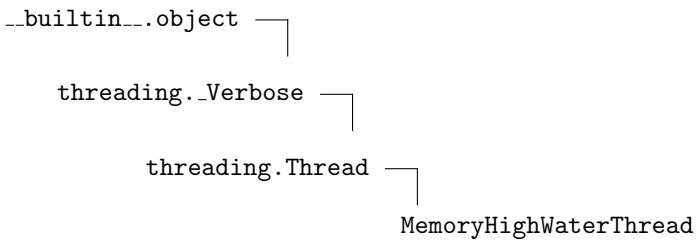
data: The object to be pickled.

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.

8.3 Module `fipy.tools.memoryLogger`

Class `MemoryHighWaterThread`



Methods

`__init__(self, pid, sampleTime=1)`

Overrides: `threading.Thread.__init__()`
`run(self)`

Overrides: `threading.Thread.run()`
`stop(self)`

Inherited from object: `__delattr__`, `__getattr__`, `__hash__`, `__new__`, `__reduce__`, `__reduce_ex__`, `__setattr__`, `__str__`

Inherited from Thread: `__repr__`, `getName`, `isAlive`, `isDaemon`, `join`, `setDaemon`, `setName`, `start`

Class `MemoryLogger`

Methods

`__init__(self, sampleTime=1)`

`__del__(self)`

`start(self)`

`stop(self)`

8.4 Module `fipy.tools.numerix`

The functions provided in this module replace the `Numeric` module. The functions work with `Variables`, arrays or numbers. For example, create a `Variable`.

```
>>> from fipy.variables.variable import Variable
>>> var = Variable(value = 0)
```

Take the tangent of such a variable. The returned value is itself a `Variable`.

```
>>> v = tan(var)
>>> v
numerix.tan(Variable(value = array(0)))
>>> print float(v)
0.0
```

Take the tangent of a int.

```
>>> tan(0)
0.0
```

Take the tangent of an array.

```
>>> print tan(array((0,0,0)))
[ 0.  0.  0.]
```

Eventually, this module will be the only place in the code where `Numeric` (or `numarray` (or `scipy_core`)) is explicitly imported.

Functions

`allclose(first, second, rtol=1.0000000000000001e-05, atol=1e-08)`

Tests whether or not `first` and `second` are equal, subject to the given relative and absolute tolerances, such that:

$$| \text{first} - \text{second} | < \text{atol} + \text{rtol} * | \text{second} |$$

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

`allequal(first, second)`

Returns `true` if every element of `first` is equal to the corresponding element of `second`.

`arccos(arr)`

Inverse cosine of x , $\cos^{-1} x$

```
>>> print tostring(arccos(0.0), precision = 3)
1.571
```

If SciPy has been loaded, the next test will return `NaN`, otherwise it will generate `OverflowError: math range error`


```

>>> try:
...     print str(arccos(2.0)) == "nan"
... except (OverflowError, ValueError):
...     print 1
1
>>> 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

```

If SciPy has been loaded, the next test will return NaN, otherwise it will generate `OverflowError: math range error`

```

>>> try:
...     print str(arccosh(0.0)) == "nan"
... except (OverflowError, ValueError):
...     print 1
1
>>> 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

```

If SciPy has been loaded, the next test will return NaN, otherwise it will generate `OverflowError: math range error`

```

>>> try:
...     print str(arcsin(2.0)) == "nan"
... except (OverflowError, ValueError):
...     print 1
1

```

```
>>> print tostring(arcsin(array((0,0.5,1.0))), precision = 3)
[ 0.    0.524  1.571]
>>> from fipy.variables.variable import Variable
>>> arcsin(Variable(value = (0,0.5,1.0)))
numerix.arcsin(Variable(value = array([ 0. ,  0.5,  1. ])))
```

Attention!

the next should really return radians, but doesn't

```
>>> print tostring(arcsin(Variable(value = (0,0.5,1.0))), precision = 3)
[ 0.    0.524  1.571]
```

`arcsinh(arr)`

Inverse hyperbolic sine of x , $\sinh^{-1} x$

```
>>> print tostring(arcsinh(1.0), precision = 3)
0.881
>>> print tostring(arcsinh(array((1,2,3))), precision = 3)
[ 0.881  1.444  1.818]
>>> from fipy.variables.variable import Variable
>>> arcsinh(Variable(value = (1,2,3)))
numerix.arcsinh(Variable(value = array([ 1.,  2.,  3.])))
>>> print tostring(arcsinh(Variable(value = (1,2,3))), precision = 3)
[ 0.881  1.444  1.818]
```

`arctan(arr)`

Inverse tangent of x , $\tan^{-1} x$

```
>>> print tostring(arctan(1.0), precision = 3)
0.785
>>> print tostring(arctan(array((0,0.5,1.0))), precision = 3)
[ 0.    0.464  0.785]
>>> from fipy.variables.variable import Variable
>>> arctan(Variable(value = (0,0.5,1.0)))
numerix.arctan(Variable(value = array([ 0. ,  0.5,  1. ])))
```

Attention!

the next should really return radians, but doesn't

```
>>> print tostring(arctan(Variable(value = (0,0.5,1.0))), precision = 3)
[ 0.    0.464  0.785]
```

`arctan2(arr, other)`

Inverse tangent of a ratio x/y , $\tan^{-1} \frac{x}{y}$

```
>>> print tostring(arctan2(3.0, 3.0), precision = 3)
0.785
>>> print tostring(arctan2(array((0, 1, 2)), 2), precision = 3)
[ 0.    0.464  0.785]
>>> from fipy.variables.variable import Variable
```

```
>>> arctan2(Variable(value = (0, 1, 2)), 2)
(numerix.arctan2(Variable(value = array([ 0., 1., 2.])), 2))
```

Attention!

the next should really return radians, but doesn't

```
>>> print toString(arctan2(Variable(value = (0, 1, 2)), 2), precision = 3)
[ 0.    0.464  0.785]
```

arctanh(arr)

Inverse hyperbolic tangent of x , $\tanh^{-1} x$

```
>>> print toString(arctanh(0.5), precision = 3)
0.549
>>> print toString(arctanh(array((0,0.25,0.5))), precision = 3)
[ 0.    0.255  0.549]
>>> from fipy.variables.variable import Variable
>>> arctanh(Variable(value = (0,0.25,0.5)))
numerix.arctanh(Variable(value = array([ 0. , 0.25, 0.5 ])))
>>> print toString(arctanh(Variable(value = (0,0.25,0.5))), precision = 3)
[ 0.    0.255  0.549]
```

ceil(arr)

The largest integer $\geq x$, $\lceil x \rceil$

```
>>> print ceil(2.3)
3.0
>>> print ceil(array((-1.5,2,2.5)))
[-1.  2.  3.]
>>> from fipy.variables.variable import Variable
>>> ceil(Variable(value = (-1.5,2,2.5), unit = "m**2"))
numerix.ceil(Variable(value = PhysicalField(array([-1.5, 2. , 2.5]),'m**2')))
>>> print ceil(Variable(value = (-1.5,2,2.5), unit = "m**2"))
[-1.  2.  3.] m**2
```

conjugate(arr)

Complex conjugate of $z = x + iy$, $z^* = x - iy$

```
>>> print conjugate(3 + 4j)
(3-4j)
>>> print allclose(conjugate(array((3 + 4j, -2j, 10))), (3 - 4j, 2j, 10))
1
>>> from fipy.variables.variable import Variable
>>> var = conjugate(Variable(value = (3 + 4j, -2j, 10), unit = "ohm"))
>>> print var.getUnit()
<PhysicalUnit ohm>
>>> print allclose(var.getNumericValue(), (3 - 4j, 2j, 10))
1
```

cos(arr)

Cosine of x , $\cos x$

```
>>> print toString(cos(2*pi/6), precision = 3)
0.5
>>> 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]
```

`crossProd(v1, v2)`

Vector cross-product of \vec{v}_1 and \vec{v}_2 , $\vec{v}_1 \times \vec{v}_2$

`dot(a1, a2, axis=1)`

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.vectorCellVariable import VectorCellVariable
>>> v1 = VectorCellVariable(mesh=mesh, value=((0,1),(1,2)))
>>> v2 = array(((0,1),(1,2)))
>>> dot(v1, v2)._getVariableClass()
<class 'fipy.variables.cellVariable.CellVariable'>
>>> dot(v2, v1)._getVariableClass()
<class 'fipy.variables.cellVariable.CellVariable'>
>>> print dot(v1, v2)
[ 1.  5.]
>>> dot(v1, v1)._getVariableClass()
<class 'fipy.variables.cellVariable.CellVariable'>
>>> print dot(v1, v1)
[ 1.  5.]
>>> type(dot(v2, v2))
```

```
<type 'numpy.ndarray'>
>>> print dot(v2, v2)
[1 5]
```

`exp(arr)`

Natural exponent of x , e^x

`floor(arr)`

The largest integer $\leq x$, $\lfloor x \rfloor$

```
>>> print floor(2.3)
2.0
>>> print floor(array((-1.5,2,2.5)))
[-2.  2.  2.]
>>> from fipy.variables.variable import Variable
>>> floor(Variable(value = (-1.5,2,2.5), unit = "m**2"))
numerix.floor(Variable(value = PhysicalField(array([-1.5,  2. ,  2.5]),'m**2'))))
>>> print floor(Variable(value = (-1.5,2,2.5), unit = "m**2"))
[-2.  2.  2.] m**2
```

`getShape(arr)`

Return the shape of `arr`

```
>>> getShape(1)
()
>>> getShape(1.)
()
>>> from fipy.variables.variable import Variable
>>> getShape(Variable(1))
()
>>> getShape(Variable(1.))
()
>>> getShape(Variable(1., unit = "m"))
()
>>> getShape(Variable("1 m"))
()
```

`getTypecode(arr)`

Returns the `getTypecode()` of the array or `Variable`. Also returns a meaningful typecode for ints and floats.

```
>>> getTypecode(1)
'l'
>>> getTypecode(1.)
'd'
>>> getTypecode(array(1))
'l'
>>> getTypecode(array(1.))
'd'
```

```
>>> from fipy.variables.variable import Variable
>>> getTypecode(Variable(1.))
'd'
>>> getTypecode(Variable(1))
'1'
>>> getTypecode([0])
Traceback (most recent call last):
...
TypeError: No typecode for object
```

`indices(dimensions, typecode=None)`

`indices(dimensions,typecode=None)` returns an array representing a grid of indices with row-only, and column-only variation.

```
>>> NUMERIX.allclose(NUMERIX.array(indices((4, 6))), NUMERIX.indices((4,6)))
1
>>> NUMERIX.allclose(NUMERIX.array(indices((4, 6, 2))), NUMERIX.indices((4, 6, 2)))
1
>>> NUMERIX.allclose(NUMERIX.array(indices((1,))), NUMERIX.indices((1,)))
1
>>> NUMERIX.allclose(NUMERIX.array(indices((5,))), NUMERIX.indices((5,)))
1
```

`isFloat(arr)`

`isInt(arr)`

`L1norm(arr)`

Parameters

arr: The array to evaluate.

Return Value

$\|\mathbf{arr}\|_1 = \sum_{j=1}^n |\mathbf{arr}_j|$ is the L^1 -norm of *arr*.

`L2norm(arr)`

Parameters

arr: The array to evaluate.

Return Value

$\|\text{arr}\|_2 = \sqrt{\sum_{j=1}^n |\text{arr}_j|^2}$ is the L^2 -norm of `arr`.

`LINFnorm(arr)`

Parameters

`arr`: The array to evaluate.

Return Value

$\frac{\|\text{arr}\|_\infty = [\sum_{j=1}^n |\text{arr}_j|^\infty]^\infty}{\max_j |\text{arr}_j|}$ is the L^∞ -norm of `arr`.

`log(arr)`

Natural logarithm of x , $\ln x \equiv \log_e x$

```
>>> print tostring(log(10), precision = 3)
2.303
>>> print tostring(log(array((0.1,1,10))), precision = 3)
[-2.303  0.      2.303]
>>> from fipy.variables.variable import Variable
>>> log(Variable(value = (0.1,1,10)))
numerix.log(Variable(value = array([ 0.1,  1. , 10. ])))
>>> print tostring(log(Variable(value = (0.1,1,10))), precision = 3)
[-2.303  0.      2.303]
```

`log10(arr)`

Base-10 logarithm of x , $\log_{10} x$

```
>>> print log10(10)
1.0
>>> print log10(array((0.1,1,10)))
[-1.  0.  1.]
>>> from fipy.variables.variable import Variable
>>> log10(Variable(value = (0.1,1,10)))
numerix.log10(Variable(value = array([ 0.1,  1. , 10. ])))
>>> print log10(Variable(value = (0.1,1,10)))
[-1.  0.  1.]
```

`max(arr)`

max function

```
>>> from fipy.tools.dimensions.physicalField import PhysicalField
>>> print max(PhysicalField(value = (0.1, -0.2, 0.3), unit = 'm'))
0.3 m
>>> print max(array((0.1, -0.2, 0.3)))
```

```

0.3
>>> from fipy.variables.variable import Variable
>>> print max(Variable(value = (0.1, -0.2, 0.3)))
0.3

```

min(arr)

min function

```

>>> from fipy.tools.dimensions.physicalField import PhysicalField
>>> print min(PhysicalField(value = (0.1, -0.2, 0.3), unit = 'm'))
-0.2 m
>>> print min(array((0.1, -0.2, 0.3)))
-0.2
>>> from fipy.variables.variable import Variable
>>> print min(Variable((0.1, -0.2, 0.3)))
-0.2

```

nonzero(a)

ones(a, t='1')

put(arr, ids, values)

The opposite of *take*. The values of *arr* at the locations specified by *ids* are set to the corresponding value of *values*.

The following is to test improvements to puts with masked arrays. Places in the code were assuming incorrect put behavior.

```

>>> maskValue = 999999
>>> arr = zeros(3, '1')
>>> ids = MA.masked_values((2, maskValue), maskValue)
>>> values = MA.masked_values((4, maskValue), maskValue)
>>> put(arr, ids, values) ## this should work
Traceback (most recent call last):
...
IndexError: index out of range for array
>>> 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 lengths of all the axes is constant (the total number of elements does not change).

