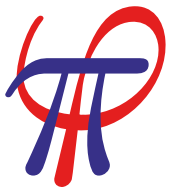


FiPy

User's Guide

Daniel Wheeler
Jonathan E. Guyer
James A. Warren

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

June 11, 2009

Version 2.0.2

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

Contents

Part I

Introduction

Introduction Contents

Overview

FiPy is an object oriented, partial differential equation (PDE) solver, written in Python [1], based on a standard finite volume (FV) approach. The framework has been developed in the Metallurgy Division and Center for Theoretical and Computational Materials Science (CTCMS), in the Materials Science and Engineering Laboratory (MSEL) at the National Institute of Standards and Technology (NIST).

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

The FiPy framework includes terms for transient diffusion, convection and standard sources, enabling the solution of arbitrary combinations of coupled elliptic, hyperbolic and parabolic PDEs. Currently implemented models include phase field [2, 3, 4] treatments of polycrystalline, dendritic, and electrochemical phase transformations as well as a level set treatment of the electrodeposition process [5].

The latest information about FiPy can be found at http://www.ctcms.nist.gov/fipy/.

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

...please read Chapter 2 "Installation and Usage" and Chapter 5 "Frequently Asked Questions".

1.2 What's new in version 2.0.2?

Warning

FiPy 2 brings unavoidable syntax changes. Please see Example 12.1 for guidance on the changes that you will need to make to your FiPy 1.x scripts.

The significant changes since version 1.2 are:

- `CellVariable` and `FaceVariable` objects can hold values of any rank.
- Much simpler syntax for specifying `Cells` for initial conditions and `Faces` for boundary conditions.

- Automated determination of the Peclet number and partitioning of `ImplicitSourceTerm` coefficients between the matrix diagonal and the right-hand-side-vector.
- Simplified `Viewer` syntax.
- Support for the Trilinos solvers.
- Support for anisotropic diffusion coefficients.

1.3 Download and Installation

Please refer to Chapter 2 "Installation and Usage" for details on download and installation. FiPy can be redistributed and/or modified freely, provided that any derivative works bear some notice that they are derived from it, and any modified versions bear some notice that they have been modified.

1.4 Support

You can communicate with the FiPy developers and with other users via our mailing list and we welcome you to use the tracking system for bugs, support requests, feature requests and patch submissions [6, 7]. We welcome collaborative efforts on this project.

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

1.5 Conventions and Notation

FiPy is driven by Python script files than you can view or modify in any text editor. FiPy sessions are invoked from a command-line shell, such as `tcsh` or `bash`.

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

```
$ like this
```

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

Text blocks of the form:

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

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

Warning

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

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

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

will correspond to, *e.g.*:

```
/some/where/FiPy-1.0/examples/diffusion/steadyState/mesh1D/input.py
```

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

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

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

We may at times use a

Note

to indicate something that may be of interest

or a

Warning

to indicate something that could cause serious problems.

1.6 Mailing List

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

To: `listproc@nist.gov`
Subject: *(optional)*
Body: `subscribe fipy` *Your Name*

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

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

List Archive

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

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

Installation and Usage

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

Note

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

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

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

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

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

2.1 Shortcuts

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

EasyInstall

http://peak.telecommunity.com/DevCenter/EasyInstall

If Python and setuptools are already available, a minimally functional installation of FiPy can be obtained by issuing the commands:

```
$ easy_install numpy
$ easy_install matplotlib
$ easy_install pysparse
$ easy_install fipy
```

Some of the Optional Packages may also be available via `easy_install`, but not all are.

Enthought Python Edition

http://www.enthought.com/epd

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

Attention!

PySparse and FiPy are not presently included in EPD, so you will need to separately install them, either manually or via EasyInstall.

Python(x,y)

http://www.pythonxy.com/

Another comprehensive Python package installer for scientific applications, presently only available for Windows. See Section 2.10 "Windows Installation" for more information.

2.2 Privileges

If you do not have administrative privileges on your computer, or if for any reason you don't want to tamper with your existing Python installation, most packages (including FiPy) will allow you to install to an alternate location. Instead of installing these packages with `python setup.py install`, you would use `python setup.py install --home=<dir>`, where `<dir>` is the desired installation directory (usually "~" to indicate your home directory). You will then need to append `<dir>/lib/python` to your `PYTHONPATH` environment variable. See the Alternate Installation section of the Python document "Installing Python Modules" [8] for more information, such as circumstances in which you should use `--prefix` instead of `--home`.

2.3 Prerequisites

Operating System

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

Note

Simple instructions for Mac OS X users are in Section 2.9 "Mac OS X Installation". Simple instructions for Windows users are in Section 2.10 "Windows Installation".

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

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

Required Packages

Warning

FiPy will not run if the following items are not installed.

Python

http://www.python.org/

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

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

If necessary, you can download and install it for your platform [9].

Note

FiPy requires at least version 2.3 of Python. FiPy has not yet been tested (and will almost certainly not work) with Python 3.0.

NumPy

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

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

Attention!

FiPy no longer uses the older Numeric or numarray packages.

PySparse

http://pysparse.sourceforge.net

FiPy requires Roman Geus' PySparse package.

You can download the PySparse archive or check it out via anonymous CVS download:

```
$ cvs -d:pserver:anonymous@pysparse.cvs.sourceforge.net:/cvsroot/pysparse login
```

and press enter at the password prompt, then:

```
$ cvs -z3 -d:pserver:anonymous@pysparse.cvs.sourceforge.net:/cvsroot/pysparse \
>     checkout pysparse
```

From within the `pysparse` base directory, follow its included instructions for building PySparse for your platform. PySparse Windows installers are available.

Note

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

Warning

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

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

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

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

Warning

FiPy requires version 1.0 or higher of PySparse.

Viewers

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

Matplotlib

http://matplotlib.sourceforge.net

Matplotlib is a Python package that displays publication quality results. It displays both 1D X-Y type plots and 2D contour plots for structured data. It does not display unstructured 2D data or 3D data. It works on all common platforms and produces publication quality hard copies. Version 0.72.1 or higher is required. Matplotlib installers for specific platforms are available [10].

Note

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

Gnuplot-py

http://gnuplot-py.sourceforge.net

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

Pygist

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

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

Attention!

Pygist requires the old Numeric module to be installed.

Warning

The facility to produce hard copies in Pygist does not work very well and may crash the FiPy run. "`.eps`" and "`.cgm`" export seem to work.

Note

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

Note

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

MayaVi

http://mayavi.sourceforge.net

The MayaVi 1 Data Visualizer is a free, easy to use scientific data visualizer. It displays 1D, 2D and 3D data. It is the only FiPy viewer available for 3D data. Other viewers are probably better for 1D or 2D viewing.

Warning

FiPy can only use MayaVi 1 to display 3D meshes consisting entirely of tetrahedrons or wedge elements. The ordering of vertices for other mesh types may not work.

Note

Is is also necessary to install the PyVTK package to use the FiPy MayaVi viewers.

Note

MayaVi 1 is outdated and we hope to have compatilibiltiy with MayaVi 2 as soon as possible.

2.4 Obtaining FiPy

FiPy is freely available for download via Subversion or as a compressed archive [11]. To obtain FiPy via anonymous Subversion, issue the following command:

```
$ svn checkout http://matforge.org/svn/fipy/tags/CURRENT
```

This will download a fairly stable version of FiPy (somewhere between version 2.0.2 and HEAD). If you prefer, you can download FiPy version 2.0.2 (recommended) with:

```
$ svn checkout http://matforge.org/svn/fipy/tags/STABLE
```

Further information about Subversion can be found in Section 2.11 and in the online Subversion Red Bean book [12].

Warning

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

Manual

You can download the latest manual [13]. Alternatively, it may be possible to build a fresh copy by issuing the following command in the base directory:

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

Note

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

2.5 Testing FiPy

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

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

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

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

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

A few tests will fail the first time as a result of the messages output in the course of caching the compiled inline code, but a repeat test should have no failures (although see "`repairing catalog by removing key`" in Chapter 5 "Frequently Asked Questions").

Note

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

```
$ setenv PYTHONPATH .:${PYTHONPATH}
```

or:

```
$ export PYTHONPATH=.:${PYTHONPATH}
```

2.6 Installing FiPy

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

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

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

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

2.7 Using FiPy

To see examples of problems that FiPy is capable of solving, you can run any of the scripts in Part II "Examples".

Note

We strongly recommend you proceed through Part II "Examples", but at the very least work through Example 6.1 "Module examples.diffusion.mesh1D" to understand the notation and basic concepts of FiPy.

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

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

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

Note

From within IPython, you would type:

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

In order to customize the examples, or to develop your own scripts, some knowledge of Python syntax is required. We recommend you familiarize yourself with the excellent Python tutorial [14] or with Dive Into Python [15].

As you gain experience, you will want to see FAQ 5.7 to learn about the flags and environment variables that modify FiPy's default behavior.

2.8 Optional Packages

Note

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

SciPy

http://www.scipy.org/

Significantly improved performance has been achieved with the judicious use of C language inlining, via the weave module of the SciPy package. SciPy download instructions [16] are available. We recommend version 0.5.2 or greater.

Note

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

gmsh

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

It is possible to create irregular meshes with this package.

Note

The Mac OS X distribution of gmsh provides a nice graphical tool, but is structured a bit differently than on other platforms. To access the underlying shell tool, create a shell alias called `gmsh` that corresponds to `<Gmsh path>/Gmsh.app/Contents/MacOS/Gmsh`.

IPython

http://ipython.scipy.org/

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

Trilinos

http://trilinos.sandia.gov

Trilinos provides solvers and preconditioners, and can be used instead of PySparse. Trilinos preconditioning allows for iterative solutions to some difficult problems that PySparse cannot solve.

Attention!

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

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

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

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

Note

Trilinos can be installed in a non-standard location by adding the `--prefix=$LOCAL_INSTALLATION_DIR` flag to the configure step. If Trilinos is installed in a nonstandard location, the path to the PyTrilinos site-packages directory should be added to the `PYTHONPATH` environment variable; this should be of the form `$INSTALL_DIR/lib/$PYTHON_VERSION/site-packages/`. Also, the path to the Trilinos lib directory should be added to the `LD_LIBRARY_PATH` (on Linux) or `DYLD_LIBRARY_PATH` (on Mac OS X) environment variable; this should be of the form `$INSTALL_DIR/lib`.

Note

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

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

Note

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

`Gen_Prolongator warning : Max eigen <= 0.0`: which is not significant to FiPy.

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

`Aztec status AZ_ill_cond: GMRES hessenberg ill-conditioned`: which indicates that GMRES is having trouble with the problem, and may indicate that trying a different solver or preconditioner may give more accurate results if GMRES fails.

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

2.9 Mac OS X Installation

We present four comparatively simple routes to installing FiPy on Mac OS X. The Fink Installation procedure is appropriate if you are already familiar with the Fink package manager. Either EasyInstall or Enthought Python Edition will get FiPy running in a minumum number of steps. The Binary Installation procedure is the next most expedient if you have never heard of Fink or if you are not comfortable with it. Please see the more general Chapter 2 "Installation and Usage" for detailed installation instructions. These instructions are not the only ways to set up FiPy on Mac OS X but represent the most expedient ways, from our experience, to have a usable installation up and running.

Attention!

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

EasyInstall

http://peak.telecommunity.com/DevCenter/EasyInstall

Note

Several packages installed with *easy_install* will need be automatically compiled, which requires that you install the Xcode Development Tools before proceding.

If Python and setuptools are already available, a minimally functional installation of FiPy can be obtained by issuing the commands:

```
$ easy_install numpy
$ easy_install matplotlib
$ easy_install pysparse
$ easy_install fipy
```

Some of the Optional Packages may also be available via `easy_install`, but not all are.

Enthought Python Edition

http://www.enthought.com/epd

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

Attention!

PySparse and FiPy are not presently included in EPD, so you will need to separately install them, either manually or via EasyInstall.

Binary Installation

Attention!

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

Note

If you wish to view 3D models, the MayaVi viewer is required, which is best installed via Fink, so you should follow the Fink Installation instructions.

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

Python

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

Note

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

Note

Another option is ActivePython, which probably is the most heavily supported installation on the Mac, but seems to lack *readline* support, but these instructions

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

worked for us.

NumPy

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

matplotlib

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

matplotlib requires:

wxPython

Download and install the latest version of `wxPythonX.X-osx-unicode-X.X.X.X-universal10.4-py2.4.dmg` from pythonmac.

PySparse

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

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

FiPy

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

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

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

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

Note

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

Fink Installation

Attention!

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

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

Xcode Development Tools

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

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

Note

Free registration with the Apple Developer Connection is required.

X11

Open the X11 application.

Set your `$DISPLAY` environment variable to `:0.0`.

Note

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

Fink

http://fink.sourceforge.net/download

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

Note

The following steps have been tested with Fink 0.8.1 on Mac OS X 10.4 "Tiger". Variations may be necessary for other OS versions.

unstable tree

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

Note

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

Note

"unstable" is not as scary as it sounds. The Fink administrators tend to be very conservative about what packages are designated "stable".

Remaining Fink packages

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

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

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

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

Attention!

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

Note

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

A few changes are needed to allow matplotlib to run:

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

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

```
backend      : TkAgg
```

or you can install wxPython with:

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

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

PySparse installation

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

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

From within the PySparse directory, execute:

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

FiPy installation

Install FiPy packages as explained above.

Note

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

Optional Packages

IPython

http://ipython.scipy.org/

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

Gmsh

http://www.geuz.org/gmsh

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

Note

This is a required package for superfill examples.

MayaVi

MayaVi 1 is a requirement if you wish to view 3D problems or improve the viewing capabilities of the superfill examples. This is one package that is probably much easier to install via Fink than by hand. You might attempt to follow the instructions at http://mayavi.sourceforge.net/mwiki/BuildingVTKOnOSX but they are quite out of date and did not work for us.

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

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

Note

MayaVi 1 is outdated and we hope to have compatilibiltiy with MayaVi 2 as soon as possible.

SciPy

http://www.scipy.org/

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

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

    Warning

    We do *not* recommend installing the other components from the ScipySuperpack. In particular, matplotlib was not usable when we tried it.
  - pythonmac includes a build of SciPy, but the latest version we tried from pythonmac, `scipy-0.5.1-py2.4-macosx10.4.mpkg.zip (MD5: 15daecd1b5709f04a41154102269359f`) was apparently not linked correctly and does not work properly, c.f.

    http://projects.scipy.org/pipermail/scipy-user/2007-January/010820.html
  - We *may* provide builds of SciPy from our own site if we conclude that we can better serve FiPy users that way.
- If you followed the Fink Installation procedure, then you should be able to type:

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

Note

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

Using FiPy on Mac OS X

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

IDLE Environment

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

Note

We are not aware of a Fink package for IDLE.

Attention!

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

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

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

The script will open in an editor window.

- Choose Run > Run Module

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

2.10 Windows Installation

These instructions are for the Windows XP and Windows 2000 platforms. Please see the more general Chapter 2 "Installation and Usage" for detailed installation instructions. These instructions are not the only way to set up FiPy on a Windows OS but represent the most expedient way from our experience to have a usable installation up and running.

Required Packages

Python

http://www.pythonxy.com

http://www.enthought.com

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

PySparse

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

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

FiPy

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

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

Optional Packages

Gmsh

http://www.geuz.org/gmsh

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

PyVTK

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

If you wish to use MayaVi 1, PyVTK is a requirement. Download and install the latest version from the FiPy downloads page (`PyVTK-x.y.z.win32.exe`).

MayaVi

http://mayavi.sourceforge.net

MayaVi 1 is a requirement if you wish to view 3D problems or improve the viewing capabilities of the superfill examples. Download the source code and run `python setup.py install` to install from the python source in order for `import mayavi` to work at the command line.

Warning

At the time of writing the `import mayavi` command is not working with the python(x,y) version of python.

Note

MayaVi 1 is outdated and we hope to have compatilibiltiy with MayaVi 2 as soon as possible.

Using FiPy on Windows

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

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

Testing

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

```
>>> run setup.py test
```

in the IPython shell.

Running Examples

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

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

2.11 Subversion tags

All stages of FiPy development are archived in a Subversion (SVN) repository at MatForge. You can browse through the code at http://matforge.org/fipy/browser and, using an SVN client, you can download various tagged revisions of FiPy depending on your needs.

Attention!

Be sure to follow Chapter 2 "Installation and Usage" to obtain all the prerequisites for FiPy.

2.12 SVN client

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

Mac OS X client

You can obtain a binary installer of `svn` from

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

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

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

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

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

2.13 SVN tags

In general, most users will not want to download the very latest state of FiPy, as these files are subject to active development and may not behave as desired. Most users will not be interested in particular version numbers either, but instead with the degree of code stability. Different "tracking tags" are used to indicate different stages of FiPy development. You will need to decide on your own risk tolerance when deciding which stage of development to track.

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

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

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

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

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

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

In addition:

`tags/version-x_y` designates a released version x.y. Any released version of FiPy will be designated with a fixed tag: The current version of FiPy is 2.0.2.

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

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

Note

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

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

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

http://matforge.org/fipy/build

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

For those who are interested in learning more about Subversion, the canonical manual is the online Subversion Red Bean book [12].

Theoretical and Numerical Background

This chapter describes the numerical methods used to solve equations in the FiPy programming environment. FiPy uses the finite volume method (FVM) to solve coupled sets of partial differential equations (PDEs). For a good introduction to the FVM see Nick Croft's PhD thesis [17], Patanker [18] or Versteek and Malalasekera [19].

Essentially, the FVM consists of dividing the solution domain into discrete finite volumes over which the state variables are approximated with linear or higher order interpolations. The derivatives in each term of the equation are satisfied with simple approximate interpolations in a process known as discretization. The (FVM) is a popular discretization technique employed to solve coupled PDEs used in many application areas (*e.g.* Fluid Dynamics).

The FVM can be thought of as a subset of the Finite Element Method (FEM), just as the Finite Difference Method (FDM) is a subset of the FVM. A system of equations fully equivalent to the FVM can be obtained with the FEM using as weighting functions the characteristic functions of FV cells, i.e., functions equal to unity [20]. Analogously, the the discretization of equations with the FVM reduces to the FDM on Cartesian grids.

3.1 General Conservation Equation

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

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

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

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

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

where ρ, $\vec{u}$ and Γ_i represent coefficients in the transient, convection and diffusion terms, respectively. These coefficients can be arbitrary functions of any parameters or variables in the system. The variable ϕ represents

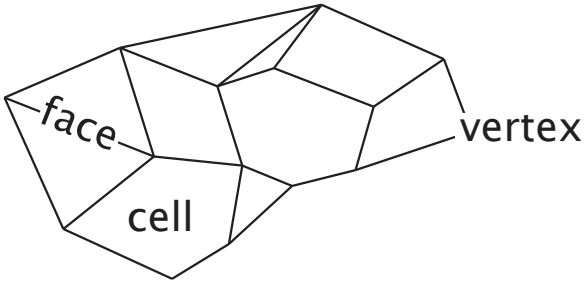


Figure 3.1: A mesh consists of cells, faces and vertices. For the purposes of FiPy, the divider between two cells is known as a face for all dimensions.

the unknown quantity in the equation. The diffusion term can represent any higher order diffusion-like term, where the order is given by the exponent n. For example, the diffusion term can represent conventional Fickean diffusion [*i.e.*, $\nabla\cdot(\Gamma\nabla\phi)$] when the exponent $n = 1$ or a Cahn-Hilliard term [*i.e.*, $\nabla\cdot(\Gamma_1\nabla[\nabla\cdot\Gamma_2\nabla\phi)]$] [21, 22, 23]] when $n = 2$. Of course, higher order terms ($n > 2$) are also possible.

3.2 Finite Volume Method

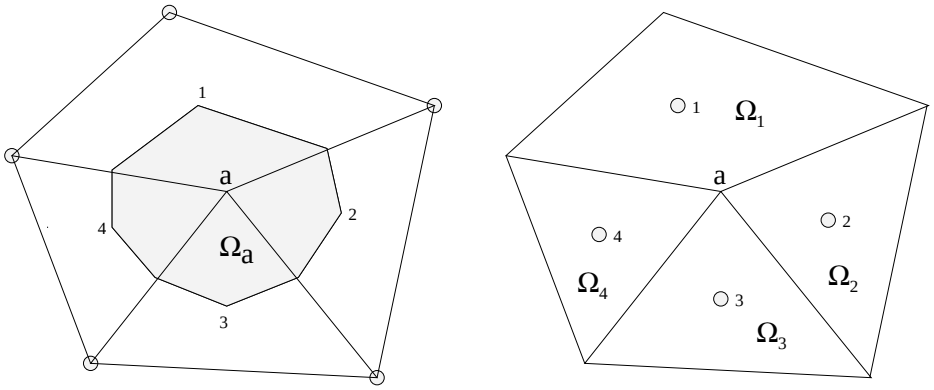
To use the FVM, the solution domain must first be divided into non-overlapping polyhedral elements or cells. A solution domain divided in such a way is generally known as a mesh (as we will see, a `Mesh` is also a FiPy object). A mesh consists of vertices, faces and cells (see Figure 3.1). In the FVM the variables of interest are averaged over control volumes (CVs). The CVs are either defined by the cells or are centered on the vertices.

Cell Centered FVM (CC-FVM)

In the CC-FVM the CVs are formed by the mesh cells with the cell center "storing" the average variable value in the CV, (see Figure 3.2). The face fluxes are approximated using the variable values in the two adjacent cells surrounding the face. This low order approximation has the advantage of being efficient and requiring matrices of low band width (the band width is equal to the number of cell neighbors plus one) and thus low storage requirement. However, the mesh topology is restricted due to orthogonality and conjunctionality requirements. The value at a face is assumed to be the average value over the face. On an unstructured mesh the face center may not lie on the line joining the CV centers, which will lead to an error in the face interpolation. FiPy currently only uses the CC-FVM.

Vertex Centered FVM (VC-FVM)

In the VC-FVM, the CV is centered around the vertices and the cells are divided into sub-control volumes that make up the main CVs (see Figure 3.2). The vertices "store" the average variable values over the CVs. The CV faces are constructed within the cells rather than using the cell faces as in the CC-FVM. The face fluxes use all the vertex values from the cell where the face is located to calculate interpolations. For this reason, the VC-FVM is less efficient and requires more storage (a larger matrix band width) than the



(a) (b)

Figure 3.2: CV structure for an unstructured mesh, (a) Ω_a represents a vertex-based CV and (b) Ω_1, Ω_2, Ω_3 and Ω_4 represent cell centered CVs.

CC-FVM. However, the mesh topology does not have the same restrictions as the CC-FVM. FiPy does not have a VC-FVM capability.

3.3 Discretization

The first step in the discretization of Equation (3.2) using the CC-FVM is to integrate over a CV and then make appropriate approximations for fluxes across the boundary of each CV. In this section, each term in Equation (3.2) will be examined separately.

Transient Term

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

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

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

Convection Term

The discretization for the convection term is given by

$$\int_V \nabla \cdot (\vec{u}\phi)\, dV = \int_S (\vec{n} \cdot \vec{u})\phi\, dS \tag{3.4}$$

$$\simeq \sum_f (\vec{n} \cdot \vec{u})_f \phi_f A_f \tag{3.5}$$

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

$$\phi_f = \alpha_f \phi_P + (1 - \alpha_f)\phi_A. \tag{3.6}$$

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

Diffusion Term

The discretization for the diffusion term is given by

$$\int_V \nabla \cdot (\Gamma \nabla\{\ldots\}) dV = \int_S \Gamma(\vec{n} \cdot \nabla\{\ldots\}) dS \tag{3.7}$$

$$\simeq \sum_f \Gamma_f (\vec{n} \cdot \nabla\{\ldots\})_f A_f \tag{3.8}$$

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

$$(\vec{n} \cdot \nabla\{\ldots\})_f \simeq \frac{\{\ldots\}_A - \{\ldots\}_P}{d_{AP}} \tag{3.9}$$

where the value of d_{AP} is the distance between neighboring cell centers. This estimate relies on the orthogonality of the mesh, and becomes increasingly inaccurate as the non-orthogonality increases. Correction terms have been derived to improve this error but are not currently included in FiPy [17].

Source Term

The discretization for the source term is given by,

$$\int_V S_\phi \, dV \simeq S_\phi V_P. \tag{3.10}$$

Including any negative dependence of S_ϕ on ϕ increases solution stability. The dependence can only be included in a linear manner so Equation (3.10) becomes

$$V_P(S_0 + S_1 \phi_P), \tag{3.11}$$

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

3.4 Linear Equations

The aim of the discretization is to reduce the continuous general equation to a set of discrete linear equations that can then be solved to obtain the value of the dependent variable at each CV center. This results in a

sparse linear system that requires an efficient iterative scheme to solve. The iterative schemes available to FiPy are currently encapsulated in the spmatrix suite of solvers and include most common solvers such as the conjugate gradient method and LU decomposition. There are plans to include other solver suites that are compatible with Python.