`sin(arr)`

Sine of x , $\sin x$

```

>>> print sin(pi/6)
0.5
>>> print sin(array((0,pi/6,pi/2)))
[ 0.  0.5  1. ]
>>> from fipy.variables.variable import Variable
>>> sin(Variable(value = (0,pi/6,pi/2), unit = "rad"))
numerix.sin(Variable(value = PhysicalField(array([ 0.          ,  0.52359878,  1.57079633]),'rad'))
>>> print sin(Variable(value = (0,pi/6,pi/2), unit = "rad"))
[ 0.  0.5  1. ]

```

`sinh(arr)`

Hyperbolic sine of x , $\sinh x$

```

>>> print sinh(0)
0.0
>>> print tostring(sinh(array((0,1,2))), precision = 3)
[ 0.  1.175  3.627]
>>> from fipy.variables.variable import Variable
>>> sinh(Variable(value = (0,1,2)))
numerix.sinh(Variable(value = array([ 0.,  1.,  2.])))
>>> print tostring(sinh(Variable(value = (0,1,2))), precision = 3)
[ 0.  1.175  3.627]

```

`sqrt(arr)`

Square root of x , \sqrt{x}

```

>>> print tostring(sqrt(2), precision = 3)
1.414
>>> print tostring(sqrt(array((1,2,3))), precision = 3)
[ 1.  1.414  1.732]
>>> from fipy.variables.variable import Variable
>>> sqrt(Variable(value = (1, 2, 3), unit = "m**2"))
numerix.sqrt(Variable(value = PhysicalField(array([ 1.,  2.,  3.]),'m**2')))
>>> print tostring(sqrt(Variable(value = (1, 2, 3), unit = "m**2")), precision = 3)
[ 1.  1.414  1.732] m

```

`sqrtDot(a1, a2)`

Return array of square roots of vector dot-products for arrays `a1` and `a2` of vectors `v1` and `v2`

Usually used with `v1==v2` to return magnitude of `v1`.

`sum(arr, axis=0)`

The sum of all the elements of `arr` along the specified axis.

`take(a, indices, axis=0, fill_value=None)`

Selects the elements of `a` corresponding to `indices`.

`tan(arr)`

Tangent of x , $\tan x$

```
>>> print toString(tan(pi/3), precision = 3)
1.732
>>> print toString(tan(array((0,pi/3,2*pi/3))), precision = 3)
[ 0.      1.732 -1.732]
>>> from fipy.variables.variable import Variable
>>> tan(Variable(value = (0,pi/3,2*pi/3), unit = "rad"))
numerix.tan(Variable(value = PhysicalField(array([ 0.      , 1.04719755,  2.0943951 ]), 'rad'))
>>> print toString(tan(Variable(value = (0,pi/3,2*pi/3), unit = "rad")), precision = 3)
[ 0.      1.732 -1.732]
```

`tanh(arr)`

Hyperbolic tangent of x , $\tanh x$

```
>>> print toString(tanh(1), precision = 3)
0.762
>>> print toString(tanh(array((0,1,2))), precision = 3)
[ 0.      0.762  0.964]
>>> from fipy.variables.variable import Variable
>>> tanh(Variable(value = (0,1,2)))
numerix.tanh(Variable(value = array([ 0.,  1.,  2.])))
>>> print toString(tanh(Variable(value = (0,1,2))), precision = 3)
[ 0.      0.762  0.964]
```

`toString(arr, max_line_width=None, precision=None, suppress_small=None, separator=' ', array_output=0)`

Returns a textual representation of a number or field of numbers. Each dimension is indicated by a pair of matching square brackets (`[]`), within which each subset of the field is output. The orientation of the dimensions is as follows: the last (rightmost) dimension is always horizontal, so that the frequent rank-1 fields use a minimum of screen real-estate. The next-to-last dimension is displayed vertically if present and any earlier dimension is displayed with additional bracket divisions.

Parameters

max_line_width: the maximum number of characters used in a single line. Default is `sys.output_line_width` or 77.

precision: the number of digits after the decimal point. Default is `sys.float_output_precision` or 8.

suppress_small: whether small values should be suppressed (and output as 0). Default is `sys.float_output_suppress_small` or `false`.

separator: what character string to place between two numbers.

array_output: Format output for an `eval`. Only used if `arr` is a `Numeric array`.

```
>>> from fipy import Variable
>>> print toString(Variable((1,0,11.2345)), precision=1)
[ 1.  0. 11.2]
>>> print toString(array((1,2)), precision=5)
[1 2]
>>> print toString(array((1.12345,2.79)), precision=2)
[ 1.12  2.79]
>>> print toString(1)
1
>>> print toString(array(1))
1
>>> print toString(array([1.23345]), precision=2)
[ 1.23]
>>> print toString(array([1]), precision=2)
[1]
>>> print toString(1.123456, precision=2)
1.12
>>> print toString(array(1.123456), precision=3)
1.123
```

`zeros(a, t='l')`

8.5 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.

8.6 Module *fipy.tools.vector*

Vector utility functions that are inexplicably absent from Numeric

Functions

`crossProd(v1, v2)`

Return vector cross-product of v1 and v2.

`prune(array, shift, start=0)`

removes elements with indices $i = \text{start} + \text{shift} * n$ where $n = 0, 1, 2, \dots$

```
>>> prune(numerix.arange(10), 3, 5)
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> prune(numerix.arange(10), 3, 2)
array([0, 1, 3, 4, 6, 7, 9])
>>> prune(numerix.arange(10), 3)
array([1, 2, 4, 5, 7, 8])
>>> prune(numerix.arange(4, 7), 3)
array([5, 6])
```

`putAdd(vector, ids, addition Vector)`

This is a temporary replacement for Numeric.put as it was not doing what we thought it was doing.

`sqrtDot(v1, v2)`

Return square root of vector dot-product of v1 and v2.

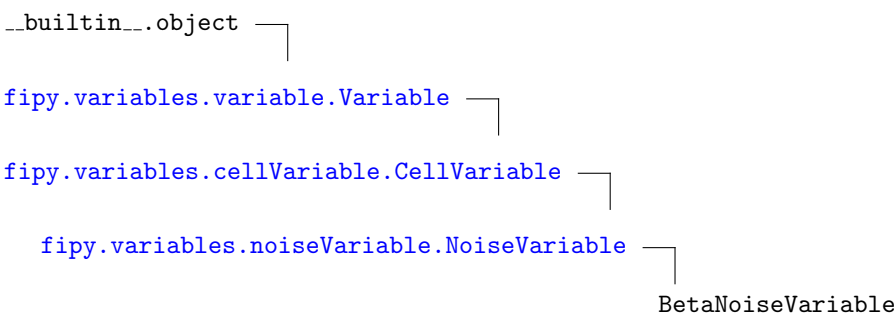
Usually used with `v1==v2` to return magnitude of v1.

Chapter 9

Package `fipy.variables`

9.1 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]

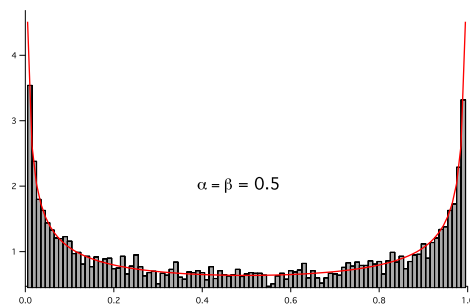
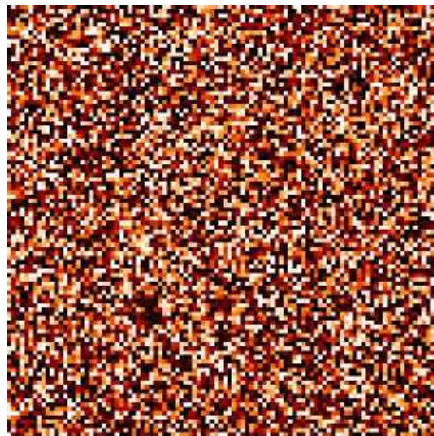
>>> if __name__ == '__main__':
...     from fipy import viewers
...     viewer = viewers.make(vars = noise, limits = {'datamin': 0, 'datamax': 1})
...     histoplot = viewers.make(vars = (histogram, betadist), limits = {'datamin': 0, 'datamax': 1})

>>> 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)`

Parameters

mesh: The mesh on which to define the noise.

alpha: The parameter α .

beta: The parameter β .

Overrides: [fipy.variables.noiseVariable.NoiseVariable.__init__\(\)](#)

Inherited from object: [__delattr__](#), [__getattr__](#), [__hash__](#), [__reduce__](#), [__reduce_ex__](#), [__setattr__](#)

Inherited from [CellVariable](#): [__call__](#), [__getstate__](#), [__setstate__](#), [getArithmeticFaceValue](#), [getCellVolumeAverage](#), [getFaceGrad](#), [getFaceGradAverage](#), [getFaceValue](#), [getGaussGrad](#), [getGrad](#), [getHarmonicFaceValue](#), [getLeastSquaresGrad](#), [getOld](#), [getValue](#), [setValue](#), [updateOld](#)

Inherited from [NoiseVariable](#): [copy](#), [scramble](#)

Inherited from [Variable](#): [__abs__](#), [__add__](#), [__and__](#), [__array__](#), [__array_wrap__](#), [__div__](#), [__eq__](#), [__float__](#), [__ge__](#), [__getitem__](#), [__gt__](#), [__le__](#), [__len__](#), [__lt__](#), [__mod__](#), [__mul__](#), [__ne__](#), [__neg__](#), [__or__](#), [__pos__](#), [__pow__](#), [__radd__](#), [__rdiv__](#), [__repr__](#), [__rmul__](#), [__rpow__](#), [__rsub__](#), [__setitem__](#), [__str__](#), [__sub__](#), [allclose](#), [allegual](#), [arccos](#), [arccosh](#), [arcsin](#), [arcsinh](#), [arctan](#), [arctan2](#), [arctanh](#), [cacheMe](#), [ceil](#), [conjugate](#), [cos](#), [cosh](#), [dontCacheMe](#), [dot](#), [exp](#), [floor](#), [getMag](#), [getMesh](#), [getName](#), [getNumericValue](#), [getShape](#), [getSubscribedVariables](#), [getTypecode](#), [getUnit](#), [inBaseUnits](#), [inUnitsOf](#), [log](#), [log10](#), [put](#), [reshape](#), [setName](#), [sin](#), [sinh](#), [sqrt](#), [sum](#), [take](#), [tan](#), [tanh](#), [tostring](#), [transpose](#)

Static Methods

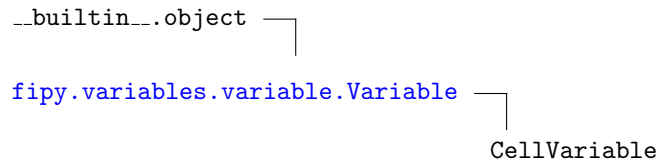
Inherited from [Variable](#): [__new__](#)

Class Variables

Name	Description
Inherited from Variable : __array_priority__	

9.2 Module `fipy.variables.cellVariable`

Class `CellVariable`



Known Subclasses: `DistanceVariable`, `HistogramVariable`, `ModularVariable`, `NoiseVariable`, `SurfactantVariable`, `_AddOverFacesVariable`, `_AdsorptionCoeff`, `_AdsorptionCoeff`, `_InterfaceSurfactantVariable`, `_MaxCoeff`, `_MetalIonSourceVariable`, `_RefinedMeshCellVariable`, `_ReMeshedCellVariable`

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')
>>> var.setValue(mesh.getCellCenters()[...,0] * mesh.getCellCenters()[...,1])

>>> 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, unit=None, hasOld=0)
```

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

```
__call__(self, point=None)
```

“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

```

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

`__getstate__(self)`

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

`__setstate__(self, dict)`

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

`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*)

`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
>>> mesh = Grid1D(dx = (1., 1.))
>>> var = CellVariable(mesh = mesh, value = (1, 2))
>>> faceValue = var.getArithmeticFaceValue()[mesh.getInteriorFaces()[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()[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()[0]]
>>> answer = (var[0] - var[1]) * (5.0 / 55.0) + var[1]
>>> numerix.allclose(faceValue, answer, atol = 1e-10, rtol = 1e-10)
1

```

`getCellVolumeAverage(self)`

Return the cell-volume-weighted average of the `CellVariable`:

$$\langle \phi \rangle_{\text{vol}} = \frac{\sum_{\text{cells}} \phi_{\text{cell}} V_{\text{cell}}}{\sum_{\text{cells}} V_{\text{cell}}}$$

```

>>> from fipy.meshes.grid2D import Grid2D
>>> mesh = Grid2D(nx = 3, ny = 1, dx = .5, dy = .1)
>>> var = CellVariable(value = (1, 2, 6), mesh = mesh)
>>> print var.getCellVolumeAverage()
3.0

```

`getFaceGrad(self)`

Return $\nabla\phi$ as a `VectorFaceVariable` using differencing for the normal direction(second-order gradient).

`getFaceGradAverage(self)`

Return $\nabla\phi$ as a `VectorFaceVariable` using averaging for the normal direction(second-order gradient)

`getFaceValue(self, func=None, **args)`

Returns a `FaceVariable` whose value is evaluated by the function provided.