Combining Equations (3.3), (3.5), (3.8) and (3.10), the complete discretization for equation (3.2) can now be written for each CV as

$$\begin{aligned}\frac{\rho_P(\phi_P - \phi_P^{\text{old}})V_P}{\Delta t} &= \sum_f (\vec{n}\cdot\vec{u})_f A_f \left[\alpha_f \phi_P + (1-\alpha_f)\,\phi_A\right] \\ &\quad + \sum_f \Gamma_f A_f \frac{(\phi_A - \phi_P)}{d_{AP}} + V_P(S_0 + S_1\phi_P). \end{aligned} \tag{3.12}$$

Equation (3.12) is now in the form of a set of linear combinations between each CV value and its neighboring values and can be written in the form

$$a_P\phi_P = \sum_f a_A\phi_A + b_P, \tag{3.13}$$

where

$$a_P = \frac{\rho_P V_P}{\Delta t} + \sum_f (a_A - F_f) - V_P S_1, \tag{3.14}$$

$$a_A = (1-\alpha_f)F_f + D_f, \tag{3.15}$$

$$b_P = V_P S_0 + \frac{\rho_P V_P \phi_P^{\text{old}}}{\Delta t}. \tag{3.16}$$

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

$$F_f = A_f(\vec{u}\cdot\vec{n})_f, \tag{3.17}$$

$$D_f = \frac{A_f\Gamma_f}{d_{AP}}. \tag{3.18}$$

3.5 Numerical Schemes

The coefficients of equation (3.13) must remain positive, since an increase in a neighboring value must result in an increase in ϕ_P to obtain physically realistic solutions. Thus, the inequalities $a_A > 0$ and $a_A - F_f > 0$ must be satisfied. The Péclet number $P_f \equiv -F_f/D_f$ is the ratio between convective strength and diffusive conductance. To achieve physically realistic solutions, the inequality

$$\frac{1}{1-\alpha_f} > P_f > -\frac{1}{\alpha_f} \tag{3.19}$$

must be satisfied. The parameter α_f is defined by the chosen scheme, depending on Equation (3.19). The various differencing schemes are:

the central differencing scheme, where

$$\alpha_f = \frac{1}{2} \tag{3.20}$$

so that $|P_f| < 2$ satisfies Equation (3.19). Thus, the central differencing scheme is only numerically stable for a low values of P_f.

the upwind scheme, where

$$\alpha_f = \begin{cases} 1 & \text{if } P_f > 0, \\ 0 & \text{if } P_f < 0. \end{cases} \tag{3.21}$$

Equation (3.21) satisfies the inequality in Equation (3.19) for all values of P_f. However the solution over predicts the diffusive term leading to excessive numerical smearing ("false diffusion").

the exponential scheme, where

$$\alpha_f = \frac{(P_f - 1)\exp(P_f) + 1}{P_f(\exp(P_f) - 1)}. \tag{3.22}$$

This formulation can be derived from the exact solution, and thus, guarantees positive coefficients while not over-predicting the diffusive terms. However, the computation of exponentials is slow and therefore a faster scheme is generally used, especially in higher dimensions.

the hybrid scheme, where

$$\alpha_f = \begin{cases} \frac{P_f - 1}{P_f} & \text{if } P_f > 2, \\ \frac{1}{2} & \text{if } |P_f| < 2, \\ -\frac{1}{P_f} & \text{if } P_f < -2. \end{cases} \tag{3.23}$$

The hybrid scheme is formulated by allowing $P_f \to \infty$, $P_f \to 0$ and $P_f \to -\infty$ in the exponential scheme. The hybrid scheme is an improvement on the upwind scheme, however, it deviates from the exponential scheme at $|P_f| = 2$.

the power law scheme, where

$$\alpha_f = \begin{cases} \frac{P_f - 1}{P_f} & \text{if } P_f > 10, \\ \frac{(P_f - 1) + (1 - P_f/10)^5}{P_f} & \text{if } 0 < P_f < 10, \\ \frac{(1 - P_f/10)^5 - 1}{P_f} & \text{if } -10 < P_f < 0, \\ -\frac{1}{P_f} & \text{if } P_f < -10. \end{cases} \tag{3.24}$$

The power law scheme overcomes the inaccuracies of the hybrid scheme, while improving on the computational time for the exponential scheme.

All of the numerical schemes presented here are available in FiPy and can be selected by the user.

Design and Implementation

The goal of FiPy is to provide a highly customizable, open source code for modeling problems involving coupled sets of PDEs. FiPy allows users to select and customize modules from within the framework. FiPy has been developed to address model problems in materials science such as poly-crystals, dendritic growth and electrochemical deposition. These applications all contain various combinations of PDEs with differing forms in conjunction with other unusual physics (over varying length scales) and unique solution procedures. The philosophy of FiPy is to enable customization while providing a library of efficient modules for common objects and data types.

4.1 Design

Numerical Approach

The solution algorithms given in the FiPy examples involve combining sets of PDEs while tracking an interface where the parameters of the problem change rapidly. The phase field method and the level set method are specialized techniques to handle the solution of PDEs in conjunction with a deforming interface. FiPy contains several examples of both methods.

FiPy uses the well-known Finite Volume Method (FVM) to reduce the model equations to a form tractable to linear solvers.

Object Oriented Structure

FiPy is programmed in an object-oriented manner. The benefit of object oriented programming mainly lies in encapsulation and inheritance. Encapsulation refers to the tight integration between certain pieces of data and methods that act on that data. Encapsulation allows parts of the code to be separated into clearly defined independent modules that can be re-applied or extended in new ways. Inheritance allows code to be reused, overridden, and new capabilities to be added without altering the original code. An object is treated by its users as an abstraction; the details of its implementation and behavior are internal.

Test Based Development

FiPy has been developed with a large number of test cases. These test cases are in two categories. The lower level tests operate on the core modules at the individual method level. The aim is that every method within

the core installation has a test case. The high level test cases operate in conjunction with example solutions and serve to test global solution algorithms and the interaction of various modules.

With this two-tiered battery of tests, at any stage in code development, the test cases can be executed and errors can be identified. A comprehensive test base provides reassurance that any code breakages will be clearly demonstrated with a broken test case. A test base also aids dissemination of the code by providing simple examples and knowledge of whether the code is working on a particular computer environment.

Open Source

In recent years, there has been a movement to release software under open source and associated unrestrictied licenses, especially within the scientific community. These licensing terms allow users to develop their own applications with complete access to the source code and then either contribute back to the main source repository or freely distribute their new adapted version.

As a product of the National Institute of Standards and Technology, the FiPy framework is placed in the public domain as a matter of U. S. Federal law. Furthermore, FiPy is built upon existing open source tools. Others are free to use FiPy as they see fit and we welcome contributions to make FiPy better.

High-Level Scripting Language

Programming languages can be broadly lumped into two categories: compiled languages and interpreted (or scripting) languages. Compiled languages are converted from a human-readable text source file to a machine-readable binary application file by a sequence of operations generally referred to as "compiling" and "linking". The binary application can then be run as many times as desired, but changes will provoke a new cycle of compiling and linking. Interpreted languages are converted from human-readable to machine-readable on the fly, each time the script is executed. Because the conversion happens every time[1], interpreted code is usually slower when running than compiled code. On the other hand, code development and debugging tends to be much easier and fluid when it's not necessary to wait for compile and link cycles after every change. Furthermore, because the conversion happens in real time, it is possible to have interactive sessions in a scripting language that are not generally possible in compiled languages.

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

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

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

for (i = 0; i < Bcols; i++) {
    for (j = 0; j < Arows; j++) {
```

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

```
            C[i][j] = 0;
            for (k = 0; k < Acols; k++) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
```

Note that the dimensions of the arrays must be supplied externally, as C provides no intrinsic mechanism for determining the shape of an array. An equivalent high-level construction might be as simple as

```
C = A * B
```

All of the error checking, dimension measuring, and space allocation is handled automatically by low-level code that is intrinsic to the high-level matrix multiplication operator. The high-level code "knows" that matrices are involved, how to get their shapes, and to interpret '`*`' as a matrix multiplier instead of an arithmetic one. All of this allows the programmer to think about the operation of interest and not worry about introducing bugs in low-level code that is not unique to their application.

Although it needn't be true, for a variety of reasons, compiled languages tend to be low-level and interpreted languages tend to be high-level. Because low-level languages operate closer to the intrinsic "machine language" of the computer, they tend to be faster at running a given task than high-level languages, but programs written in them take longer to write and debug. Because running performance is a paramount concern, most scientific codes are written in low-level compiled languages like FORTRAN or C.

A rather common scenario in the development of scientific codes is that the first draft hard-codes all of the problem parameters. After a few (hundred) iterations of recompiling and relinking the application to explore changes to the parameters, code is added to read an input file containing a list of numbers. Eventually, the point is reached where it is impossible to remember which parameter comes in which order or what physical units are required, so code is added to, for example, interpret a line beginning with '`#`' as a comment. At this point, the scientist has begun developing a scripting language without even knowing it. Unfortunately for them, very few scientists have actually studied computer science or actually know anything about the design and implementation of script interpreters. Even if they have the expertise, the time spent developing such a language interpreter is time not spent actually doing research.

In contrast, a number of very powerful scripting languages, such as Tcl, Java, Python, Ruby, and even the venerable BASIC, have open source interpreters that can be embedded directly in an application, giving scientific codes immediate access to a high-level scripting language designed by someone who actually knew what they were doing.

We have chosen to go a step further and not just embed a full-fledged scripting language in the FiPy framework, but instead to design the framework from the ground up in a scripting language. While runtime performance is unquestionably important, many scientific codes are run relatively little, in proportion to the time spent developing them. If a code can be developed in a day instead of a month, it may not matter if it takes another day to run instead of an hour. Furthermore, there are a variety of mechanisms for diagnosing and optimizing those portions of a code that are actually time-critical, rather than attempting to optimize all of it by using a language that is more palatable to the computer than to the programmer. Thus FiPy, rather than taking the approach of writing the fast numerical code first and then dealing with the issue of user interaction, initially implements most modules in high-level scripting language and only translates to low-level compiled code those portions that prove inefficient.

Python Programming Language

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

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

4.2 Implementation

The Python classes that make up FiPy are described in detail in the *FiPy Programmer's Reference*, but we give a brief overview here. FiPy is based around three fundamental Python classes: `Mesh`, `Variable`, and `Term`. Using the terminology of Chapter 3:

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

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

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

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

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

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

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

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

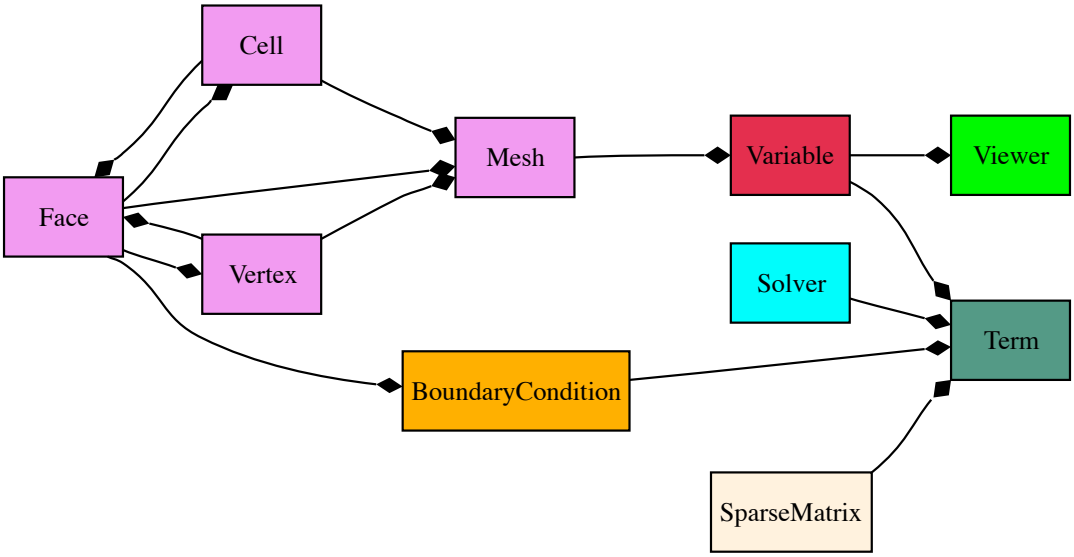


Figure 4.1: Primary object relationships in FiPy.

A `Term` object represents any of the terms in Equation (3.2) or any linear combination of such terms. Early in the development of FiPy, a distinction was made between `Equation` objects, which represented all of Equation (3.2), and `Term` objects, which represented the individual terms in that equation. The `Equation` object has since been eliminated as redundant. `Term` objects can be single entities such as an `ImplicitDiffusionTerm` or a linear combination of other `Term` objects that build up to form an expression such as Equation (3.2).

Beyond these three fundamental classes of `Mesh`, `Variable`, and `Term`, FiPy is composed of a number of related classes. The relationships between these classes are shown in Figure 4.1. A `Mesh` object is composed of `Cell` objects. Each `Cell` is defined by its bounding `Face` objects and each `Face` is defined by its bounding `Vertex` objects. A `Term` object encapsulates the contributions to the `SparseMatrix` that defines the solution of an equation. `BoundaryCondition` objects are used to describe the conditions on the boundaries of the `Mesh`, and each `Term` interprets the `BoundaryCondition` objects as necessary to modify the `SparseMatrix`. An equation constructed from `Term` objects can apply a unique `Solver` to invert its `SparseMatrix` in the most expedient and stable fashion. At any point during the solution, a `Viewer` can be invoked to display the values of the solved `Variable` objects.

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

Frequently Asked Questions

5.1 How do I represent an equation in FiPy?

As explained in Chapter 3, the canonical governing equation that can be solved by FiPy for the dependent `CellVariable` ϕ is

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

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

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

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

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

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

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

where `<Specific>` can be any of `CentralDiff`, `Exponential`, `Hybrid`, `PowerLaw`, `Upwind`, `ExplicitUpwind`, or `VanLeer`. The differences between these convection schemes are described in Section 3.5. The velocity coefficient `u` must be a rank-1 `FaceVariable`, or a constant vector in the form of a Python list or tuple, *e.g.* `((1,), (2,))` for a vector in 2D.

Note As discussed in Section 3.5, the convection schemes need to calculate a Péclet number, and therefore need to know about any diffusion term used in the problem. It is hoped that this dependency can be automated in the future.

Warning `VanLeerConvectionTerm` not mentioned and no discussion of explicit forms.

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

Either

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

or

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

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

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

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

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

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

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

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

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

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

which represents $5\partial_x^2 + \partial_y^2$. Notice that the tensor, written in the form of a list, is contained within a list. This is because the first index of the list refers to the order of the term not the first index of the tensor (see the FAQ, Chapter 5.1 "How do I represent a term $\nabla^4\phi$ or $\nabla\cdot(\Gamma_1\nabla(\nabla\cdot(\Gamma_2\nabla\phi)))$ such as for Cahn-Hilliard? "). This notation, although succinct can sometimes be confusing so a number of cases are interpreted below.

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

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

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

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

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

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

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

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

- Generally, anisotropy is not conveniently aligned along the coordinate axes; in these cases, it is necessary to apply a rotation matrix in order to calculate the correct tensor values, see Chapter 6.6 "Module examples.diffusion.anisotropy" for details.

What if the term isn't one of those?

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

```
>>> 3 * kappa**2 + b * sin(theta)
```

Note Functions like `sin()` can be obtained from the fipy.tools.numerix module.

Warning Generally, things will not work as expected if the equivalent function is used from the NumPy or SciPy library.

If, however, the source depends on the variable that is being solved for, it can be advantageous to linearize the source and cast part of it as an implicit source term, *e.g.*, $3\kappa^2 + \phi\sin\theta$ might be written as

```
>>> 3 * kappa**2 + ImplicitSourceTerm(coeff = sin(theta))
```
```

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

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

How do I represent a diffusive source?

If the governing equation for ϕ is

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

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

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

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

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

How do I represent a convective source?

The convection of an independent field ξ as in

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

can be rendered as

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

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

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

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

How do I represent a transient source?

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

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

does not have an abstract form in FiPy and should be discretized directly, in the manner of Equation (3.3), as

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

This technique is used in Example 8.4.

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

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

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

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

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

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

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

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

Note Due to the nonlinearity of the coefficient, it will probably be necessary to "sweep" the solution to convergence as discussed in FAQ 5.3.

5.2 How can I see what I'm doing?

How do I export data?

The way to save your calculations depends on how you plan to make use of the data. If you want to save it for "restart" (so that you can continue or redirect a calculation from some intermediate stage), then you'll want to "pickle" the Python data with the `dump` module. This is illustrated in Examples 8.4, 8.5, 8.6, and 9.8.

On the other hand, pickled FiPy data is of little use to anything besides Python and FiPy. If you want to import your calculations into another piece of software, whether to make publication-quality graphs or movies, or to perform some analysis, or as input to another stage of a multiscale model, then you can save your data as an ASCII text file of tab-separated-values with a `TSVViewer`. This is illustrated in Example 6.3.

How do I save a plot image?

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

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

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

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

gnuplot accepts "`.eps`".

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

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

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

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

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

How do I make a movie?

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

Why don't the `Viewers` look the way I want?

FiPy's viewers are utilitarian. They're designed to let you see *something* with a minimum of effort. Because different plotting packages have different capabilities and some are easier to install on some platforms than on others, we have tried to support a range of Python plotters with a minimal common set of features. Many of these packages are capable of much more, however. Often, you can invoke the `Viewer` you want, and then issue supplemental commands for the underlying plotting package. The better option is to make a "subclass" of the FiPy `Viewer` that comes closest to producing the image you want. You can then override just the behavior you wan to change, while letting FiPy do most of the heavy lifting. See 8.4 for an example of creating a custom Matplotlib `Viewer` class.

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

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

iterations This is the lowest layer of repetition, which you'll generally need to spend the least time thinking about. FiPy solves PDEs by discretizing them into a set of linear equations in matrix form, as explained in Sections 3.3 and 3.4. It is not always practical, or even possible, to exactly solve these matrix equations on a computer. FiPy thus employs "iterative solvers", which make successive approximations until the linear equations have been satisfactorily solved. FiPy chooses a default number of iterations and solution tolerance, which you will not generally need to change. If you do wish to change these defaults, you'll need to create a new `Solver` object with the desired number of iterations and solution tolerance, *e.g.*

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

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

Solver iterations are changed from their defaults in Examples 11.1 and 12.2.

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

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

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

```
>>> myVar = CellVariable(..., hasOld=1)
```

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

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

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

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

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

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

Sweeps are used to achieve better solutions in Examples 6.1, 8.1, 8.2, and 11.1.

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

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

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

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

or more elaborate:

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

A majority of the examples in this manual illustrate time evolving behavior. Notably, boundary conditions are made a function of elapsed time in Example 6.1. The timestep is chosen based on the expected interfacial velocity in Example 8.1. The timestep is gradually increased as the kinetics slow down in Example 10.1.

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

```
>>> myVar = CellVariable(..., hasOld=1)
    :
    :
>>> mySolver = LinearPCGSolver(iterations=1234, tolerance=5e-6)
    :
```

```
    :
>>> while elapsedTime < totalElapsedTime:
...     myVar.updateOld()
...     while residual > desiredResidual:
...         residual = eq.sweep(var=myVar, dt=timeStep, ...)
...     elapsedTime += timeStep
```

5.4 Why the distinction between CellVariable and FaceVariable coefficients?

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

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

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

This distinction is not generally important for smoothly varying functions, but can dramatically affect the solution when sharp changes are present. Also, for many problems, such as a conserved concentration field that cannot be allowed to drop below zero, a harmonic average is more appropriate than an arithmetic average. If you experience problems (unstable or wrong results, or excessively small timesteps), you may need to explicitly supply the desired FaceVariable rather than letting FiPy assume one.

5.5 How do I represent boundary conditions?

What is a FixedValue boundary condition?

This is simply a Dirichlet boundary condition by another name.

What does the FixedFlux boundary condition actually represent?

In FiPy a FixedFlux boundary condition object represents the quantity

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

on a given boundary edge with $\vec{n}$ pointing out of the boundary. The quantity Γ represents the diffusion coefficient and $\vec{u}$ represents the convection coefficient for a general convection-diffusion equation of the type given in Eq. (3.2). See Example 7.3 for a usage case.

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

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

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

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

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

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

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

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

How do I apply an outlet or inlet boundary condition?

There is no good way to do this with the standard boundary conditions in FiPy and thus one needs to use the method suggested above, see 5.5 "I can't get the `FixedValue` or `FixedFlux` boundary condition objects to work right. What do I do now?". Currently, boundary conditions for the `ConvectionTerm` assume a `FixedFlux` boundary condition with a `value` of 0. This is in fact not the most intuitive default boundary condition, a natural outlet or inlet boundary condition would in fact be more sensible. In order to apply an inlet/outlet boundary condition one needs a separate exterior convection coefficient (velocity vector) to hold the boundary values,

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

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

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

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

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

As in the previous section, the coefficient values may need updating on the exterior faces between sweeps. See Example 7.4 "Module examples.convection.source" for an example of this usage.

How do I apply a fixed gradient?

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

```
FixedFlux(value=gradient / diffusionCoeff, faces=myFaces)
```

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

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

This is not an "implicit" boundary condition so would in general require sweeps to reach convergence. See Example 7.4 "Module examples.convection.source" for an example of this usage. In the case of a non-zero gradient one would need to employ the techniques in both 5.5 "I can't get the `FixedValue` or `FixedFlux` boundary condition objects to work right. What do I do now?" and 5.5 "How do I apply an outlet or inlet boundary condition?" without using either a `FixedValue` or a `FixedFlux` object.

How do I apply spatially varying boundary conditions?

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

$$\phi = xy \quad \text{on } x > 1/2 \text{ and } y > 1/2 \tag{5.1}$$
$$\vec{n} \cdot \vec{F} = 0 \quad \text{elsewhere} \tag{5.2}$$

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

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

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

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

Further demonstrations of spatially varying boundary condition can be found in Examples 6.2 "Module examples.diffusion.mesh20x20" and 6.3 "Module examples.diffusion.circle"

5.6 What does this error message mean?

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

"`MA.MA.MAError: Cannot automatically convert masked array to Numeric because data is masked in one or`" This not-so-helpful error message could mean a number of things, but the most likely explanation is that the solution has become unstable and is diverging to $\pm\infty$. This can be caused by taking too large a timestep or by using explicit terms instead of implicit ones.

"`repairing catalog by removing key`" This message (not really an error, but may cause test failures) can result when using the SciPy weave package via the `--inline` flag. It is due to a bug in SciPy that has been patched in their source repository: `http://www.scipy.org/mailinglists/mailman?fn=scipy-dev/2005-June/003010.html`.

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

5.7 How do I change FiPy's default behavior?

FiPy tries to make reasonable choices, based on what packages it finds installed, but there may be times that you wish to override these behaviors.

Command-line Flags

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

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

`--Pysparse` Forces the use of the Trilinos solvers. This flag takes precedence over the `FIPY_SOLVERS` environment variable.

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

Environment Variables

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

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

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

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

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

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

5.8 Why don't my scripts work anymore?

FiPy has experienced two major API changes. The steps necessary to upgrade older scripts are discussed in Chapter 12.

5.9 What if my question isn't answered here?

Please post your question to the mailing list [6] or file a Tracker request [7].

Part II

Examples

Note

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

These examples can be used in at least four ways:

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

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

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

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

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

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

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

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

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

  Note

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

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

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

Example Contents

Diffusion Examples

6.1 Module examples.diffusion.mesh1D

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

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

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

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

$$\frac{\partial \phi}{\partial t} = D\nabla^2\phi. \tag{6.1}$$

The first step is to define a one dimensional domain with 50 solution points. The `Grid1D` object represents a linear structured grid. The parameter `dx` refers to the grid spacing (set to unity here).

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

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

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

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

We'll let

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

for now.

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

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

are formed with a value

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

and a set of faces over which they apply.