```

>>> from fipy.meshes.grid1D import Grid1D
>>> mesh = Grid1D(nx=2)
>>> var = CellVariable(value=(0, 1), mesh=mesh)
>>> print var.getFaceValue()
[ 0.  0.5  1. ]
>>> print var.getFaceValue(lambda a, b: numerix.minimum(a,b))
[ 0.  0.  1.]

```

`getGaussGrad(self)`

Return $\frac{1}{V_P} \sum_f \vec{n}_f \phi_f A_f$ as a `VectorCellVariable` (first-order gradient).

`getGrad(self)`

Return $\nabla\phi$ as a `VectorCellVariable` (first-order gradient).

`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
>>> mesh = Grid1D(dx = (1., 1.))
>>> var = CellVariable(mesh = mesh, value = (1, 2))
>>> # faceValue = var.getHarmonicFaceValue()[mesh.getInteriorFaces()[0]]
>>> faceValue = var.getHarmonicFaceValue()[mesh.getInteriorFaces()]
>>> 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()[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()[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
```

getLeastSquaresGrad(*self*)

Return $\nabla\phi$, which is determined by solving for $\nabla\phi$ in the following matrix equation,

$$\nabla\phi \cdot \sum_f d_{AP}^2 \vec{n}_{AP} \otimes \vec{n}_{AP} = \sum_f d_{AP}^2 (\vec{n} \cdot \nabla\phi)_{AP}$$

The matrix equation is derived by minimizing the following least squares sum,

$$F(\phi_x, \phi_y) = \sqrt{\sum_f (d_{AP} \vec{n}_{AP} \cdot \nabla\phi - d_{AP} (\vec{n}_{AP} \cdot \nabla\phi)_{AP})^2}$$

Tests

```
>>> from fipy import Grid2D
>>> print CellVariable(mesh=Grid2D(nx=2, ny=2, dx=0.1, dy=2.0), value=(0,1,3,6)).getLeastSquaresGrad()
[[8.0 1.2]
 [8.0 2.0]
 [24.0 1.2]
 [24.0 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]]
```

`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.]
```

`getValue(self, points=(), cells=())`

“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
```

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

`setValue(self, value, cells=(), unit=None, where=None)`

Patched values can be set by using either `cells` or `where`.

```
>>> from fipy.meshes.grid1D import Grid1D
>>> mesh = Grid1D(nx = 4)
>>> v1 = CellVariable(value=(4,7,2,6), mesh=mesh)
>>> print v1
[ 4.  7.  2.  6.]
>>> v1.setValue(4, where=(0, 0, 1, 1))
>>> print v1
```

```

[ 4.  7.  4.  4.]
>>> v1.setValue((5,2,7,8), where=(0, 1, 1, 1))
>>> print v1
[ 4.  2.  7.  8.]
>>> v1.setValue(3, unit = 'm')
>>> print v1
[ 3.  3.  3.  3.] m
>>> import warnings
>>> warnings.filterwarnings("ignore", "'where' should be used instead of 'cells'", DeprecationWarning)
>>> v1.setValue(1, cells=mesh.getCells()[2:])
>>> warnings.resetwarnings()
>>> print v1
[ 3.  3.  1.  1.] m
>>> v1.setValue(4)
>>> print v1
[ 4.  4.  4.  4.]

```

Overrides: [fipy.variables.variable.Variable.setValue\(\)](#)

`updateOld(self)`

Set the values of the previous solution sweep to the current values.

Inherited from object: `__delattr__`, `__getattr__`, `__hash__`, `__reduce__`, `__reduce_ex__`, `__setattr__`

Inherited from Variable: `__abs__`, `__add__`, `__and__`, `__array__`, `__array_wrap__`, `__div__`, `__eq__`, `__float__`, `__ge__`, `__getitem__`, `__gt__`, `__le__`, `__len__`, `__lt__`, `__mod__`, `__mul__`, `__ne__`, `__neg__`, `__or__`, `__pos__`, `__pow__`, `__radd__`, `__rdiv__`, `__repr__`, `__rmul__`, `__rpow__`, `__rsub__`, `__setitem__`, `__str__`, `__sub__`, `allclose`, `allequal`, `arccos`, `arccosh`, `arcsin`, `arcsinh`, `arctan`, `arctan2`, `arctanh`, `cacheMe`, `ceil`, `conjugate`, `cos`, `cosh`, `dontCacheMe`, `dot`, `exp`, `floor`, `getMag`, `getMesh`, `getName`, `getNumericValue`, `getShape`, `getSubscribedVariables`, `getTypecode`, `getUnit`, `inBaseUnits`, `inUnitsOf`, `log`, `log10`, `put`, `reshape`, `setName`, `sin`, `sinh`, `sqrt`, `sum`, `take`, `tan`, `tanh`, `tostring`, `transpose`

Static Methods

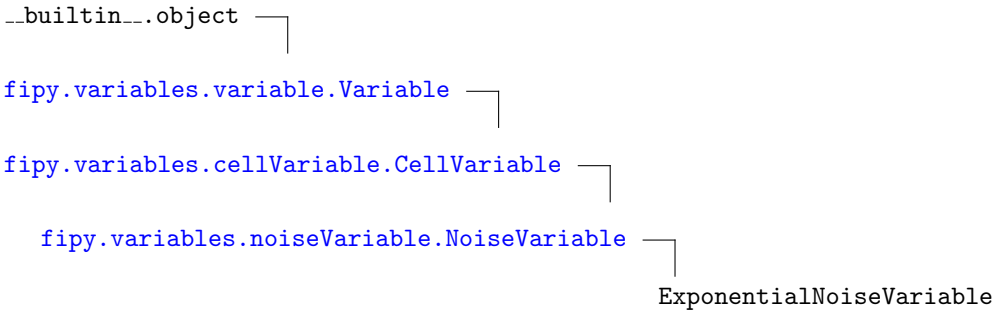
Inherited from Variable: `__new__`

Class Variables

Name	Description
Inherited from Variable: <code>__array_priority__</code>	

9.3 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 viewers
...     viewer = viewers.make(vars = noise, limits = {'datamin': 0, 'datamax': 5})
...     histoplot = viewers.make(vars = (histogram, expdist), limits = {'datamin': 0, 'datamax': 1.

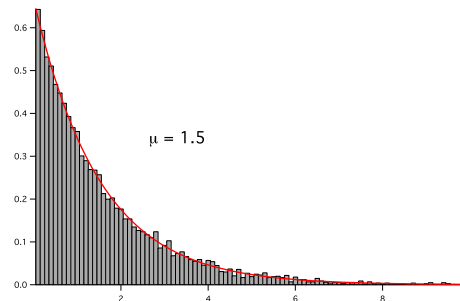
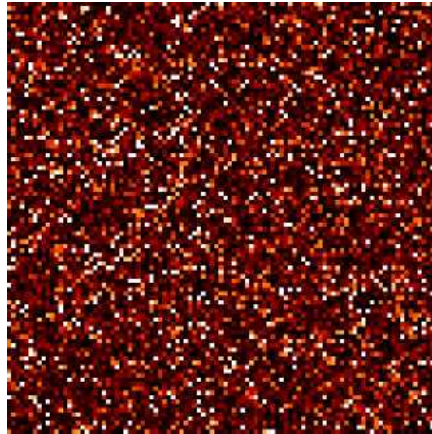
>>> from fipy.tools.numerix import arange, exp

>>> 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)
```

Parameters

mesh: The mesh on which to define the noise.

mean: The mean of the distribution μ .

Overrides: [fipy.variables.noiseVariable.NoiseVariable.__init__\(\)](#)

Inherited from object: `__delattr__`, `__getattr__`, `__hash__`, `__reduce__`, `__reduce_ex__`, `__setattr__`

Inherited from [CellVariable](#): `__call__`, `__getstate__`, `__setstate__`, `getArithmeticFaceValue`, `getCellVolumeAverage`, `getFaceGrad`, `getFaceGradAverage`, `getFaceValue`, `getGaussGrad`, `getGrad`, `getHarmonicFaceValue`, `getLeastSquaresGrad`, `getOld`, `getValue`, `setValue`, `updateOld`

Inherited from [NoiseVariable](#): `copy`, `scramble`

Inherited from Variable: `__abs__`, `__add__`, `__and__`, `__array__`, `__array_wrap__`, `__div__`, `__eq__`, `__float__`, `__ge__`, `__getitem__`, `__gt__`, `__le__`, `__len__`, `__lt__`, `__mod__`, `__mul__`, `__ne__`, `__neg__`, `__or__`, `__pos__`, `__pow__`, `__radd__`, `__rdiv__`, `__repr__`, `__rmul__`, `__rpow__`, `__rsub__`, `__setitem__`, `__str__`, `__sub__`, `allclose`, `allequal`, `arccos`, `arccosh`, `arcsin`, `arcsinh`, `arctan`, `arctan2`, `arctanh`, `cacheMe`, `ceil`, `conjugate`, `cos`, `cosh`, `dontCacheMe`, `dot`, `exp`, `floor`, `getMag`, `getMesh`, `getName`, `getNumericValue`, `getShape`, `getSubscribedVariables`, `getTypecode`, `getUnit`, `inBaseUnits`, `inUnitsOf`, `log`, `log10`, `put`, `reshape`, `setName`, `sin`, `sinh`, `sqrt`, `sum`, `take`, `tan`, `tanh`, `tostring`, `transpose`

Static Methods

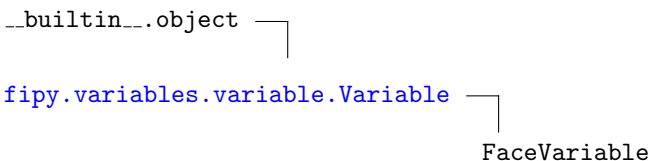
Inherited from Variable: `__new__`

Class Variables

Name	Description
Inherited from Variable:	<code>__array_priority__</code>

9.4 Module `fipy.variables.faceVariable`

Class FaceVariable



Known Subclasses: `_Alpha`, `_Alpha`, `_Alpha`, `_Alpha`, `_Alpha`, `_CellToFaceVariable`

Methods

`__init__(self, mesh, name='', value=0.0, unit=None)`

Overrides: `fipy.variables.variable.Variable.__init__()`
`setValue(self, value, faces=(), unit=None, where=None)`

Overrides: `fipy.variables.variable.Variable.setValue()`

Inherited from object: `__delattr__`, `__getattr__`, `__hash__`, `__reduce__`, `__reduce_ex__`, `__setattr__`

Inherited from Variable: `__abs__`, `__add__`, `__and__`, `__array__`, `__array_wrap__`, `__call__`, `__div__`, `__eq__`, `__float__`, `__ge__`, `__getitem__`, `__gt__`, `__le__`, `__len__`, `__lt__`, `__mod__`, `__mul__`, `__ne__`, `__neg__`, `__or__`, `__pos__`, `__pow__`, `__radd__`, `__rdiv__`, `__repr__`, `__rmul__`, `__rpow__`, `__rsub__`, `__setitem__`, `__str__`, `__sub__`, `allclose`, `allequal`, `arccos`, `arccosh`, `arcsin`, `arcsinh`, `arctan`, `arctan2`, `arctanh`, `cacheMe`, `ceil`, `conjugate`, `copy`, `cos`, `cosh`, `dontCacheMe`, `dot`, `exp`, `floor`, `getMag`, `getMesh`, `getName`, `getNumericValue`, `getShape`, `getSubscribedVariables`, `getTypecode`, `getUnit`, `getValue`, `inBaseUnits`, `inUnitsOf`, `log`, `log10`, `put`, `reshape`, `setName`, `sin`, `sinh`, `sqrt`, `sum`, `take`, `tan`, `tanh`, `tostring`, `transpose`

Static Methods

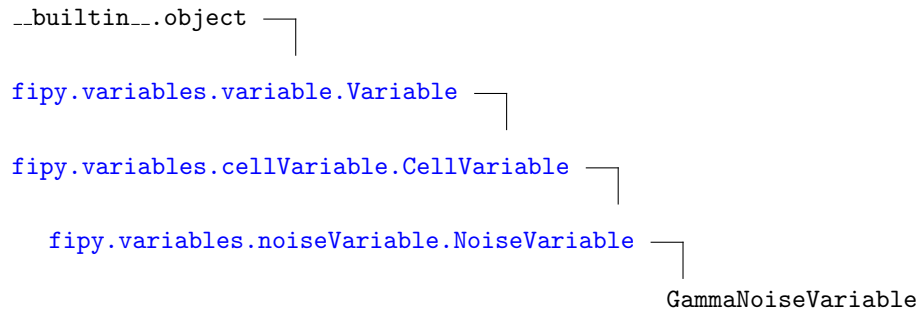
Inherited from Variable: `__new__`

Class Variables

Name	Description
Inherited from Variable:	<code>__array_priority__</code>

9.5 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 viewers
...     viewer = viewers.make(vars = noise, limits = {'datamin': 0, 'datamax': 30})
...     histoplot = viewers.make(vars = (histogram, gammadist), limits = {'datamin': 0, 'datamax':

>>> from fipy.tools.numerix import arange, exp
>>> from scipy.special import gamma as Gamma

>>> for shape in arange(1,8,1):
...     alpha.setValue(shape)

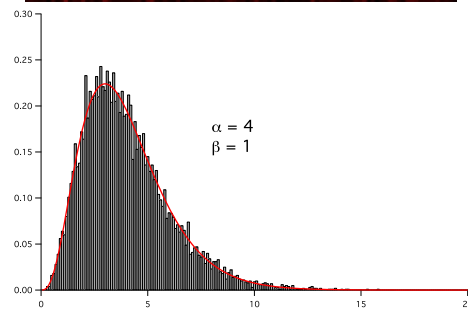
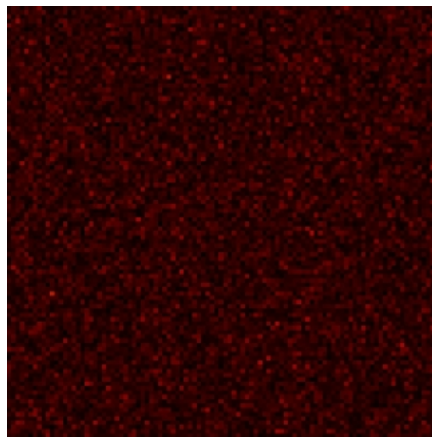
```

```

...     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)`

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.noiseVariable.NoiseVariable.__init__\(\)](#)

Inherited from object: `__delattr__`, `__getattr__`, `__hash__`, `__reduce__`, `__reduce_ex__`, `__setattr__`

Inherited from `CellVariable`: `__call__`, `__getstate__`, `__setstate__`, `getArithmeticFaceValue`, `getCellVolumeAverage`, `getFaceGrad`, `getFaceGradAverage`, `getFaceValue`, `getGaussGrad`, `getGrad`, `getHarmonicFaceValue`, `getLeastSquaresGrad`, `getOld`, `getValue`, `setValue`, `updateOld`

Inherited from `NoiseVariable`: `copy`, `scramble`

Inherited from `Variable`: `__abs__`, `__add__`, `__and__`, `__array__`, `__array_wrap__`, `__div__`, `__eq__`, `__float__`, `__ge__`, `__getitem__`, `__gt__`, `__le__`, `__len__`, `__lt__`, `__mod__`, `__mul__`, `__ne__`, `__neg__`, `__or__`, `__pos__`, `__pow__`, `__radd__`, `__rdiv__`, `__repr__`, `__rmul__`, `__rpow__`, `__rsub__`, `__setitem__`, `__str__`, `__sub__`, `allclose`, `allequal`, `arccos`, `arccosh`, `arcsin`, `arcsinh`, `arctan`, `arctan2`, `arctanh`, `cacheMe`, `ceil`, `conjugate`, `cos`, `cosh`, `dontCacheMe`, `dot`, `exp`, `floor`, `getMag`, `getMesh`, `getName`, `getNumericValue`, `getShape`, `getSubscribedVariables`, `getTypecode`, `getUnit`, `inBaseUnits`, `inUnitsOf`, `log`, `log10`, `put`, `reshape`, `setName`, `sin`, `sinh`, `sqrt`, `sum`, `take`, `tan`, `tanh`, `tostring`, `transpose`

Static Methods

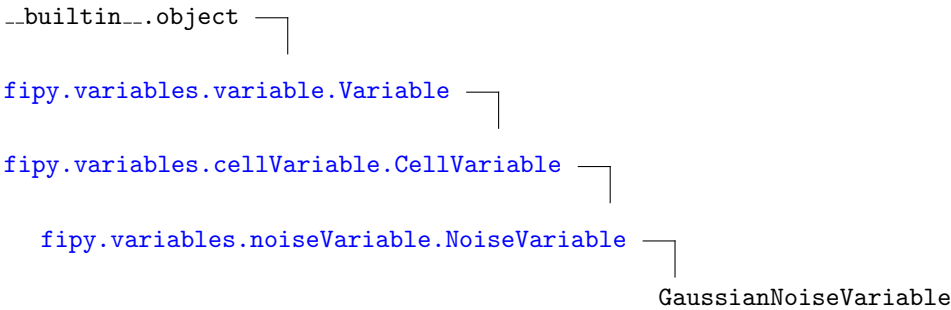
Inherited from `Variable`: `__new__`

Class Variables

Name	Description
Inherited from <code>Variable</code> : <code>__array_priority__</code>	

9.6 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

$$\frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$$

For example, the variance of thermal noise that is uncorrelated in space and time is often expressed as

$$\langle \eta(\vec{r}, t)\eta(\vec{r}', t') \rangle = Mk_B T \delta(\vec{r} - \vec{r}') \delta(t - t')$$

which can be obtained with:

```
sigmaSqr = Mobility * kBoltzmann * Temperature / (mesh.getCellVolumes() * timeStep)
GaussianNoiseVariable(mesh = mesh, variance = sigmaSqr)
```

Note

If the time step will change as the simulation progresses, either through use of an adaptive iterator or by making manual changes at different stages, remember to declare `timeStep` as a `Variable` and to change its value with its `setValue()` method.

```
>>> import sys
>>> from fipy.tools.numerix import *

>>> mean = 0.
>>> variance = 4.
```

We generate noise on a non-uniform cartesian mesh with cell dimensions of x^2 and y^3 .

```
>>> from fipy.meshes.grid2D import Grid2D
>>> mesh = Grid2D(dx = arange(0.1, 5., 0.1)**2, dy = arange(0.1, 3., 0.1)**3)
>>> 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

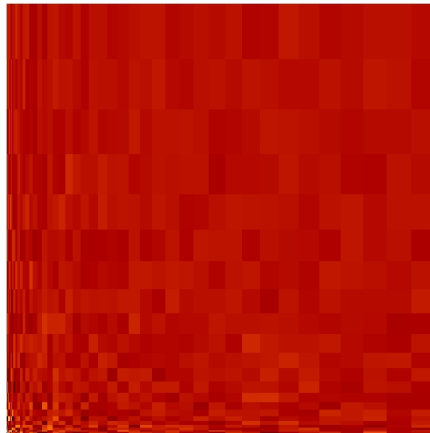
```
>>> from fipy.variables.cellVariable import CellVariable
>>> gauss = CellVariable(mesh = histogram.getMesh())
>>> x = histogram.getMesh().getCellCenters()[...,0]
>>> gauss.setValue((1/(sqrt(variance * 2 * pi))) * exp(-(x - mean)**2 / (2 * variance)))

>>> if __name__ == '__main__':
...     from fipy import viewers
...     viewer = viewers.make(vars = noise, limits = {'datamin':-5, 'datamax':5})
...     histoplot = viewers.make(vars = (histogram, gauss))

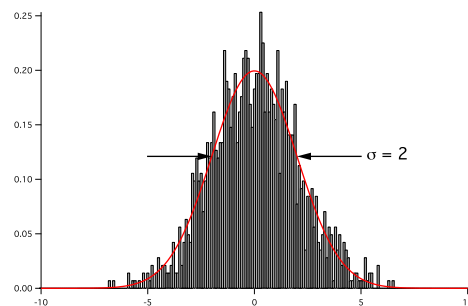
>>> for i in range(10):
...     noise.scramble()
...     if __name__ == '__main__':
...         viewer.plot()
...         histoplot.plot()

>>> 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)`

Parameters

mesh: The mesh on which to define the noise.

mean: The mean of the noise distribution, μ .

variance: The variance of the noise distribution, σ^2 .

Overrides: `fipy.variables.noiseVariable.NoiseVariable.__init__()`

Inherited from object: `__delattr__`, `__getattr__`, `__hash__`, `__reduce__`, `__reduce_ex__`, `__setattr__`

Inherited from `CellVariable`: `__call__`, `__getstate__`, `__setstate__`, `getArithmeticFaceValue`, `getCellVolumeAverage`, `getFaceGrad`, `getFaceGradAverage`, `getFaceValue`, `getGaussGrad`, `getGrad`, `getHarmonicFaceValue`, `getLeastSquaresGrad`, `getOld`, `getValue`, `setValue`, `updateOld`

Inherited from `NoiseVariable`: `copy`, `scramble`

Inherited from `Variable`: `__abs__`, `__add__`, `__and__`, `__array__`, `__array_wrap__`, `__div__`, `__eq__`, `__float__`, `__ge__`, `__getitem__`, `__gt__`, `__le__`, `__len__`, `__lt__`, `__mod__`, `__mul__`, `__ne__`, `__neg__`, `__or__`, `__pos__`, `__pow__`, `__radd__`, `__rdiv__`, `__repr__`, `__rmul__`, `__rpow__`, `__rsub__`, `__setitem__`, `__str__`, `__sub__`, `allclose`, `allegual`, `arccos`, `arccosh`, `arcsin`, `arcsinh`, `arctan`, `arctan2`, `arctanh`, `cacheMe`, `ceil`, `conjugate`, `cos`, `cosh`, `dontCacheMe`, `dot`, `exp`, `floor`, `getMag`, `getMesh`, `getName`, `getNumericValue`, `getShape`, `getSubscribedVariables`, `getTypecode`, `getUnit`, `inBaseUnits`, `inUnitsOf`, `log`, `log10`, `put`, `reshape`, `setName`, `sin`, `sinh`, `sqrt`, `sum`, `take`, `tan`, `tanh`, `tostring`, `transpose`

Static Methods

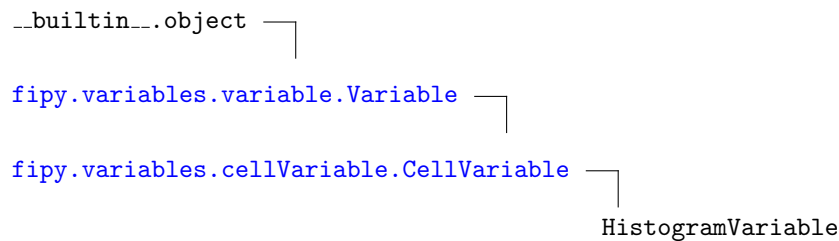
Inherited from `Variable`: `__new__`

Class Variables

Name	Description
Inherited from <code>Variable</code> : <code>__array_priority__</code>	

9.7 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.cellVariable.CellVariable.__init__()`

Inherited from object: `__delattr__`, `__getattr__`, `__hash__`, `__reduce__`, `__reduce_ex__`, `__setattr__`

Inherited from `CellVariable`: `__call__`, `__getstate__`, `__setstate__`, `copy`, `getArithmeticFaceValue`, `getCellVolumeAverage`, `getFaceGrad`, `getFaceGradAverage`, `getFaceValue`, `getGaussGrad`, `getGrad`, `getHarmonicFaceValue`, `getLeastSquaresGrad`, `getOld`, `getValue`, `setValue`, `updateOld`

Inherited from `Variable`: `__abs__`, `__add__`, `__and__`, `__array__`, `__array_wrap__`, `__div__`, `__eq__`, `__float__`, `__ge__`, `__getitem__`, `__gt__`, `__le__`, `__len__`, `__lt__`, `__mod__`, `__mul__`, `__ne__`, `__neg__`, `__or__`, `__pos__`, `__pow__`, `__radd__`, `__rdiv__`, `__repr__`, `__rmul__`, `__rpow__`, `__rsub__`, `__setitem__`, `__str__`, `__sub__`, `allclose`, `allequal`, `arccos`, `arccosh`, `arcsin`, `arcsinh`, `arctan`, `arctan2`, `arctanh`, `cacheMe`, `ceil`, `conjugate`, `cos`, `cosh`, `dontCacheMe`, `dot`, `exp`, `floor`, `getMag`, `getMesh`, `getName`, `getNumericValue`, `getShape`, `getSubscribedVariables`, `getTypecode`, `getUnit`, `inBaseUnits`, `inUnitsOf`, `log`, `log10`, `put`, `reshape`, `setName`, `sin`, `sinh`, `sqrt`, `sum`, `take`, `tan`, `tanh`, `tostring`, `transpose`

Static Methods

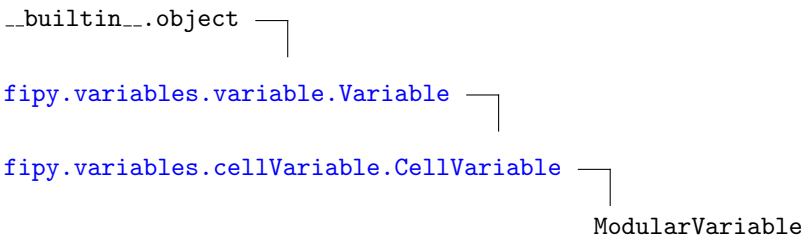
Inherited from `Variable`: `__new__`

Class Variables

Name	Description
Inherited from Variable:	__array_priority__

9.8 Module `fipy.variables.modularVariable`

Class `ModularVariable`



The `ModularVariable` defines a variable that exists on the circle between $-\pi$ and π . The following examples show how `ModularVariable` works. When subtracting the answer wraps back around the circle.

```

>>> from fipy.meshes.grid1D import Grid1D
>>> mesh = Grid1D(nx = 2)
>>> from fipy.tools import numerix
>>> pi = numerix.pi
>>> v1 = ModularVariable(mesh = mesh, value = (2*pi/3, -2*pi/3))
>>> v2 = ModularVariable(mesh = mesh, value = -2*pi/3)
>>> print numerix.allclose(v2 - v1, (2*pi/3, 0))
1

```

Obtaining the arithmetic face value.

```

>>> print numerix.allclose(v1.getArithmeticFaceValue(), (2*pi/3, -pi, -2*pi/3))
1

```

Obtaining the gradient.

```

>>> print numerix.allclose(v1.getGrad(), (pi/3, pi/3))
1

```

Obtaining the gradient at the faces.

```

>>> print numerix.allclose(v1.getFaceGrad(), [[0], [2*pi/3], [0]])
1

```

Obtaining the gradient at the faces but without modular arithmetic.

```

>>> print numerix.allclose(v1.getFaceGradNoMod(), [[0], [-4*pi/3], [0]])
1

```

Methods

```

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

```

Creates a `ModularVariable` object.

Parameters

mesh: The mesh that defines the geometry of this variable.

name: The user-readable name of the variable.

value: The initial value.

unit: The physical units of the variable.

hasOld: Whether a variable keeps its old variable.

Overrides: `fipy.variables.cellVariable.CellVariable.__init__()`

`__rsub__(self, other)`

Overrides: `fipy.variables.variable.Variable.__rsub__()`

`__sub__(self, other)`

Overrides: `fipy.variables.variable.Variable.__sub__()`

`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 `VectorFaceVariable` (second-order gradient). Adjusted for a `ModularVariable`

Overrides: `fipy.variables.cellVariable.CellVariable.getFaceGrad()`

`getFaceGradNoMod(self)`

$\nabla\phi$ as a `VectorFaceVariable` (second-order gradient). Not adjusted for a `ModularVariable`

`getGrad(self)`

Return $\nabla\phi$ as a `VectorCellVariable` (first-order gradient). Adjusted for a `ModularVariable`

Overrides: `fipy.variables.cellVariable.CellVariable.getGrad()`

`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\(\)](#)

Inherited from object: [__delattr__](#), [__getattr__](#), [__hash__](#), [__reduce__](#), [__reduce_ex__](#), [__setattr__](#)

Inherited from [CellVariable](#): [__call__](#), [__getstate__](#), [__setstate__](#), [copy](#), [getCellVolumeAverage](#), [getFaceGradAverage](#), [getFaceValue](#), [getGaussGrad](#), [getHarmonicFaceValue](#), [getLeastSquaresGrad](#), [getOld](#), [getValue](#), [setValue](#)

Inherited from [Variable](#): [__abs__](#), [__add__](#), [__and__](#), [__array__](#), [__array_wrap__](#), [__div__](#), [__eq__](#), [__float__](#), [__ge__](#), [__getitem__](#), [__gt__](#), [__le__](#), [__len__](#), [__lt__](#), [__mod__](#), [__mul__](#), [__ne__](#), [__neg__](#), [__or__](#), [__pos__](#), [__pow__](#), [__radd__](#), [__rdiv__](#), [__repr__](#), [__rmul__](#), [__rpow__](#), [__setitem__](#), [__str__](#), [allclose](#), [allequal](#), [arccos](#), [arccosh](#), [arcsin](#), [arcsinh](#), [arctan](#), [arctan2](#), [arctanh](#), [cacheMe](#), [ceil](#), [conjugate](#), [cos](#), [cosh](#), [dontCacheMe](#), [dot](#), [exp](#), [floor](#), [getMag](#), [getMesh](#), [getName](#), [getNumericValue](#), [getShape](#), [getSubscribedVariables](#), [getTypecode](#), [getUnit](#), [inBaseUnits](#), [inUnitsOf](#), [log](#), [log10](#), [put](#), [reshape](#), [setName](#), [sin](#), [sinh](#), [sqrt](#), [sum](#), [take](#), [tan](#), [tanh](#), [tostring](#), [transpose](#)

Static Methods

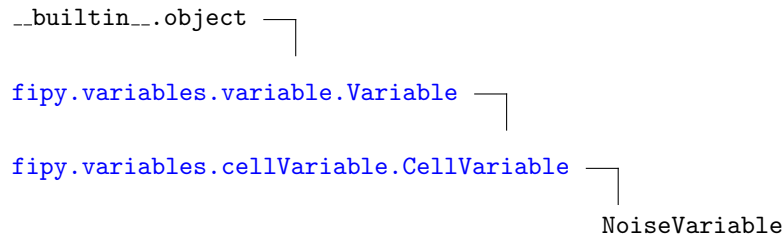
Inherited from [Variable](#): [__new__](#)

Class Variables

Name	Description
Inherited from Variable : __array_priority__	

9.9 Module `fipy.variables.noiseVariable`

Class `NoiseVariable`



Known Subclasses: `BetaNoiseVariable`, `ExponentialNoiseVariable`, `GammaNoiseVariable`, `GaussianNoiseVariable`, `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)
```

Overrides: `fipy.variables.cellVariable.CellVariable.__init__()`
`copy(self)`

Copy the value of the `NoiseVariable` to a static `CellVariable`.