Note

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

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

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

Note

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

If you have ever tried to numerically solve Eq. (6.1), you most likely attempted "explicit finite differencing" with code something like:

```
for step in range(steps):
    for j in range(cells):
        phi_new[j] = phi_old[j] \
          + (D * dt / dx**2) * (phi_old[j+1] - 2 * phi_old[j] + phi_old[j-1])
    time += dt
```

plus additional code for the boundary conditions. In FiPy, you would write

```
>>> eqX = TransientTerm() == ExplicitDiffusionTerm(coeff=D)
```

The largest stable timestep that can be taken for this explicit 1D diffusion problem is $\Delta t \leq \Delta x^2/(2D)$. We limit our steps to 90% of that value for good measure

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

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

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

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

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

```
>>> x = mesh.getCellCenters()[0]
>>> t = timeStepDuration * steps
```

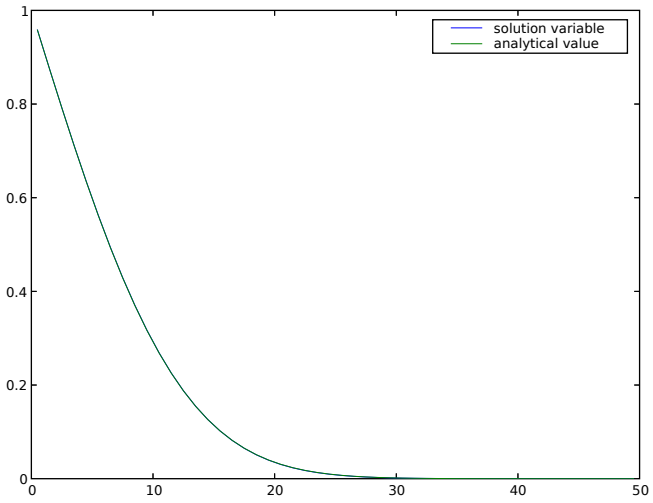
```
>>> try:
...     from scipy.special import erf
...     phiAnalytical.setValue(1 - erf(x / (2 * sqrt(D * t))))
... except ImportError:
...     print "The SciPy library is not available to test the solution to \
... the transient diffusion equation"
```

We then solve the equation by repeatedly looping in time:

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

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

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



Although explicit finite differences are easy to program, we have just seen that this 1D transient diffusion problem is limited to taking rather small time steps. If, instead, we represent Eq. (6.1) as:

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

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

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

reset the problem

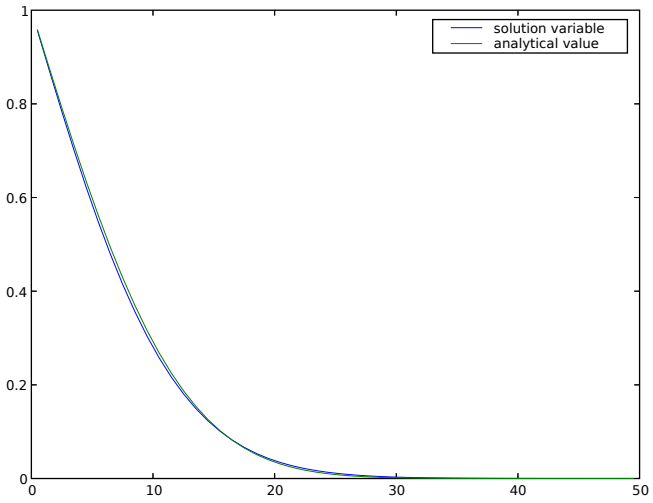
```
>>> phi.setValue(valueRight)
```

and rerun with much larger time steps

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

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


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



Note that although much larger *stable* timesteps can be taken with this implicit version (there is, in fact, no limit to how large an implicit timestep you can take for this particular problem), the solution is less *accurate*. One way to achieve a compromise between *stability* and *accuracy* is with the Crank-Nicholson scheme, represented by:

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

which is essentially an average of the explicit and implicit schemes from above. This can be rendered in FiPy as easily as

```
>>> eqCN = eqX + eqI
```

We again reset the problem

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

and apply the Crank-Nicholson scheme until the end, when we apply one step of the fully implicit scheme to drive down the error (see, *e.g.*, [24, §19.2]).

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


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


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

As mentioned above, there is no stable limit to how large a time step can be taken for the implicit diffusion problem. In fact, if the time evolution of the problem is not interesting, it is possible to eliminate the time step altogether by omitting the `TransientTerm`. The steady-state diffusion equation

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

is represented in FiPy by

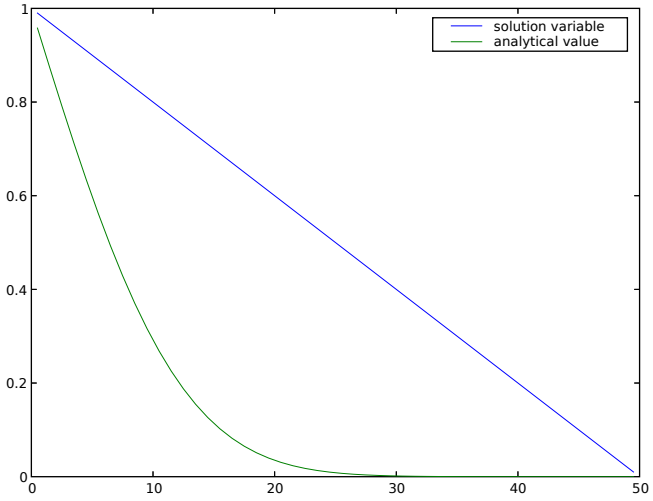
```
>>> ImplicitDiffusionTerm(coeff=D).solve(var=phi,
...                                      boundaryConditions=BCs)
...


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

The analytical solution to the steady-state problem is no longer an error function, but simply a straight line, which we can confirm to a tolerance of 10^{-10}.

```
>>> L = nx * dx
>>> print phi.allclose(valueLeft + (valueRight - valueLeft) * x / L,
...                    rtol = 1e-10, atol = 1e-10)
1


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



Often, boundary conditions may be functions of another variable in the system or of time. For example, to have

$$\phi = \begin{cases} (1 + \sin t)/2 & \text{on } x = 0 \\ 0 & \text{on } x = L \end{cases}$$

we will need to declare time t as a `Variable`

```
>>> time = Variable()
```

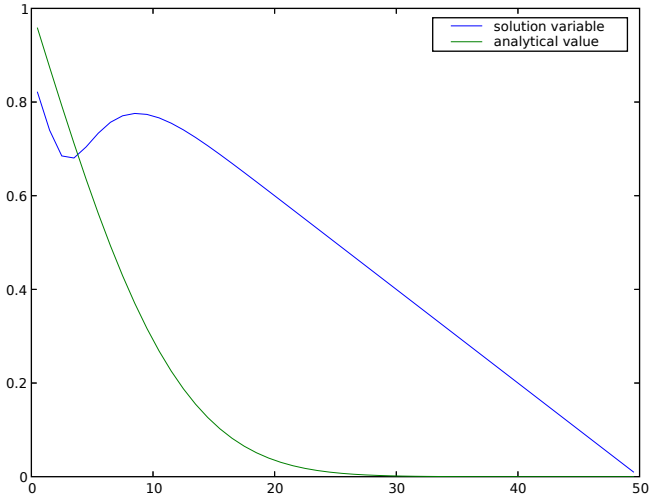
and then declare our boundary condition as a function of this `Variable`

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

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

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


>>> if __name__ == '__main__':
...     raw_input("Time-dependent boundary condition. Press <return> to proceed...")
```



Many interesting problems do not have simple, uniform diffusivities. We consider a steady-state diffusion problem

$$\nabla \cdot (D \nabla \phi) = 0,$$

with a spatially varying diffusion coefficient

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

and with boundary conditions $\phi = 0$ at $x = 0$ and $D\frac{\partial \phi}{\partial x} = 1$ at $x = L$, where L is the length of the solution domain. Exact numerical answers to this problem are found when the mesh has cell centers that lie at $L/4$ and $3L/4$, or when the number of cells in the mesh N_i satisfies $N_i = 4i + 2$, where i is an integer. The mesh we've been using thus far is satisfactory, with $Ni = 50$ and $i = 12$. Because FiPy considers diffusion to be a flux from one `Cell` to the next, through the intervening `Face`, we must define the non-uniform diffusion coefficient on the mesh faces

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

The boundary conditions are a fixed value of

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

to the left and a fixed flux of

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

to the right:

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

We re-initialize the solution variable

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

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

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

The analytical solution is simply

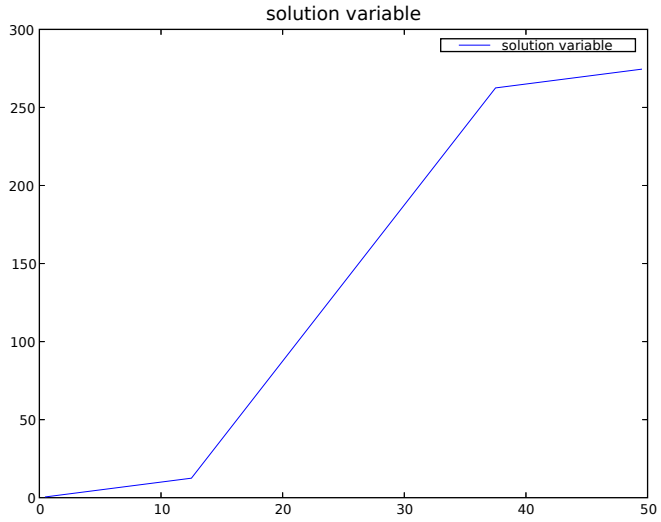
$$\phi = \begin{cases} x & \text{for } 0 < x < L/4, \\ 10x - 9L/4 & \text{for } L/4 \leq x < 3L/4, \\ x + 18L/4 & \text{for } 3L/4 \leq x < L, \end{cases}$$

or

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

And finally, we can plot the result

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



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

$$\frac{\partial \phi}{\partial t} = \nabla \cdot [D(\phi) \nabla \phi]. \tag{6.2}$$

With such a non-linearity, it is generally necessary to "sweep" the solution to convergence. This means that each time step should be calculated over and over, using the result of the previous sweep to update the coefficients of the equation, without advancing in time. In FiPy, this is accomplished by creating a solution variable that explicitly retains its "old" value by specifying `hasOld` when you create it. The variable does not move forward in time until it is explicity told to `updateOld()`. In order to compare the effects of different numbers of sweeps, let us create a list of variables: `phi[0]` will be the variable that is actually being solved and `phi[1]` through `phi[4]` will display the result of taking the corresponding number of sweeps (`phi[1]` being equivalent to not sweeping at all).

```
>>> valueLeft = 1.
>>> valueRight = 0.
>>> phi = [
...     CellVariable(name="solution variable",
...                  mesh=mesh,
...                  value=valueRight,
...                  hasOld=1),
...     CellVariable(name="1 sweep",
...                  mesh=mesh),
...     CellVariable(name="2 sweeps",
...                  mesh=mesh),
...     CellVariable(name="3 sweeps",
...                  mesh=mesh),
...     CellVariable(name="4 sweeps",
...                  mesh=mesh)
... ]
```

If, for example,

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

we would simply write Eq. (6.2) as

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

Note

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

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

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

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

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

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

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

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

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

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

```
>>> for sweeps in range(1,5):
...     phi[0].setValue(valueRight)
...     for step in range(steps):
...         # only move forward in time once per time step
...         phi[0].updateOld()
...
...         # but "sweep" many times per time step
...         for sweep in range(sweeps):
```

```
...             res = eq.sweep(var=phi[0],
...                            boundaryConditions=BCs,
...                            dt=timeStepDuration)
...         if __name__ == '__main__':
...             viewer.plot()
...
...     # copy the final result into the appropriate display variable
...     phi[sweeps].setValue(phi[0])
...     if __name__ == '__main__':
...         viewer.plot()
...         raw_input("Implicit variable diffusity. %d sweep(s). \
... Residual = %f. Press <return> to proceed..." % (sweeps, (abs(res))))
```

As can be seen, sweeping does not dramatically change the result, but the "residual" of the equation (a measure of how accurately it has been solved) drops about an order of magnitude with each additional sweep.

Attention!

Choosing an optimal balance between the number of time steps, the number of sweeps, the number of solver iterations, and the solver tolerance is more art than science and will require some experimentation on your part for each new problem.

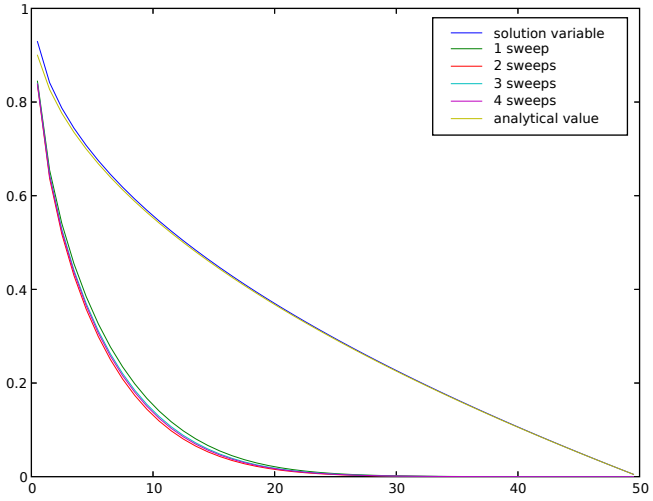
Finally, we can increase the number of steps to approach equilibrium, or we can just solve for it directly

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

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

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

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



If this example had been written primarily as a script, instead of as documentation, we would delete every line that does not begin with either ">>>" or "...", and then delete those prefixes from the remaining lines, leaving:

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

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

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

    .
    .
    .

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

```
print phi[0].allclose(phiAnalytical, atol = 1e-1)
# Expect:
# 1
#
if __name__ == '__main__':
    viewer.plot()
    raw_input("Implicit variable diffusity - steady-state. \
Press <return> to proceed...")
```

Your own scripts will tend to look like this, although you can always write them as doctest scripts if you choose. You can obtain a plain script like this from some `.../example.py` by typing:

```
$ python setup.py copy_script --From .../example.py --To myExample.py
```

at the command line.

Most of the FiPy examples will be a mixture of plain scripts and doctest documentation/tests.

6.2 Module examples.diffusion.mesh20x20

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

```
>>> from fipy import *

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

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

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

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

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

We apply Dirichlet boundary conditions

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

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

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

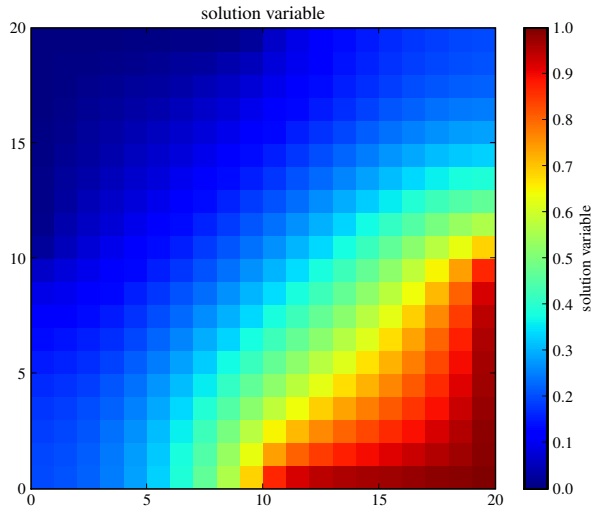
>>> BCs = (FixedValue(faces=facesTopLeft, value=valueTopLeft),
...        FixedValue(faces=facesBottomRight, value=valueBottomRight))
```

We create a viewer to see the results

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

and solve the equation by repeatedly looping in time:

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



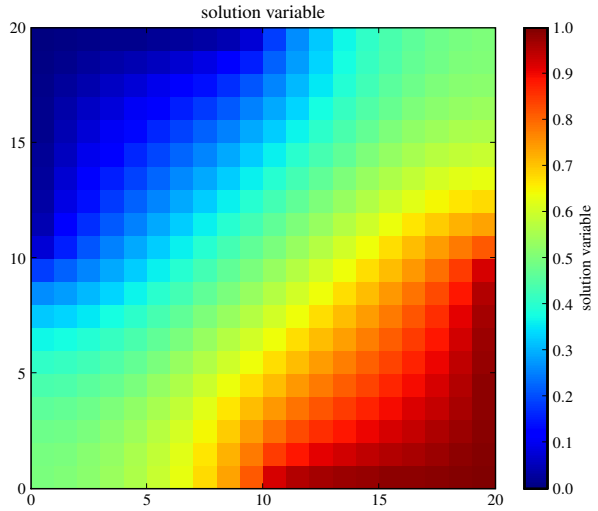
We can test the value of the bottom-right corner cell.

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

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

We can also solve the steady-state problem directly

```
>>> ImplicitDiffusionTerm().solve(var=phi,
...                               boundaryConditions = BCs)
>>> if __name__ == '__main__':
...     viewer.plot()
```



and test the value of the bottom-right corner cell.

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

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

6.3 Module examples.diffusion.circle

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

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

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

The mesh created by gmsh is then imported into FiPy using the `GmshImporter2D` object.

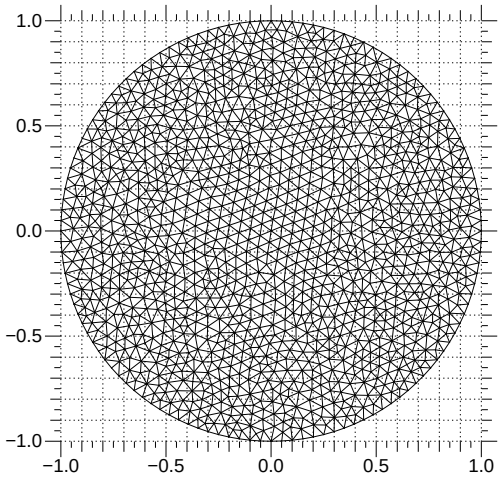
```
>>> from fipy import *
>>> mesh = GmshImporter2D('''
...                        cellSize = %(cellSize)g;
...                        radius = %(radius)g;
...                        Point(1) = {0, 0, 0, cellSize};
...                        Point(2) = {-radius, 0, 0, cellSize};
...                        Point(3) = {0, radius, 0, cellSize};
...                        Point(4) = {radius, 0, 0, cellSize};
...                        Point(5) = {0, -radius, 0, cellSize};
...                        Circle(6) = {2, 1, 3};
...                        Circle(7) = {3, 1, 4};
...                        Circle(8) = {4, 1, 5};
...                        Circle(9) = {5, 1, 2};
...                        Line Loop(10) = {6, 7, 8, 9};
...                        Plane Surface(11) = {10};
...                        ''' % locals())
```

Using this mesh, we can construct a solution variable

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

We can now create a viewer to see the mesh

```
>>> viewer = None
>>> if __name__ == '__main__':
...     try:
...         viewer = Viewer(vars=phi, datamin=-1, datamax=1.)
...         viewer.plotMesh()
...         raw_input("Irregular circular mesh. Press <return> to proceed...")
...     except:
...         print "Unable to create a viewer for an irregular mesh (try Gist2DViewer or Matplotlib2DViewer)"
```



We set up a transient diffusion equation

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

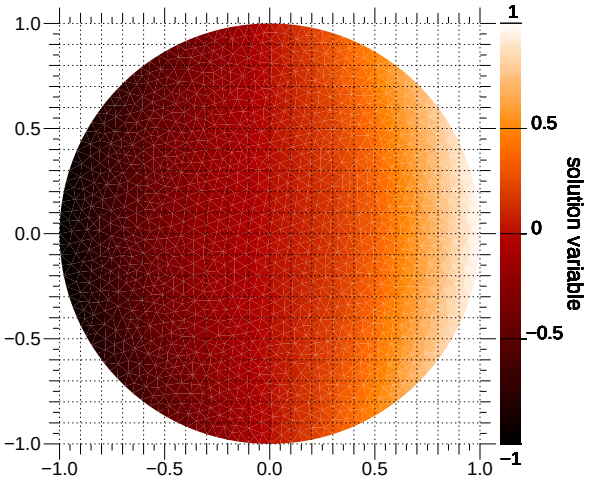
The following line extracts the `x` coordinate values on the exterior faces. These are used as the boundary condition fixed values.

```
>>> X, Y = mesh.getFaceCenters()
```

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

We first step through the transient problem

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



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

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

x	y	solution variable	solution variable_grad_x	solution variable_grad_y
0.975559734792414	0.0755414402612554	0.964844363287199	-0.229687917881182	0.00757854476106331
0.0442864953037566	0.7919189316238 4	0.0375859836421991	-0.773936613923853	-0.205560697612547
0.0246775505084069	0.771959648896982	0.020853932412869	-0.723540342405813	-0.182589694334729
0.223345558247991	-0.807931073108895	0.203035857140125	-0.777466238738658	0.0401235242511506
-0.00726763301939488	-0.775978916110686	-0.00412895434496877	-0.650055516507232	-0.183112882869288
-0.0220279064527904	-0.187563765977912	-0.012771874945585	-0.35707168379437	-0.056072788439713
0.111223320911545	-0.679586798311355	0.0911595298310758	-0.613455176718145	0.0256182541329463
-0.78996770899909	-0.0173672729866294	-0.693887874335319	-1.00671109050419	-0.127611490372511
-0.703545986179876	-0.435813500559859	-0.635004192597412	-0.896203033957194	-0.00855563518923689
0.888641841567831	-0.408558914368324	0.877939107374768	-0.32195762184087	-0.22696791637322
0.38212257821916	-0.51732949653553	0.292889724306196	-0.854466141879776	0.199715815696975
-0.359068256998365	0.757882581524374	-0.323541041763627	-0.870534227755687	0.0792631912863636
-0.459673905457569	-0.701526587772079	-0.417577664032421	-0.725460726303266	-0.119132299176163
-0.338256179134518	-0.523565732643067	-0.254030052182524	-0.923505840608445	-0.192224240688976
0.8749875471263 8	0.174119064688993	0.836057900916614	-1.11590500805745	-0.211010116496191
-0.484106960369249	0.0705987421869745	-0.319827850867342	-0.867894407968447	0.051246727010685
-0.0221203060940465	-0.216026820080053	-0.0152729438559779	-0.341246696530392	-0.0538476142281317

The values are listed at the `Cell` centers. Particularly for irregular meshes, no specific ordering should be relied upon. Vector quantities are listed in multiple columns, one for each mesh dimension.

This problem again has an analytical solution that depends on the error function, but it's a bit more complicated due to the varying boundary conditions and the different horizontal diffusion length at different vertical positions

```
>>> x, y = mesh.getCellCenters()
>>> t = timeStepDuration * steps

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

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

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

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

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

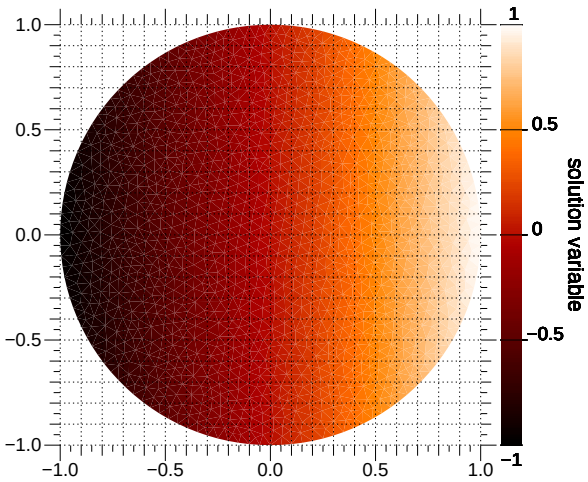
```
>>> ImplicitDiffusionTerm(coeff=D).solve(var=phi,
...                                      boundaryConditions=BCs)
```

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

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

Display the results if run as a script.