Overrides: `fipy.variables.cellVariable.CellVariable.copy()`

```
scramble(self)
```

Generate a new random distribution.

Inherited from object: `__delattr__`, `__getattr__`, `__hash__`, `__reduce__`, `__reduce_ex__`, `__setattr__`

Inherited from `CellVariable`: `__call__`, `__getstate__`, `__setstate__`, `getArithmeticFaceValue`, `getCellVolumeAverage`, `getFaceGrad`, `getFaceGradAverage`, `getFaceValue`, `getGaussGrad`, `getGrad`, `getHarmonicFaceValue`, `getLeastSquaresGrad`, `getOld`, `getValue`, `setValue`, `updateOld`

Inherited from *Variable*: `__abs__`, `__add__`, `__and__`, `__array__`, `__array_wrap__`, `__div__`, `__eq__`, `__float__`, `__ge__`, `__getitem__`, `__gt__`, `__le__`, `__len__`, `__lt__`, `__mod__`, `__mul__`, `__ne__`, `__neg__`, `__or__`, `__pos__`, `__pow__`, `__radd__`, `__rdiv__`, `__repr__`, `__rmul__`, `__rpow__`, `__rsub__`, `__setitem__`, `__str__`, `__sub__`, `allclose`, `allequal`, `arccos`, `arccosh`, `arcsin`, `arcsinh`, `arctan`, `arctan2`, `arctanh`, `cacheMe`, `ceil`, `conjugate`, `cos`, `cosh`, `dontCacheMe`, `dot`, `exp`, `floor`, `getMag`, `getMesh`, `getName`, `getNumericValue`, `getShape`, `getSubscribedVariables`, `getTypecode`, `getUnit`, `inBaseUnits`, `inUnitsOf`, `log`, `log10`, `put`, `reshape`, `setName`, `sin`, `sinh`, `sqrt`, `sum`, `take`, `tan`, `tanh`, `tostring`, `transpose`

Static Methods

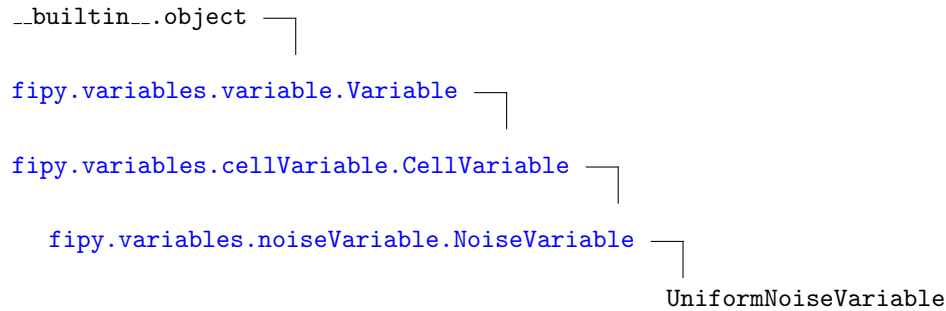
Inherited from *Variable*: `__new__`

Class Variables

Name	Description
Inherited from <i>Variable</i> : <code>__array_priority__</code>	

9.10 Module `fipy.variables.uniformNoiseVariable`

Class `UniformNoiseVariable`



Represents a uniform distribution of random numbers.

We generate noise on a uniform cartesian mesh

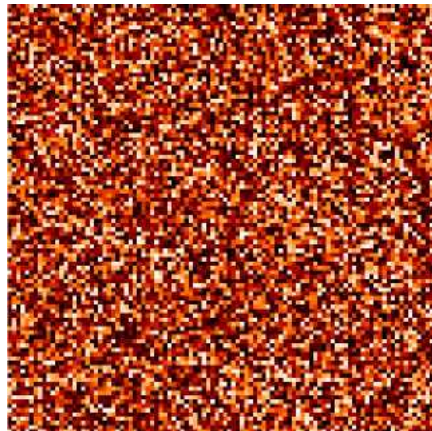
```
>>> from fipy.meshes.grid2D import Grid2D
>>> noise = UniformNoiseVariable(mesh = Grid2D(nx = 100, ny = 100))
```

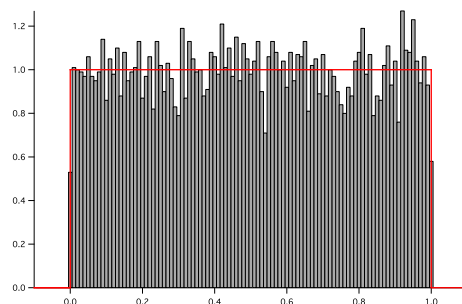
and histogram the noise

```
>>> 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 = viewers.make(vars = noise, limits = {'datamin':0, 'datamax':1})
...     histoplot = viewers.make(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)`

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.noiseVariable.NoiseVariable.__init__()`

Inherited from object: `__delattr__`, `__getattr__`, `__hash__`, `__reduce__`, `__reduce_ex__`, `__setattr__`

Inherited from `CellVariable`: `__call__`, `__getstate__`, `__setstate__`, `getArithmeticFaceValue`, `getCellVolumeAverage`, `getFaceGrad`, `getFaceGradAverage`, `getFaceValue`, `getGaussGrad`, `getGrad`, `getHarmonicFaceValue`, `getLeastSquaresGrad`, `getOld`, `getValue`, `setValue`, `updateOld`

Inherited from `NoiseVariable`: `copy`, `scramble`

Inherited from `Variable`: `__abs__`, `__add__`, `__and__`, `__array__`, `__array_wrap__`, `__div__`, `__eq__`, `__float__`, `__ge__`, `__getitem__`, `__gt__`, `__le__`, `__len__`, `__lt__`, `__mod__`, `__mul__`, `__ne__`, `__neg__`, `__or__`, `__pos__`, `__pow__`, `__radd__`, `__rdiv__`, `__repr__`, `__rmul__`, `__rpow__`, `__rsub__`, `__setitem__`, `__str__`, `__sub__`, `allclose`, `allequal`, `arccos`, `arccosh`, `arcsin`, `arcsinh`, `arctan`, `arctan2`, `arctanh`, `cacheMe`, `ceil`, `conjugate`, `cos`, `cosh`, `dontCacheMe`, `dot`, `exp`, `floor`, `getMag`, `getMesh`, `getName`, `getNumericValue`, `getShape`, `getSubscribedVariables`, `getTypecode`, `getUnit`, `inBaseUnits`, `inUnitsOf`, `log`, `log10`, `put`, `reshape`, `setName`, `sin`, `sinh`, `sqrt`, `sum`, `take`, `tan`, `tanh`, `tostring`, `transpose`

Static Methods

Inherited from `Variable`: `__new__`

Class Variables

Name	Description
Inherited from Variable:	__array_priority__

9.11 Module `fipy.variables.variable`

Class Variable

```
__builtin__.object ┌
                   │
                   └ Variable
```

Known Subclasses: `CellVariable`, `FaceVariable`, `VectorCellVariable`, `VectorFaceVariable`, `_CellVolumeAverageVariable`, `_Constant`, `_MagVariable`, `_NewAxisVariable`, `_SumVariable`

Methods

```
__init__(self, value=0.0, unit=None, array=None, name='', mesh=None, 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
mesh: the mesh that defines the geometry of this Variable

Overrides: `__builtin__.object.__init__()`

```
__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
```

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

```
__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))
>>> print (a == 0) & (b == 1)
[0 1 0 0]
>>> 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)
>>> print (a == 0) & (b == 1)
[0 1 0 0]
>>> print a & b
[0 0 0 1]
```

```
__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.]
```

It is an error to convert a dimensional `Variable` to a `Numeric array`

```
>>> v = Variable(value = [2,3], unit = "m")
>>> numerix.array(v)
Traceback (most recent call last):
...
TypeError: Numeric array value must be dimensionless
```

Convert a list of 1 element `Variables` to an array

```
>>> numerix.array([Variable(0), Variable(0)])
[[0,]
 [0,]]
>>> print Variable(0) + Variable(0)
0
>>> numerix.array([Variable(0) + Variable(0), Variable(0)])
>>> numerix.array([Variable(0), Variable(0) + Variable(0)])
[[0,]
 [0,]]
```

```
__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.variable.binOp'>
```

```
>>> from scipy.special import gamma as Gamma
>>> print type(Gamma(Variable([1.0, 2.0])))
<type 'numpy.ndarray'>
```

`__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
```

`__div__(self, other)`

`__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
```

`__float__(self)`

`__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
```

`__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
```

`__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
```

`__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
```

`__len__(self)`

`__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)

```

```
__mod__(self, other)
```

```
__mul__(self, other)
```

```
__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

```

```
__neg__(self)
```

```
__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))
>>> print (a == 0) | (b == 1)
[1 1 0 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)
>>> print (a == 0) | (b == 1)
[1 1 0 1]
>>> print a | b
[0 1 1 1]
```

`--pos--(self)`

`--pow--(self, other)`

`--radd--(self, other)`

`--rdiv--(self, other)`

`--repr--(self)`

Overrides: `--builtin--.object.--repr--()`

`--rmul--(self, other)`

`--rpow--(self, other)`

`--rsub--(self, other)`

`--setitem--(self, index, value)`

`--str--(self)`

Overrides: `--builtin--.object.--str--()`

`--sub--(self, other)`

`allclose(self, other, rtol=1e-10, atol=1e-10)`

```
>>> var = Variable((1, 1))
>>> print var.allclose((1, 1))
1
>>> print var.allclose((1,))
1
>>> print var.allclose((1,1,1))
Traceback (most recent call last):
...
ValueError
```

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))
>>> var.allclose(numerix.ones(10001))
Traceback (most recent call last):
...
ValueError
```

`allequal(self, other)`

`arccos(self)`

`arccosh(self)`

`arcsin(self)`

`arcsinh(self)`

`arctan(self)`

`arctan2(self, other)`

`arctanh(self)`

`cacheMe(self, recursive=False)`

`ceil(self)`

`conjugate(self)`

`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]))
```

`cos(self)`

`cosh(self)`

`dontCacheMe(self, recursive=False)`

`dot(self, other)`

`exp(self)`

`floor(self)`

`getMag(self)`

`getMesh(self)`

`getName(self)`

`getNumericValue(self)`

`getShape(self)`

```
>>> Variable(value = 3).getShape()
()
>>> Variable(value = (3,)).getShape()
(1,)
>>> Variable(value = (3,4)).getShape()
(2,)

>>> Variable(value = "3 m").getShape()
()
>>> Variable(value = (3,), unit = "m").getShape()
(1,)
>>> Variable(value = (3,4), unit = "m").getShape()
(2,)

>>> from fipy.meshes.grid2D import Grid2D
>>> from fipy.variables.cellVariable import CellVariable
>>> mesh = Grid2D(nx = 2, ny = 3)
>>> var = CellVariable(mesh = mesh)
>>> var.getShape()
(6,)
>>> var.getArithmeticFaceValue().getShape()
(17,)
>>> var.getGrad().getShape()
(6, 2)
>>> var.getFaceGrad().getShape()
(17, 2)
```

`getSubscribedVariables(self)`

`getTypecode(self)`

Returns the Numpy typecode of the underlying array.

```
>>> Variable(1).getTypecode()
'1'
>>> Variable(1.).getTypecode()
'd'
>>> Variable((1,1.)).getTypecode()
'd'
```

`getUnit(self)`

Return the unit object of *self*. >>> Variable(value = "1 m").getUnit()
<PhysicalUnit m>

`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
```

`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']
```

`log(self)`

`log10(self)`

`put(self, indices, value)`

`reshape(self, shape)`

`setName(self, name)`

`setValue(self, value, unit=None, array=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: array dimensions must agree
```

`sin(self)`

`sinh(self)`

`sqrt(self)`

```
>>> from fipy.meshes.grid1D import Grid1D
>>> mesh= Grid1D(nx=3)

>>> from fipy.variables.vectorCellVariable import VectorCellVariable
>>> var = VectorCellVariable(mesh=mesh, value=((0.), (2.), (3.)))
>>> print (var.dot(var)).sqrt()
[ 0.  2.  3.]
```

`sum(self, index=0)`

`take(self, ids, axis=0)`

`tan(self)`

`tanh(self)`

`tostring(self, max_line_width=None, precision=None, suppress_small=None, separator=' ')`

`transpose(self)`

Inherited from object: `__delattr__`, `__getattr__`, `__hash__`, `__reduce__`, `__reduce_ex__`, `__setattr__`

Static Methods

`__new__(cls, *args, **kwargs)`

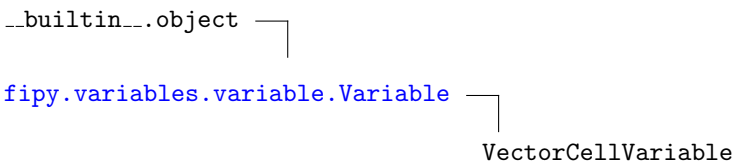
Overrides: `__builtin__.object.__new__()`

Class Variables

Name	Description
<code>__array_priority__</code>	Value: 100.0 (<i>type=float</i>)

9.12 Module `fipy.variables.vectorCellVariable`

Class `VectorCellVariable`



Known Subclasses: `_FaceGradContributions`, `_GaussCellGradVariable`,
`_LeastSquaresCellGradVariable`

Methods

`__init__(self, mesh, name='', value=0.0, unit=None)`

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

`__call__(self, point=None)`

“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

```

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

`dot(self, other)`

Overrides: `fipy.variables.variable.Variable.dot()`

`getArithmeticFaceValue(self)`

Return a `VectorFaceVariable` with values determined by the arithmetic mean from the neighboring cells.