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



6.4 Module examples.diffusion.electrostatics

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

```
>>> from fipy import *

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

Given the electrostatic potential ϕ,

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

the permittivity ϵ,

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

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

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

and the charge density ρ,

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

The dimensionless Poisson equation is

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

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

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

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

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

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

and we solve for the electrostatic potential

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

This problem has the analytical solution

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

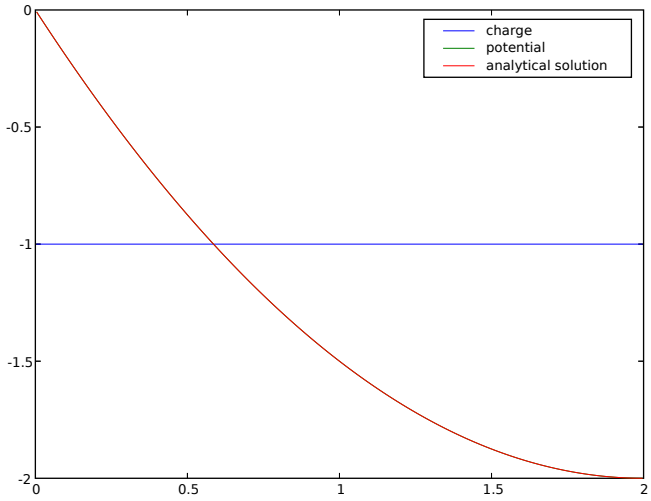
```
>>> x = mesh.getCellCenters()[0]
>>> analytical = CellVariable(mesh=mesh, name="analytical solution",
...                           value=(x**2)/2 - 2*x)
```

which has been satisifactorily obtained

```
>>> print potential.allclose(analytical, rtol = 2e-5, atol = 2e-5)
1
```

If we are running the example interactively, we view the result

```
>>> if __name__ == '__main__':
...     viewer = Viewer(vars=(charge, potential, analytical))
...     viewer.plot()
...     raw_input("Press any key to continue...")
```



Next, we segregate all of the electrons to right side of the domain

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

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

and again solve for the electrostatic potential

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

which now has the analytical solution

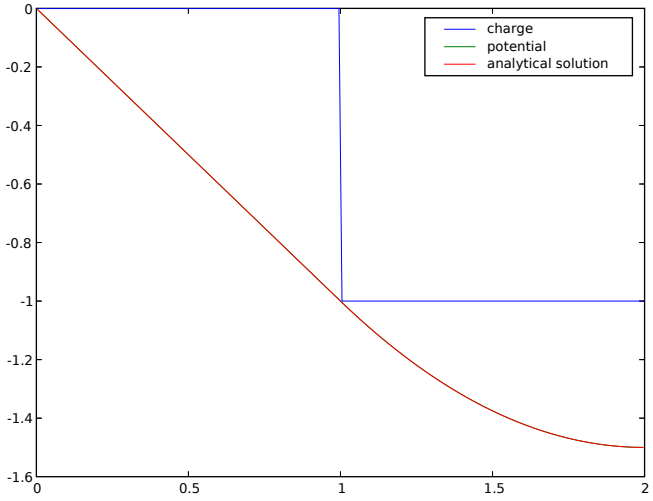
$$\psi(x) = \begin{cases} -x & \text{for } x \leq L/2, \\ \frac{(x-1)^2}{2} - x & \text{for } x > L/2. \end{cases}$$

```
>>> analytical.setValue(-x)
>>> analytical.setValue(((x-1)**2)/2 - x, where=x > L/2)
```

```
>>> print potential.allclose(analytical, rtol = 2e-5, atol = 2e-5)
1
```

and again view the result

```
>>> if __name__ == '__main__':
...     viewer.plot()
...     raw_input("Press any key to continue...")
```



Finally, we segregate all of the electrons to the left side of the domain

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

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

and again solve for the electrostatic potential

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

which has the analytical solution

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

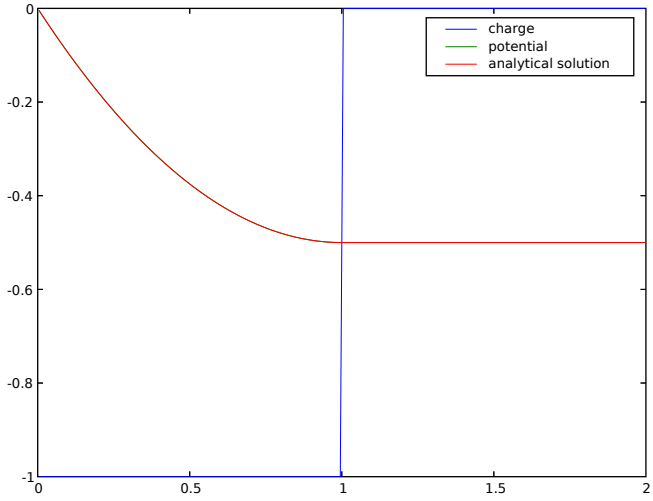
We again verify that the correct equilibrium is attained

```
>>> analytical.setValue((x**2)/2 - x)
>>> analytical.setValue(-0.5, where=x > L / 2)

>>> print potential.allclose(analytical, rtol = 2e-5, atol = 2e-5)
1
```

and once again view the result

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



6.5 Module examples.diffusion.nthOrder.input4thOrder1D

This example uses the `ImplicitDiffusionTerm` class to solve the equation

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

on a 1D mesh of length

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

We create an appropriate mesh

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

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

and initialize the solution variable to 0

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

For this problem, we impose the boundary conditions:

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

or

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

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

We initialize the steady-state equation

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

and use the `LinearLUSolver` for stability.

We perform one implicit timestep to achieve steady state

```
>>> eq.solve(var=var,
...          boundaryConditions=BCs,
...          solver=LinearLUSolver(tolerance=1e-11))
```

The analytical solution is:

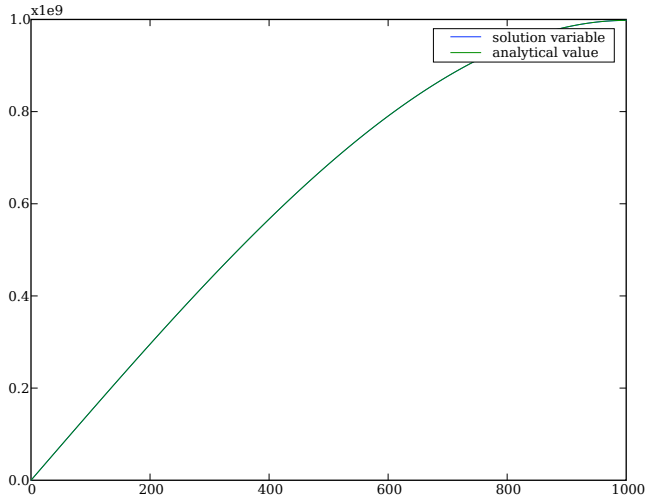
$$\phi = \frac{\alpha_4}{6} x^3 + \frac{\alpha_3}{2} x^2 + \left(\alpha_2 - \frac{\alpha_4}{2} L^2 - \alpha_3 L \right) x + \alpha_1$$

or

```
>>> analytical = CellVariable(mesh=mesh, name='analytical value')
>>> x = mesh.getCellCenters()[0]
>>> analytical.setValue(alpha4 / 6. * x**3 + alpha3 / 2. * x**2 + \
...                     (alpha2 - alpha4 / 2. * L**2 - alpha3 * L) * x + alpha1)
>>> print var.allclose(analytical, rtol=1e-4)
1
```

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

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



6.6 Module examples.diffusion.anisotropy

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

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

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

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

A new matrix is formed by rotating Γ' such that

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

and

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

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

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

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

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

Import a mesh previously created using Gmsh.

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

Set the center most cell to have a value.

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

Choose an orientation for the anisotropy.

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

Make the equation, viewer and solve.

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

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

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

>>> for i in range(40):
...     var.updateOld()
...     res = 1.
...
...     while res > 1e-2:
...         res = eqn.sweep(var, dt=dt)
...
...     if __name__ == '__main__':
...         viewer.plot()
...     time += dt
```

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

```
>>> X, Y = numerix.dot(mesh.getCellCenters(), CellVariable(mesh=mesh, rank=2, value=rotationMatrix)
>>> solution = mass * numerix.exp(-(X**2 / gamma_prime[0][0] + Y**2 / gamma_prime[1][1]) / (4 * time
>>> print max(abs((var - solution) / max(solution))) < 0.05
True
```

Convection Examples

7.1 Module examples.convection.exponential1D.mesh1D

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

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

with coefficients $D = 1$ and $\vec{u} = (10,)$, or

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

We define a 1D mesh

```
>>> from fipy import *

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

and impose the boundary conditions

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

or

```
>>> valueLeft = 0.
>>> valueRight = 1.
>>> boundaryConditions = (
...     FixedValue(faces=mesh.getFacesLeft(), value=valueLeft),
...     FixedValue(faces=mesh.getFacesRight(), value=valueRight),
...     )
```

The solution variable is initialized to `valueLeft`:

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

The equation is created with the `ImplicitDiffusionTerm` and `ExponentialConvectionTerm`. The scheme used by the convection term needs to calculate a Peclet number and thus the diffusion term instance must be passed to the convection term.

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

More details of the benefits and drawbacks of each type of convection term can be found in Section 3.5 "Numerical Schemes".

Essentially, the `ExponentialConvectionTerm` and `PowerLawConvectionTerm` will both handle most types of convection-diffusion cases, with the `PowerLawConvectionTerm` being more efficient.

We solve the equation

```
>>> eq.solve(var=var, boundaryConditions=boundaryConditions)
```

and test the solution against the analytical result

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

or

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

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

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

7.2 Module examples.convection.exponential1DSource.mesh1D

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

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

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

We define a 1D mesh

```
>>> from fipy import *

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

and impose the boundary conditions

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

or

```
>>> valueLeft = 0.
>>> valueRight = 1.
>>> boundaryConditions = (
...     FixedValue(faces=mesh.getFacesRight(), value=valueRight),
...     FixedValue(faces=mesh.getFacesLeft(), value=valueLeft),
...     )
```

The solution variable is initialized to `valueLeft`:

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

We define the convection-diffusion equation with source

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


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

and test the solution against the analytical result:

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

or

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

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

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

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

7.3 Module examples.convection.robin

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

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

The analytical solution for this equation is given by,

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

where

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

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

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

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

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

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

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

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

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

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

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

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

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

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

7.4 Module examples.convection.source

This example solves the following equation.

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

with $\phi(0) = 1$ at $x = 0$. The boundary condition at $x = L$ will require the implementation of an outflow boundary condition, which is not currently implemented in FiPy. An `ImplicitSourceTerm` object will be used to represent this term. The derivative of ϕ can be represented by a `ConvectionTerm` with a constant unitary velocity field from left to right. The following is an example code that includes a test against the analytical result.

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

```
>>> L = 10.
>>> nx = 5000
>>> dx =  L / nx
>>> mesh = Grid1D(dx=dx, nx=nx)
>>> phi0 = 1.0
>>> alpha = 1.0
>>> phi = CellVariable(name=r"$\phi$", mesh=mesh, value=phi0)
>>> solution = CellVariable(name=r"solution", mesh=mesh, value=phi0 * exp(-alpha * mesh.getCellCente

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

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

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

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

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

Phase Field Examples

The phase field method is a "diffuse interface" technique for modeling phase transformations and interface motion. Several good review articles have been written on the subject [2, 3, 4].

8.1 Module examples.phase.simple

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

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

$$\frac{1}{M_\phi}\frac{\partial \phi}{\partial t} = \kappa_\phi \nabla^2 \phi - \frac{\partial f}{\partial \phi} \tag{8.1}$$

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

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

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

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

and for the interpolation function is

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

We create a 1D solution mesh

```
>>> from fipy import *

>>> L = 1.
>>> nx = 400
>>> dx = L / nx


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

We create the phase field variable

```
>>> phase = CellVariable(name = "phase",
...                      mesh = mesh)
```

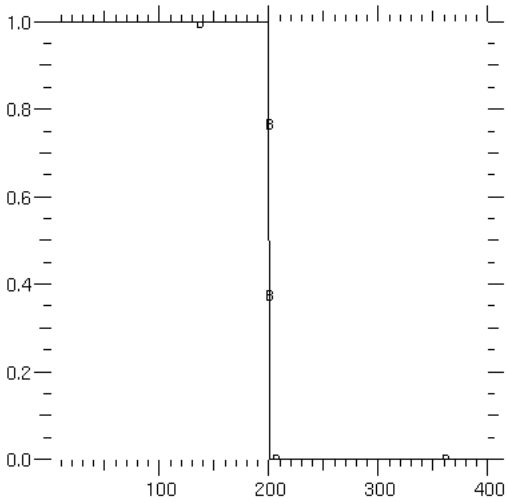
and set a step-function initial condition

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

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

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

```
>>> if __name__ == '__main__':
...     viewer = Viewer(vars = (phase,))
...     viewer.plot()
...     raw_input("Initial condition. Press <return> to proceed...")
```



We choose the parameter values,

```
>>> kappa = 0.0025
>>> W = 1.
>>> Lv = 1.
>>> Tm = 1.
>>> T = Tm
>>> enthalpy = Lv * (T - Tm) / Tm
```

We build the equation by assembling the appropriate terms. Since, with $T = T_M$ we are interested in a steady-state solution, we omit the transient term $(1/M_\phi)\frac{\partial \phi}{\partial t}$. The analytical solution for this steady-state phase field problem, in an infinite domain, is

$$\phi = \frac{1}{2}\left[1 - \tanh \frac{x - L/2}{2\sqrt{\kappa/W}}\right] \tag{8.2}$$

or

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

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

Note

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

The source term is

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

where $m_\phi \equiv -[W(1-2\phi) + 30\phi(1-\phi)L\frac{T-T_M}{T_M}]$. The simplest approach is to add this source explicitly

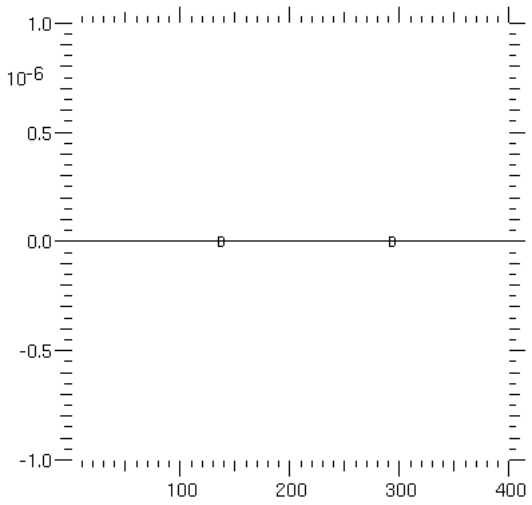
```
>>> mPhi = -((1 - 2 * phase) * W + 30 * phase * (1 - phase) * enthalpy)
>>> S0 = mPhi * phase * (1 - phase)
>>> eq = S0 + ImplicitDiffusionTerm(coeff=kappa)
```

After solving this equation

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

we obtain the surprising result that ϕ is zero everywhere.