```

>>> from fipy.meshes.grid2D import Grid2D
>>> mesh = Grid2D(dx = 1., nx = 2)
>>> var = VectorCellVariable(mesh, value = numerix.array(((0,0),(1,1))))
>>> answer = numerix.array(((0, 0), (1, 1), (0, 0), (1, 1),
...                          (0, 0), (.5, .5), (1, 1)))
>>> numerix.allclose(answer, numerix.array(var.getArithmeticFaceValue()))
1

```

`getDivergence(self)`

`getHarmonicFaceValue(self)`

Return a `VectorFaceVariable` with values determined by the harmonic mean from the neighboring cells.

```
>>> from fipy.meshes.grid2D import Grid2D
>>> mesh = Grid2D(dx = 1., nx = 2)
>>> var = VectorCellVariable(mesh, value = numerix.array(((0,0),(1,1))))
>>> print var.getHarmonicFaceValue()
[[ 0.  0.]
 [ 1.  1.]
 [ 0.  0.]
 [ 1.  1.]
 [ 0.  0.]
 [ 0.  0.]
 [ 1.  1.]]

>>> from fipy.meshes.grid2D import Grid2D
>>> mesh = Grid2D(dx = 0.1, nx = 2)
>>> var = VectorCellVariable(mesh, value = numerix.array(((0,0.3),(2.,0.4))))
>>> print var.getHarmonicFaceValue()
[[ 0.          0.3         ]
 [ 2.          0.4         ]
 [ 0.          0.3         ]
 [ 2.          0.4         ]
 [ 0.          0.3         ]
 [ 0.          0.34285714]
 [ 2.          0.4         ]]
```

Inherited from object: `__delattr__`, `__getattr__`, `__hash__`, `__reduce__`, `__reduce_ex__`, `__setattr__`

Inherited from `Variable`: `__abs__`, `__add__`, `__and__`, `__array__`, `__array_wrap__`, `__div__`, `__eq__`, `__float__`, `__ge__`, `__getitem__`, `__gt__`, `__le__`, `__len__`, `__lt__`, `__mod__`, `__mul__`, `__ne__`, `__neg__`, `__or__`, `__pos__`, `__pow__`, `__radd__`, `__rdiv__`, `__repr__`, `__rmul__`, `__rpow__`, `__rsub__`, `__setitem__`, `__str__`, `__sub__`, `allclose`, `allequal`, `arccos`, `arccosh`, `arcsin`, `arcsinh`, `arctan`, `arctan2`, `arctanh`, `cacheMe`, `ceil`, `conjugate`, `copy`, `cos`, `cosh`, `dontCacheMe`, `exp`, `floor`, `getMag`, `getMesh`, `getName`, `getNumericValue`, `getShape`, `getSubscribedVariables`, `getTypecode`, `getUnit`, `getValue`, `inBaseUnits`, `inUnitsOf`, `log`, `log10`, `put`, `reshape`, `setName`, `setValue`, `sin`, `sinh`, `sqrt`, `sum`, `take`, `tan`, `tanh`, `tostring`, `transpose`

Static Methods

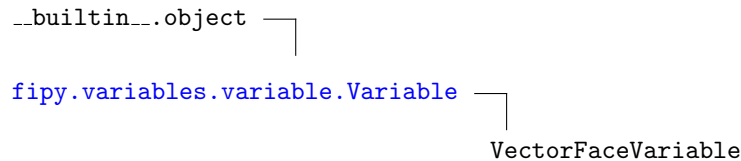
Inherited from `Variable`: `__new__`

Class Variables

Name	Description
Inherited from Variable:	__array_priority__

9.13 Module `fipy.variables.vectorFaceVariable`

Class `VectorFaceVariable`



Known Subclasses: `_ConvectionCoeff`, `_FaceGradVariable`, `_VectorCellToFaceVariable`, `_VectorFaceDifferenceVariable`

Methods

```
__init__(self, mesh, name='', value=0.0, unit=None)
```

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

```
__call__(self, point=None)
```

“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

```

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

```
dot(self, other)
```

Overrides: `fipy.variables.variable.Variable.dot()`

```
getDivergence(self)
```

Inherited from object: `__delattr__`, `__getattr__`, `__hash__`, `__reduce__`, `__reduce_ex__`, `__setattr__`

Inherited from `Variable`: `__abs__`, `__add__`, `__and__`, `__array__`, `__array_wrap__`, `__div__`, `__eq__`, `__float__`, `__ge__`, `__getitem__`, `__gt__`, `__le__`, `__len__`, `__lt__`, `__mod__`, `__mul__`, `__ne__`, `__neg__`, `__or__`, `__pos__`, `__pow__`, `__radd__`, `__rdiv__`, `__repr__`, `__rmul__`, `__rpow__`, `__rsub__`, `__setitem__`, `__str__`, `__sub__`, `allclose`, `allequal`, `arccos`, `arccosh`, `arcsin`, `arcsinh`, `arctan`, `arctan2`, `arctanh`, `cacheMe`, `ceil`, `conjugate`, `copy`, `cos`, `cosh`, `dontCacheMe`, `exp`, `floor`, `getMag`, `getMesh`, `getName`,

`getNumericValue`, `getShape`, `getSubscribedVariables`, `getTypecode`, `getUnit`, `getValue`,
`inBaseUnits`, `inUnitsOf`, `log`, `log10`, `put`, `reshape`, `setName`, `setValue`, `sin`, `sinh`, `sqrt`, `sum`,
`take`, `tan`, `tanh`, `tostring`, `transpose`

Static Methods

Inherited from `Variable`: `__new__`

Class Variables

Name	Description
Inherited from <code>Variable</code> : <code>__array_priority__</code>	

Chapter 10

Package `fipy.viewers`

10.1 Functions

`make(vars, title=None, limits=None)`

Generic function for creating a `Viewer`. The `make` function 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 `limits` parameter can be used to constrain the view. For example:

```
fipy.viewers.make(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.

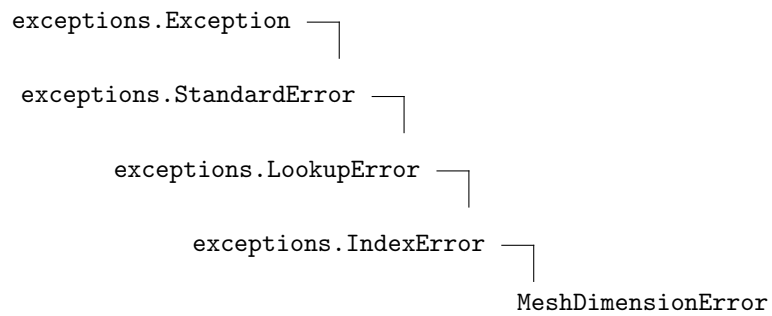
Parameters

vars: a `CellVariable` or tuple of `CellVariable` objects to plot

title: displayed at the top of the `Viewer` window

limits: a 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.

10.2 Class `MeshDimensionError`

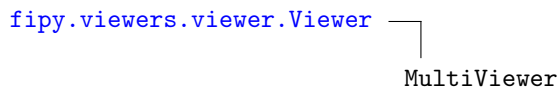


Methods

Inherited from `Exception`: `__init__`, `__getitem__`, `__str__`

10.3 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)`

Overrides: `fipy.viewers.viewer.Viewer.__init__()`
`getViewers(self)`

`plot(self)`

Overrides: `fipy.viewers.viewer.Viewer.plot()`
`setLimits(self, limits)`

Update the limits.

Parameters


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.

Overrides: `fipy.viewers.viewer.Viewer.setLimits()` (*inherited documentation*)

Inherited from `Viewer`: `getVars`

10.4 Module `fipy.viewers.tsvViewer`

Class `TSVViewer`

`fipy.viewers.viewer.Viewer`  `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   ...
:
:
```

Any cell centers that lie outside the `limits` provided will not be included. Any values that lie outside the `datamin` or `datamax` of `limits` will be replaced with `nan`.

All variables must have the same mesh.

It tries to do something reasonable with `VectorCellVariable` and `VectorFaceVariable` objects.

Methods

`__init__(self, vars, limits=None, title=None)`

Creates a `TSVViewer`.

Parameters

vars: A tuple or list of `CellVariable`, `VectorCellVariable`, `FaceVariable`, `VectorFaceVariable` objects.

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.

title: displayed at the top of the Viewer output

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()
x      var      var_grad_x
0.2    0         2.5
0.6    2         6.25
1      5         3.75

>>> from fipy.meshes.grid2D import Grid2D
>>> m = Grid2D(nx = 2, dx = .1, ny = 2, dy = 0.3)
>>> v = CellVariable(mesh = m, name = "var", value = (0, 2, -2, 5))
>>> TSVViewer(vars = (v, v.getGrad())).plot()
x      y      var      var_grad_x      var_grad_y
0.05   0.15   0        10        -3.333333333333333
0.15   0.15   2         10         5
0.05   0.45  -2         35        -3.333333333333333
0.15   0.45   5         35         5
```

Overrides: [fipy.viewers.viewer.Viewer.plot\(\)](#)

Inherited from [Viewer](#): [getVar](#)s, [setLimits](#)

10.5 Module `fipy.viewers.viewer`

Class `Viewer`

Known Subclasses: `MultiViewer`, `TSVViewer`, `GistViewer`, `GnuplotViewer`, `MatplotlibViewer`, `MayaviDistanceViewer`, `MayaviSurfactantViewer`, `MayaviViewer`

Attention!

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

Methods

`__init__(self, vars, limits=None, title=None)`

Create a `Viewer` object.

Parameters

vars: a `CellVariable` or tuple of `CellVariable` objects to plot

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.

title: displayed at the top of the `Viewer` window

`getVars(self)`

`plot(self, filename=None)`

Update the display of the viewed variables.

`setLimits(self, limits)`

Update the limits.

Parameters

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.

Bibliography

- [1] Guido van Rossum, *Python Reference Manual*. URL <http://docs.python.org/ref/>. 7
- [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>. 118, 123, 124, 126, 131, 139
- [3] Konrad Hinsen. URL http://starship.python.net/~hinsen/ScientificPython/ScientificPythonManual/Scientific_31.html. 141

Index

- AdsorbingSurfactantEquation, *see*
 - fipy.models.levelSet.surfactant.adsorbingSurfactantEquation
- BetaNoiseVariable, *see*
 - fipy.variables.betaNoiseVariable
- BoundaryCondition, *see*
 - fipy.boundaryConditions.boundaryCondition
- CellTerm, *see* fipy.terms.cellTerm
- CellVariable, *see*
 - fipy.variables.cellVariable
- Cell, *see* fipy.meshes.numMesh.cell, *see* fipy.meshes.pyMesh.cell
- CentralDifferenceConvectionTerm, *see* fipy.terms.centralDiffConvectionTerm
- ConvectionTerm, *see*
 - fipy.terms.convectionTerm
- DiffusionTerm, *see* fipy.terms.diffusionTerm
- DistanceVariable, *see* fipy.models.levelSet.distanceFunction.distanceVariable
- ExplicitDiffusionTerm, *see*
 - fipy.terms.explicitDiffusionTerm
- ExplicitNthOrderDiffusionTerm, *see*
 - fipy.terms.nthOrderDiffusionTerm
- ExplicitUpwindConvectionTerm, *see*
 - fipy.terms.explicitUpwindConvectionTerm
- ExponentialConvectionTerm, *see* fipy.terms.exponentialConvectionTerm
- ExponentialNoiseVariable, *see*
 - fipy.variables.exponentialNoiseVariable
- Face2D, *see* fipy.meshes.pyMesh.face2D
- FaceIterator, *see* fipy.meshes.meshIterator
- FaceTerm, *see* fipy.terms.faceTerm
- FaceVariable, *see*
 - fipy.variables.faceVariable
- Face, *see* fipy.meshes.numMesh.face, *see* fipy.meshes.pyMesh.face
- FixedFlux, *see*
 - fipy.boundaryConditions.fixedFlux
- FixedValue, *see* fipy.boundaryConditions.fixedValue
- GammaNoiseVariable, *see*
 - fipy.variables.gammaNoiseVariable
- GapFillMesh, *see* fipy.models.levelSet.electroChem.gapFillMesh
- GaussianNoiseVariable, *see* fipy.variables.gaussianNoiseVariable
- GmshImporter2DIn3DSpace, *see*
 - fipy.meshes.numMesh.gmshImport
- GmshImporter2D, *see*
 - fipy.meshes.numMesh.gmshImport
- GmshImporter3D, *see*
 - fipy.meshes.numMesh.gmshImport
- Grid1D, *see* fipy.meshes.numMesh.grid1D
- Grid2D, *see* fipy.meshes.numMesh.grid2D, *see* fipy.meshes.pyMesh.grid2D
- Grid3D, *see* fipy.meshes.numMesh.grid3D
- HistogramVariable, *see*
 - fipy.variables.histogramVariable
- HybridConvectionTerm, *see*
 - fipy.terms.hybridConvectionTerm
- IllConditionedPreconditionerWarning, *see*
 - fipy.solvers.solver
- ImplicitDiffusionTerm, *see*
 - fipy.terms.implicitDiffusionTerm
- ImplicitSourceTerm, *see*
 - fipy.terms.implicitSourceTerm
- LinearCGSSolver, *see*
 - fipy.solvers.linearCGSSolver
- LinearGMRESSolver, *see*
 - fipy.solvers.linearGMRESSolver
- LinearJORSolver, *see*
 - fipy.solvers.linearJORSolver
- LinearLUSolver, *see*
 - fipy.solvers.linearLUSolver
- LinearPCGSolver, *see*

- `fipy.solvers.linearPCGSolver`
- `LinearScipyCGSolver`, *see*
 - `fipy.solvers.linearScipyCGSolver`
- `LinearScipyGMRESSolver`, *see* `fipy.solvers.linearScipyGMRESSolver`
- `LinearScipyLUSolver`, *see*
 - `fipy.solvers.linearScipyLUSolver`
- `MatrixIllConditionedWarning`, *see*
 - `fipy.solvers.solver`
- `MaximumIterationWarning`, *see*
 - `fipy.solvers.solver`
- `MemoryHighWaterThread`, *see*
 - `fipy.tools.memoryLogger`
- `MemoryLogger`, *see* `fipy.tools.memoryLogger`
- `Mesh1D`, *see* `fipy.meshes.numMesh.mesh1D`
- `Mesh2D`, *see* `fipy.meshes.numMesh.mesh2D`
- `MeshAdditionError`, *see*
 - `fipy.meshes.numMesh.mesh`
- `MeshDimensionError`, *see* `fipy.viewers`
- `MeshExportError`, *see*
 - `fipy.meshes.numMesh.gmshExport`
- `MeshImportError`, *see*
 - `fipy.meshes.numMesh.gmshImport`
- `MeshIterator`, *see* `fipy.meshes.meshIterator`
- `Mesh`, *see* `fipy.meshes.common.mesh`, *see*
 - `fipy.meshes.numMesh.mesh`, *see*
 - `fipy.meshes.pyMesh.mesh`
- `ModularVariable`, *see*
 - `fipy.variables.modularVariable`
- `MultiViewer`, *see* `fipy.viewers.multiViewer`
- `NoiseVariable`, *see*
 - `fipy.variables.noiseVariable`
- `NthOrderBoundaryCondition`, *see*
 - `fipy.boundaryConditions.nthOrderBoundaryCondition`
- `NthOrderDiffusionTerm`, *see*
 - `fipy.terms.nthOrderDiffusionTerm`
- `PIDStepper`, *see* `fipy.steps.pidStepper`
- `PeriodicGrid1D`, *see* `fipy.meshes.numMesh.periodicGrid1D`
- `PeriodicGrid2DLeftRight`, *see* `fipy.meshes.numMesh.periodicGrid2D`
- `PeriodicGrid2DTopBottom`, *see* `fipy.meshes.numMesh.periodicGrid2D`
- `PeriodicGrid2D`, *see* `fipy.meshes.numMesh.periodicGrid2D`
- `PhysicalField`, *see* `fipy.tools.dimensions.physicalField`
- `PhysicalUnit`, *see* `fipy.tools.dimensions.physicalField`
- `PowerLawConvectionTerm`, *see*
 - `fipy.terms.powerLawConvectionTerm`
- `PreconditionerNotPositiveDefiniteWarning`, *see* `fipy.solvers.solver`
- `PreconditionerWarning`, *see*
 - `fipy.solvers.solver`
- `PseudoRKQSStepper`, *see*
 - `fipy.steps.pseudoRKQSStepper`
- `ScalarQuantityOutOfRangeWarning`, *see*
 - `fipy.solvers.solver`
- `SkewedGrid2D`, *see*
 - `fipy.meshes.numMesh.skewedGrid2D`
- `SolverConvergenceWarning`, *see*
 - `fipy.solvers.solver`
- `Solver`, *see* `fipy.solvers.solver`
- `SourceTerm`, *see* `fipy.terms.sourceTerm`
- `StagnatedSolverWarning`, *see*
 - `fipy.solvers.solver`
- `Stepper`, *see* `fipy.steps.stepper`
- `SurfactantEquation`, *see*
 - `fipy.models.levelSet.surfactant.surfactantEquation`
- `SurfactantVariable`, *see*
 - `fipy.models.levelSet.surfactant.surfactantVariable`
- `TSVViewer`, *see* `fipy.viewers.tsvViewer`
- `Term`, *see* `fipy.terms.term`
- `TransientTerm`, *see* `fipy.terms.transientTerm`
- `TrenchMesh`, *see* `fipy.models.levelSet.electroChem.gapFillMesh`
- `Tri2D`, *see* `fipy.meshes.numMesh.tri2D`
- `UniformGrid1D`, *see*
 - `fipy.meshes.numMesh.uniformGrid1D`
- `UniformGrid2D`, *see*
 - `fipy.meshes.numMesh.uniformGrid2D`
- `UniformGrid3D`, *see*
 - `fipy.meshes.numMesh.uniformGrid3D`
- `UniformNoiseVariable`, *see* `fipy.variables.uniformNoiseVariable`
- `UpwindConvectionTerm`, *see*
 - `fipy.terms.upwindConvectionTerm`
- `VanLeerConvectionTerm`, *see*
 - `fipy.terms.vanLeerConvectionTerm`
- `Variable`, *see* `fipy.variables.variable`
- `VectorCellVariable`, *see*
 - `fipy.variables.vectorCellVariable`

- VectorFaceVariable, *see*
 - fipy.variables.vectorFaceVariable
- Vertex, *see* fipy.meshes.pyMesh.vertex
- Viewer, *see* fipy.viewers.viewer
- fipy.__version__, 8
- fipy.boundaryConditions.
 - boundaryCondition, 9–10
 - BoundaryCondition, 9–10
 - __init__(), 9
 - __repr__(), 10
- fipy.boundaryConditions.fixedFlux, 11
 - FixedFlux, 11
 - __init__(), 11
- fipy.boundaryConditions.fixedValue, 12
 - FixedValue, 12
- fipy.boundaryConditions.
 - nthOrderBoundaryCondition, 13
 - NthOrderBoundaryCondition, 13
 - __init__(), 13
- fipy.meshes.common.mesh, 15–19
 - Mesh, 15–19
 - __add__(), 15
 - __init__(), 15
 - __mul__(), 17
 - __repr__(), 18
 - getCellCenters(), 18
 - getCellVolumes(), 18
 - getCells(), 18
 - getDim(), 18
 - getExteriorFaces(), 18
 - getFaces(), 18
 - getInteriorFaces(), 18
 - getNearestCell(), 19
 - getNumberOfCells(), 19
 - setScale(), 19
- fipy.meshes.grid1D, 20
 - Grid1D(), 20
- fipy.meshes.grid2D, 21
 - Grid2D(), 21
- fipy.meshes.grid3D, 22
 - Grid3D(), 22
- fipy.meshes.meshIterator, 23–24
 - FaceIterator, 23
 - getAreas(), 23
 - getCenters(), 23
 - MeshIterator, 23–24
 - __add__(), 23
 - __array__(), 23
 - __getitem__(), 23
 - __init__(), 23
 - __iter__(), 23
 - __len__(), 24
 - __repr__(), 24
 - __str__(), 24
 - getIDs(), 24
 - getMesh(), 24
 - where(), 24
- fipy.meshes.numMesh.cell, 25
 - Cell, 25
 - __cmp__(), 25
 - __init__(), 25
 - getCenter(), 25
 - getID(), 25
 - getMesh(), 25
 - getNormal(), 25
- fipy.meshes.numMesh.face, 26
 - Face, 26
 - __init__(), 26
 - getArea(), 26
 - getCellID(), 26
 - getCenter(), 26
 - getID(), 26
 - getMesh(), 26
- fipy.meshes.numMesh.gmshExport, 27
 - MeshExportError, 27
 - exportAsMesh(), 27
- fipy.meshes.numMesh.gmshImport, 28–33
 - GmshImporter2DIn3DSpace, 32
 - __init__(), 32
 - GmshImporter2D, 31–32
 - __init__(), 32
 - getCellVolumes(), 32
 - GmshImporter3D, 32–33
 - __init__(), 33
 - getCellVolumes(), 33
 - MeshImportError, 33
- fipy.meshes.numMesh.grid1D, 34–35
 - Grid1D, 34–35
 - __getstate__(), 34
 - __init__(), 34
 - __repr__(), 34
 - __setstate__(), 35
 - getDim(), 35
 - getFacesLeft(), 35
 - getFacesRight(), 35
 - getPhysicalShape(), 35
 - getScale(), 35
 - getShape(), 35

- fipy.meshes.numMesh.grid2D, 36–37
 - Grid2D, 36–37
 - __getstate__(), 36
 - __init__(), 36
 - __repr__(), 36
 - __setstate__(), 36
 - getFacesBottom(), 36
 - getFacesLeft(), 36
 - getFacesRight(), 37
 - getFacesTop(), 37
 - getPhysicalShape(), 37
 - getScale(), 37
 - getShape(), 37
- fipy.meshes.numMesh.grid3D, 38–39
 - Grid3D, 38–39
 - __getstate__(), 38
 - __init__(), 38
 - __repr__(), 38
 - __setstate__(), 38
 - getFacesBack(), 38
 - getFacesBottom(), 38
 - getFacesFront(), 39
 - getFacesLeft(), 39
 - getFacesRight(), 39
 - getFacesTop(), 39
 - getPhysicalShape(), 39
 - getScale(), 39
 - getShape(), 39
- fipy.meshes.numMesh.mesh1D, 44–45
 - Mesh1D, 44–45
 - __mul__(), 44
- fipy.meshes.numMesh.mesh2D, 46–47
 - Mesh2D, 46–47
 - __mul__(), 46
- fipy.meshes.numMesh.mesh, 40–43
 - MeshAdditionError, 43
 - Mesh, 40–43
 - __add__(), 40
 - __getstate__(), 42
 - __init__(), 40
 - __mul__(), 42
 - __setstate__(), 43
 - getExteriorFaces(), 43
 - getFaceCellIDs(), 43
 - getFaceCenters(), 43
 - getInteriorFaces(), 43
 - getVertexCoords(), 43
- fipy.meshes.numMesh.periodicGrid1D, 48–49
 - PeriodicGrid1D, 48–49
 - __init__(), 49
- fipy.meshes.numMesh.periodicGrid2D, 50–53
 - PeriodicGrid2DLeftRight, 51–52
 - __init__(), 52
 - PeriodicGrid2DTopBottom, 52–53
 - __init__(), 53
 - PeriodicGrid2D, 50–51
 - __init__(), 51
- fipy.meshes.numMesh.skewedGrid2D, 54–55
 - SkewedGrid2D, 54–55
 - __getstate__(), 54
 - __init__(), 54
 - __setstate__(), 54
 - getFacesBottom(), 54
 - getFacesLeft(), 54
 - getFacesRight(), 54
 - getFacesTop(), 54
 - getPhysicalShape(), 54
 - getScale(), 55
 - getShape(), 55
- fipy.meshes.numMesh.tri2D, 56–57
 - Tri2D, 56–57
 - __getstate__(), 56
 - __init__(), 56
 - __setstate__(), 56
 - getFacesBottom(), 56
 - getFacesLeft(), 57
 - getFacesRight(), 57
 - getFacesTop(), 57
 - getPhysicalShape(), 57
 - getScale(), 57
 - getShape(), 57
- fipy.meshes.numMesh.uniformGrid1D, 58–59
 - UniformGrid1D, 58–59
 - __init__(), 58
 - __mul__(), 58
 - getCellCenters(), 59
 - getCellVolumes(), 59
 - getFaceCellIDs(), 59
 - getFaceCenters(), 59
 - getInteriorFaces(), 59
 - getVertexCoords(), 59
- fipy.meshes.numMesh.uniformGrid2D, 60–61
 - UniformGrid2D, 60–61
 - __init__(), 60
 - __mul__(), 60
 - getCellCenters(), 61
 - getCellVolumes(), 61
 - getExteriorFaces(), 61

- getFaceCellIDs(), 61
 - getFaceCenters(), 61
 - getInteriorFaces(), 61
 - getVertexCoords(), 61
- fipy.meshes.numMesh.uniformGrid3D, 62–63
 - UniformGrid3D, 62–63
 - __init__(), 62
 - __mul__(), 62
 - getCellCenters(), 63
 - getCellVolumes(), 63
 - getExteriorFaces(), 63
 - getFaceCellIDs(), 63
 - getFaceCenters(), 63
 - getInteriorFaces(), 63
 - getVertexCoords(), 63
- fipy.meshes.pyMesh.cell, 64
 - Cell, 64
 - __init__(), 64
 - __repr__(), 64
 - getBoundingCells(), 64
 - getCenter(), 64
 - getFaceIDs(), 64
 - getFaceOrientations(), 64
 - getFaces(), 64
 - getID(), 64
 - getVolume(), 64
- fipy.meshes.pyMesh.face2D, 67
 - Face2D, 67
- fipy.meshes.pyMesh.face, 65–66
 - Face, 65–66
 - __init__(), 65
 - __repr__(), 65
 - addBoundingCell(), 65
 - getArea(), 65
 - getCellDistance(), 65
 - getCellID(), 65
 - getCells(), 65
 - getCenter(), 65
 - getID(), 66
 - getNormal(), 66
- fipy.meshes.pyMesh.grid2D, 68–70
 - Grid2D, 68–70
 - __init__(), 69
 - getCellCenters(), 69
 - getCellVolumes(), 69
 - getFacesBottom(), 69
 - getFacesLeft(), 69
 - getFacesRight(), 70
 - getFacesTop(), 70
 - getPhysicalShape(), 70
 - getShape(), 70
- fipy.meshes.pyMesh.mesh, 71
 - Mesh, 71
 - __init__(), 71
 - getExteriorFaces(), 71
 - getFaceOrientations(), 71
 - getPhysicalShape(), 71
 - getScale(), 71
 - setScale(), 71
- fipy.meshes.pyMesh.vertex, 72
 - Vertex, 72
 - __init__(), 72
 - __repr__(), 72
 - getCoordinates(), 72
- fipy.models.levelSet.advection.
 - advectionEquation, 73
 - buildAdvectionEquation(), 73
- fipy.models.levelSet.advection.
 - higherOrderAdvectionEquation, 74
 - buildHigherOrderAdvectionEquation(), 74
- fipy.models.levelSet.distanceFunction.
 - distanceVariable, 75–79
 - DistanceVariable, 75–79
 - __init__(), 77
 - calcDistanceFunction(), 77
 - extendVariable(), 78
 - getCellInterfaceAreas(), 78
- fipy.models.levelSet.electroChem.
 - gapFillMesh, 80–83
 - GapFillMesh, 80–81
 - __getstate__(), 81
 - __init__(), 81
 - __setstate__(), 81
 - buildTransitionMesh(), 81
 - getBottomFaces(), 81
 - getCellIDsAboveFineRegion(), 81
 - getFineMesh(), 81
 - getTopFaces(), 81
 - TrenchMesh, 81–83
 - __getstate__(), 83
 - __init__(), 83
 - __setstate__(), 83
 - getElectrolyteMask(), 83
- fipy.models.levelSet.electroChem.
 - metalIonDiffusionEquation, 84–85
 - buildMetalIonDiffusionEquation(), 84
- fipy.models.levelSet.surfactant.
 - adsorbingSurfactantEquation, 86–90

- AdsorbingSurfactantEquation, 86–90
 - __init__(), 89
 - solve(), 90
 - sweep(), 90
- fipy.models.levelSet.surfactant.
 - surfactantBulkDiffusionEquation, 91
 - buildSurfactantBulkDiffusionEquation(), 91
- fipy.models.levelSet.surfactant.
 - surfactantEquation, 92–93
 - SurfactantEquation, 92–93
 - __init__(), 92
 - solve(), 92
 - sweep(), 92
- fipy.models.levelSet.surfactant.
 - surfactantVariable, 94–96
 - SurfactantVariable, 94–96
 - __init__(), 94
 - copy(), 95
 - getInterfaceVar(), 95
- fipy.solvers.linearCGSSolver, 97
 - LinearCGSSolver, 97
- fipy.solvers.linearGMRESSolver, 98
 - LinearGMRESSolver, 98
- fipy.solvers.linearJORSolver, 99
 - LinearJORSolver, 99
 - __init__(), 99
- fipy.solvers.linearLUSolver, 100
 - LinearLUSolver, 100
 - __init__(), 100
- fipy.solvers.linearPCGSolver, 101
 - LinearPCGSolver, 101
- fipy.solvers.linearScipyCGSolver, 102
 - LinearScipyCGSolver, 102
- fipy.solvers.linearScipyGMRESSolver, 103
 - LinearScipyGMRESSolver, 103
- fipy.solvers.linearScipyLUSolver, 104
 - LinearScipyLUSolver, 104
 - __init__(), 104
- fipy.solvers.solver, 105–109
 - IllConditionedPreconditionerWarning, 105
 - __str__(), 105
 - MatrixIllConditionedWarning, 105–106
 - __str__(), 106
 - MaximumIterationWarning, 106
 - __str__(), 106
 - PreconditionerNotPositiveDefiniteWarning, 106–107
 - __str__(), 107
 - PreconditionerWarning, 107
 - ScalarQuantityOutOfRangeWarning, 107
 - __str__(), 107
 - SolverConvergenceWarning, 108
 - __init__(), 108
 - __str__(), 108
 - Solver, 107–108
 - __init__(), 108
 - __repr__(), 108
 - StagnatedSolverWarning, 108–109
 - __str__(), 109
- fipy.steppers.pidStepper, 114
 - PIDStepper, 114
 - __init__(), 114
- fipy.steppers.pseudoRKQSStepper, 115
 - PseudoRKQSStepper, 115
 - __init__(), 115
- fipy.steppers.stepper, 116
 - Stepper, 116
 - __init__(), 116
 - failFn(), 116
 - step(), 116
 - successFn(), 116
 - sweepFn(), 116
- fipy.terms.cellTerm, 117
 - CellTerm, 117
 - __init__(), 117
- fipy.terms.centralDiffConvectionTerm, 118
 - CentralDifferenceConvectionTerm, 118
- fipy.terms.convectionTerm, 119–120
 - ConvectionTerm, 119–120
 - __init__(), 119
 - __neg__(), 120
- fipy.terms.diffusionTerm, 121
 - DiffusionTerm, 121
 - __init__(), 121
 - __neg__(), 121
- fipy.terms.explicitDiffusionTerm, 122
 - ExplicitDiffusionTerm, 122
- fipy.terms.explicitUpwindConvectionTerm, 123
 - ExplicitUpwindConvectionTerm, 123
- fipy.terms.exponentialConvectionTerm, 124
 - ExponentialConvectionTerm, 124
- fipy.terms.faceTerm, 125
 - FaceTerm, 125

- __init__(), 125
- fipy.terms.hybridConvectionTerm, 126
 - HybridConvectionTerm, 126
- fipy.terms.implicitDiffusionTerm, 127
 - ImplicitDiffusionTerm, 127
- fipy.terms.implicitSourceTerm, 128
 - ImplicitSourceTerm, 128
- fipy.terms.nthOrderDiffusionTerm, 129–130
 - ExplicitNthOrderDiffusionTerm, 129
 - __init__(), 129
 - NthOrderDiffusionTerm, 129–130
 - __init__(), 129
- fipy.terms.powerLawConvectionTerm, 131
 - PowerLawConvectionTerm, 131
- fipy.terms.sourceTerm, 132
 - SourceTerm, 132
 - __init__(), 132
- fipy.terms.term, 133–136
 - Term, 133–136
 - __add__(), 133, 134
 - __eq__(), 133
 - __init__(), 133
 - __neg__(), 134
 - __pos__(), 134
 - __repr__(), 134
 - __rsub__(), 134
 - __sub__(), 134
 - cacheMatrix(), 134
 - cacheRHSvector(), 135
 - getMatrix(), 135
 - getRHSvector(), 135
 - justResidualVector(), 135
 - solve(), 135
 - sweep(), 135
- fipy.terms.transientTerm, 137–138
 - TransientTerm, 137–138
- fipy.terms.upwindConvectionTerm, 139
 - UpwindConvectionTerm, 139
- fipy.terms.vanLeerConvectionTerm, 140
 - VanLeerConvectionTerm, 140
- fipy.tools.dimensions.physicalField, 141–162
 - PhysicalField, 143–157
 - __abs__(), 144
 - __add__(), 145, 148
 - __array__(), 145
 - __div__(), 145
 - __eq__(), 146
 - __float__(), 146
 - __ge__(), 146
 - __getitem__(), 146
 - __gt__(), 146
 - __init__(), 144
 - __le__(), 147
 - __len__(), 147
 - __lt__(), 147
 - __mod__(), 147
 - __mul__(), 147, 148
 - __ne__(), 148
 - __neg__(), 148
 - __nonzero__(), 148
 - __pos__(), 148
 - __pow__(), 148
 - __rdiv__(), 148
 - __repr__(), 148
 - __rpow__(), 149
 - __rsub__(), 149
 - __setitem__(), 149
 - __str__(), 149
 - __sub__(), 149
 - allclose(), 150
 - allequal(), 150
 - arccosh(), 150
 - arccos(), 150
 - arcsin(), 150
 - arctan2(), 151
 - arctanh(), 151
 - arctan(), 150
 - ceil(), 151
 - conjugate(), 151
 - convertToUnit(), 151
 - copy(), 152
 - cosh(), 152
 - cos(), 152
 - dot(), 153
 - floor(), 153
 - getNumericValue(), 153
 - getShape(), 153
 - getUnit(), 153
 - inBaseUnits(), 153
 - inSIUnits(), 153
 - inUnitsOf(), 153
 - isCompatible(), 154
 - log10(), 154
 - log(), 154
 - put(), 154
 - reshape(), 155
 - sinh(), 155

- sin(), 155
- sqrt(), 155
- sum(), 156
- take(), 156
- tanh(), 156
- tan(), 156
- tostring(), 157
- PhysicalUnit, 157–162
 - __cmp__(), 157
 - __div__(), 157
 - __init__(), 157
 - __mul__(), 158, 159
 - __pow__(), 159
 - __rdiv__(), 159
 - __repr__(), 159, 160
 - conversionFactorTo(), 160
 - conversionTupleTo(), 161
 - isAngle(), 161
 - isCompatible(), 161
 - isDimensionlessOrAngle(), 161
 - isDimensionless(), 161
 - name(), 162
 - setName(), 162
- fipy.tools.dump, 163
 - read(), 163
 - write(), 163
- fipy.tools.memoryLogger, 164–165
 - MemoryHighWaterThread, 164
 - __init__(), 164
 - run(), 164
 - stop(), 164
 - MemoryLogger, 164–165
 - __del__(), 164
 - __init__(), 164
 - start(), 165
 - stop(), 165
- fipy.tools.numerix, 166–177
 - L1norm(), 172
 - L2norm(), 172
 - LINFnorm(), 173
 - allclose(), 166
 - allequal(), 166
 - arccosh(), 167
 - arccos(), 166
 - arcsinh(), 168
 - arcsin(), 167
 - arctan2(), 168
 - arctanh(), 169
 - arctan(), 168
 - ceil(), 169
 - conjugate(), 169
 - cosh(), 170
 - cos(), 170
 - crossProd(), 170
 - dot(), 170
 - exp(), 171
 - floor(), 171
 - getShape(), 171
 - getTypecode(), 171
 - indices(), 172
 - isFloat(), 172
 - isInt(), 172
 - log10(), 173
 - log(), 173
 - max(), 173
 - min(), 174
 - nonzero(), 174
 - ones(), 174
 - put(), 174
 - reshape(), 175
 - sinh(), 175
 - sin(), 175
 - sqrtDot(), 176
 - sqrt(), 175
 - sum(), 176
 - take(), 176
 - tanh(), 176
 - tan(), 176
 - tostring(), 176
 - zeros(), 177
- fipy.tools.parser, 178
 - parse(), 178
- fipy.tools.vector, 179
 - crossProd(), 179
 - prune(), 179
 - putAdd(), 179
 - sqrtDot(), 179
- fipy.variables.betaNoiseVariable, 181–183
 - BetaNoiseVariable, 181–183
 - __init__(), 183
- fipy.variables.cellVariable, 184–189
 - CellVariable, 184–189
 - __call__(), 184
 - __getstate__(), 185
 - __init__(), 184
 - __setstate__(), 185
 - copy(), 185
 - getArithmeticFaceValue(), 185

- getCellVolumeAverage(), 186
- getFaceGradAverage(), 186
- getFaceGrad(), 186
- getFaceValue(), 186
- getGaussGrad(), 186
- getGrad(), 186
- getHarmonicFaceValue(), 187
- getLeastSquaresGrad(), 187
- getOld(), 188
- getValue(), 188
- setValue(), 188
- updateOld(), 189
- fipy.variables.exponentialNoiseVariable, 190–192
 - ExponentialNoiseVariable, 190–192
 - __init__(), 191
- fipy.variables.faceVariable, 193
 - FaceVariable, 193
 - __init__(), 193
 - setValue(), 193
- fipy.variables.gammaNoiseVariable, 194–196
 - GammaNoiseVariable, 194–196
 - __init__(), 195
- fipy.variables.gaussianNoiseVariable, 197–199
 - GaussianNoiseVariable, 197–199
 - __init__(), 199
- fipy.variables.histogramVariable, 200–202
 - HistogramVariable, 200–202
 - __init__(), 200
- fipy.variables.modularVariable, 203–205
 - ModularVariable, 203–205
 - __init__(), 203
 - __rsub__(), 204
 - __sub__(), 204
 - getArithmeticFaceValue(), 204
 - getFaceGradNoMod(), 204
 - getFaceGrad(), 204
 - getGrad(), 204
 - updateOld(), 204
- fipy.variables.noiseVariable, 206–207
 - NoiseVariable, 206–207
 - __init__(), 206
 - copy(), 206
 - scramble(), 206
- fipy.variables.uniformNoiseVariable, 208–210
 - UniformNoiseVariable, 208–210
 - __init__(), 209
- fipy.variables.variable, 211–222
 - Variable, 211–222
 - __abs__(), 211
 - __add__(), 211, 216
 - __and__(), 212
 - __array__(), 212
 - __array_wrap__(), 212
 - __call__(), 213
 - __div__(), 213
 - __eq__(), 213
 - __float__(), 213
 - __ge__(), 213
 - __getitem__(), 214
 - __gt__(), 214
 - __init__(), 211
 - __le__(), 214
 - __len__(), 214
 - __lt__(), 214
 - __mod__(), 215
 - __mul__(), 215, 216
 - __ne__(), 215
 - __neg__(), 215
 - __new__(), 222
 - __or__(), 215
 - __pos__(), 216
 - __pow__(), 216
 - __rdiv__(), 216
 - __repr__(), 216
 - __rpow__(), 216
 - __rsub__(), 216
 - __setitem__(), 216
 - __str__(), 216
 - __sub__(), 216
 - allclose(), 217
 - allequal(), 217
 - arccosh(), 217
 - arccos(), 217
 - arcsinh(), 217
 - arcsin(), 217
 - arctan2(), 217
 - arctanh(), 217
 - arctan(), 217
 - cacheMe(), 218
 - ceil(), 218
 - conjugate(), 218
 - copy(), 218
 - cosh(), 218
 - cos(), 218
 - dontCacheMe(), 218

- dot(), 218
- exp(), 218
- floor(), 219
- getMag(), 219
- getMesh(), 219
- getName(), 219
- getNumericValue(), 219
- getShape(), 219
- getSubscribedVariables(), 219
- getTypecode(), 220
- getUnit(), 220
- getValue(), 220
- inBaseUnits(), 220
- inUnitsOf(), 220
- log10(), 221
- log(), 221
- put(), 221
- reshape(), 221
- setName(), 221
- setValue(), 221
- sinh(), 222
- sin(), 221
- sqrt(), 222
- sum(), 222
- take(), 222
- tanh(), 222
- tan(), 222
- tostring(), 222
- transpose(), 222
- fipy.variables.vectorCellVariable, 223–225
 - VectorCellVariable, 223–225
 - __call__(), 223
 - __init__(), 223
 - dot(), 223
 - getArithmeticFaceValue(), 223
 - getDivergence(), 224
 - getHarmonicFaceValue(), 224
- fipy.variables.vectorFaceVariable, 226–227
 - VectorFaceVariable, 226–227
 - __call__(), 226
 - __init__(), 226
 - dot(), 226
 - getDivergence(), 226
- fipy.viewers.multiViewer, 231
 - MultiViewer, 231
 - __init__(), 231
 - getViewers(), 231
 - plot(), 231
 - setLimits(), 231
- fipy.viewers.tsvViewer, 232–233
 - TSVViewer, 232–233
 - __init__(), 232
 - plot(), 232
- fipy.viewers.viewer, 234
 - Viewer, 234
 - __init__(), 234
 - getVars(), 234
 - plot(), 234
 - setLimits(), 234
- fipy.steppers, 110–113
 - L1error(), 111
 - L2error(), 112
 - LINFerror(), 112
 - error(), 111
 - residual(), 112
 - sweepMonotonic(), 113
- fipy.viewers, 228–230
 - MeshDimensionError, 229–230
 - make(), 229