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



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

The resolution to this problem is to apply relaxation to obtain the desired answer, i.e., the solution is allowed to relax in time from the initial condition to the desired equilibrium solution. To do so, we reintroduce the transient term from Equation (8.1)

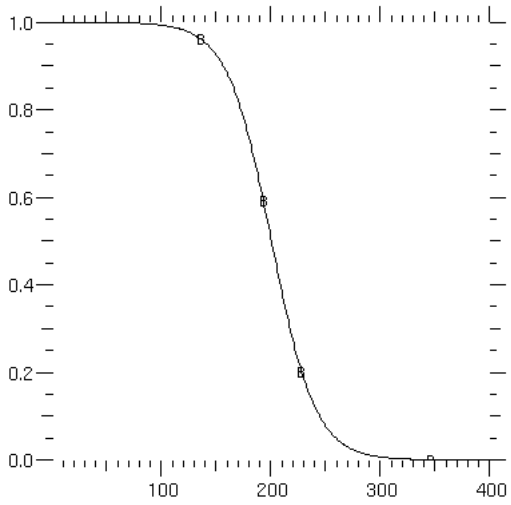
```
>>> eq = TransientTerm() == ImplicitDiffusionTerm(coeff=kappa) + S0
```

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

```
>>> for i in range(13):
...     eq.solve(var = phase)
...     if __name__ == '__main__':
...         viewer.plot()
```

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

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



Note

The solution is only found accurate to $\approx 4.3 \times 10^{-5}$ because the infinite-domain analytical solution (8.2) is not an exact representation for the solution in a finite domain of length L.

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

Solution performance can be improved if we exploit the dependence of the source on ϕ. By doing so, we can make the source semi-implicit, improving the rate of convergence over the fully explicit approach. The source can only be semi-implicit because we employ sparse linear algebra routines to solve the PDEs, i.e., there is no fully implicit way to represent a term like ϕ^4 in the linear set of equations $\mathsf{M}\vec{\phi} - \vec{b} = 0$. By linearizing a source as $S = S_0 - S_1\phi$, we make it more implicit by adding the coefficient S_1 to the matrix diagonal. For numerical stability, this linear coefficient must never be negative.

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

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

Note

Because `mPhi` is a variable field, the quantities `(mPhi > 0)` and `(mPhi < 0)` evaluate to variable *fields* of ones and zeroes, instead of simple boolean values.

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

Note

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

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


>>> for i in range(8):
...     eq.solve(var = phase)
>>> print phase.allclose(analyticalArray, rtol = 1e-4, atol = 1e-4)
1
>>> if __name__ == '__main__':
...     viewer.plot()
...     raw_input("Kobayashi, semi-implicit. Press <return> to proceed...")
```

In general, the best convergence is obtained when the linearization gives a good representation of the relationship between the source and the dependent variable. The best practical advice is to perform a Taylor expansion of the source about the previous value of the dependent variable such that $S = S_{\text{old}} + \left.\frac{\partial S}{\partial \phi}\right|_{\text{old}} (\phi - \phi_{\text{old}}) = (S - \frac{\partial S}{\partial \phi}\phi)_{\text{old}} + \left.\frac{\partial S}{\partial \phi}\right|_{\text{old}} \phi$. Now, if our source term is represented by $S = S_0 + S_1\phi$, then $S_1 = \left.\frac{\partial S}{\partial \phi}\right|_{\text{old}}$ and $S_0 = (S - \frac{\partial S}{\partial \phi}\phi)_{\text{old}} = S_{\text{old}} - S_1\phi_{\text{old}}$. In this way, the linearized source will be tangent to the curve of the actual source as a function of the dependendent variable.

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

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

and

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

or

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

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

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

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

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

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

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

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

and thus must redeclare ϕ on the new mesh

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

We choose the parameter values appropriate for nickel, given in [25]

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

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

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

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

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

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

and make a new viewer

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

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

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

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

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

and then quench by 1 K

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

In order to have a stable numerical solution, the interface must not move more than one grid point per time step, we thus set the timestep according to the grid spacing Δx, the linear kinetic coefficient β, and the undercooling $|T_m - T|$

Again we use the "`sweep()`" method as a replacement for "`solve()`".

```
>>> velocity = beta * abs(Tm - T()) # cm / s
>>> timeStep = .1 * dx / velocity # s
>>> elapsed = 0
>>> while elapsed < 0.1 * L / velocity:
...     phase.updateOld()
...     res = 1e+10
...     while res > 1e-5:
...         res = eq.sweep(var=phase, dt=timeStep)
...     elapsed += timeStep
...     if __name__ == '__main__':
...         viewer2.plot()
```

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

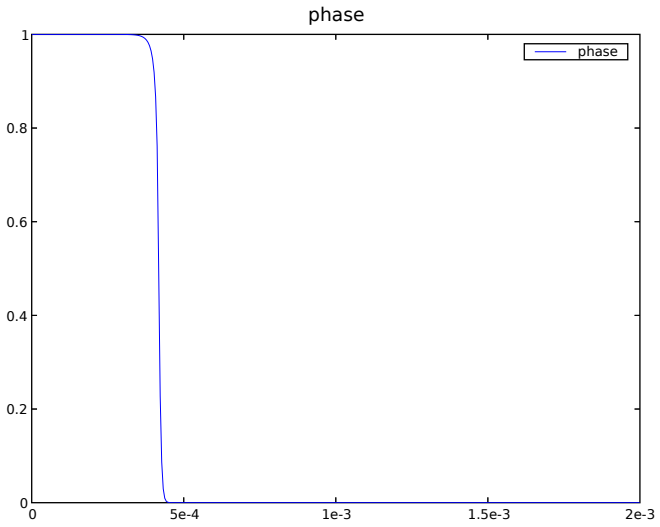
```
>>> print phase.allclose(analyticalArray, rtol = 5, atol = 2e-3)
1
```

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

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

```
>>> print abs(1 - V_fit / velocity) < 3.3e-2
True
>>> print abs(1 - d_fit / delta) < 2e-2
True
```

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



8.2 Module examples.phase.binary

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

```
>>> from fipy import *

>>> nx = 400
>>> dx = 5e-6 # cm
>>> L = nx * dx
>>> mesh = Grid1D(dx=dx, nx=nx)
```

The Helmholtz free energy functional can be written as the integral [2, 4, 26]

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

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

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

composition C

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

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

and temperature[2] T

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

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

$$\frac{\partial \phi}{\partial t} = M_\phi \left(\kappa_\phi \nabla^2 \phi - \frac{\partial f}{\partial \phi} \right) \tag{8.3}$$

for phase and

$$\frac{\partial C}{\partial t} = \nabla \cdot \left(M_C \nabla \frac{\partial f}{\partial C} \right) \tag{8.4}$$

for solute.

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

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

where $f_A^L(T)$, $f_B^L(T)$, $f_A^S(T)$, and $f_B^S(T)$ are the free energy densities of the pure components. There are a variety of choices for the interpolation function $p(\phi)$ and the barrier function $g(\phi)$, such as those shown in `examples.phase.simple.input`

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

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

The desired thermodynamic model can then be applied to obtain $f(\phi, C, T)$, such as for a regular solution,

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

where

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

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

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

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

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

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

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

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

and likewise for component B.

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

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

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

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

With these assumptions

$$\begin{aligned}\frac{\partial f}{\partial \phi} &= (1 - C)\frac{\partial f_A}{\partial \phi} + C\frac{\partial f_B}{\partial \phi} \\ &= \left\{(1 - C)\frac{L_A\left(T - T_M^A\right)}{T_M^A} + C\frac{L_B\left(T - T_M^B\right)}{T_M^B}\right\} p'(\phi) + \left\{(1 - C)\frac{W_A}{2} + C\frac{W_B}{2}\right\} g'(\phi)\end{aligned} \tag{8.5}$$

and

$$\begin{aligned}\frac{\partial f}{\partial C} &= \left[f_B(\phi, T) + \frac{RT}{V_m}\ln C\right] - \left[f_A(\phi, T) + \frac{RT}{V_m}\ln(1 - C)\right] \\ &= \left[\mu_B(\phi, C, T) - \mu_A(\phi, C, T)\right]/V_m\end{aligned} \tag{8.6}$$

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

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

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

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

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

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

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

and define the averages

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

We can now linearize the source exactly as before

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

Using the same gradient energy coefficient and phase field mobility

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

we define the phase field equation

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

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

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

for j in range(faces):
```

```
        flux[j] = ((Mc[j+.5] + Mc[j-.5]) / 2) \
          * (deltaChemPot(phase[j+.5], C[j+.5], T) \
            - deltaChemPot(phase[j-.5], C[j-.5], T)) / dx

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

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

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

Although the second syntax would essentially work as written, such an explicit implementation would be very slow. In order to take advantage of FiPy's implicit solvers, it is necessary to reduce Eq. (8.4) to the canonical form of Eq. (3.2), hence we must expand Eq. (8.6) as

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

In either bulk phase, $\nabla p(\phi) = \nabla g(\phi) = 0$, so we can then reduce Eq. (8.4) to

$$\begin{aligned}\frac{\partial C}{\partial t} &= \nabla \cdot \left(M_C \nabla \left\{\frac{RT}{V_m}\left[\ln C - \ln(1-C)\right]\right\}\right) \\ &= \nabla \cdot \left[\frac{M_C RT}{C(1-C)V_m}\nabla C\right]\end{aligned} \tag{8.7}$$

and, by comparison with Fick's second law

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

we can associate the mobility M_C with the intrinsic diffusivity D by $M_C \equiv DC(1-C)V_m/RT$ and write Eq. (8.4) as

$$\begin{aligned}\frac{\partial C}{\partial t} &= \nabla \cdot (D \nabla C) \\ &\quad + \nabla \cdot \left(\frac{DC(1-C)V_m}{RT}\left\{\left[\frac{L_B\left(T - T_M^B\right)}{T_M^B} - \frac{L_A\left(T - T_M^A\right)}{T_M^A}\right]\nabla p(\phi) + \frac{W_B - W_A}{2}\nabla g(\phi)\right\}\right).\end{aligned} \tag{8.8}$$

The first term is clearly a `DiffusionTerm`. The second is less obvious, but by factoring out C, we can see that this is a `ConvectionTerm` with a velocity

$$\vec{u}_\phi = \frac{D(1-C)V_m}{RT}\left\{\left[\frac{L_B\left(T - T_M^B\right)}{T_M^B} - \frac{L_A\left(T - T_M^A\right)}{T_M^A}\right]\nabla p(\phi) + \frac{W_B - W_A}{2}\nabla g(\phi)\right\}$$

due to phase transformation, such that

$$\frac{\partial C}{\partial t} = \nabla \cdot (D \nabla C) + \nabla \cdot (C \vec{u}_\phi)$$

or

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

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


>>> diffusionEq = (TransientTerm()
...                 == ImplicitDiffusionTerm(coeff=D)
...                 + PowerLawConvectionTerm(coeff=phaseTransformationVelocity))
```

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

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

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

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

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

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

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

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

The phase fraction is predicted by the lever rule

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

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

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

In this simple, binary, ideal solution case, we can derive explicit expressions for the solidus and liquidus compositions. In general, this may not be possible or practical. In that event, the root-finding facilities in SciPy can be used. We'll need a function to return the two conditions for equilibrium

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

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

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

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

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

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

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

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

We plot the result against the sharp interface solution

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

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

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

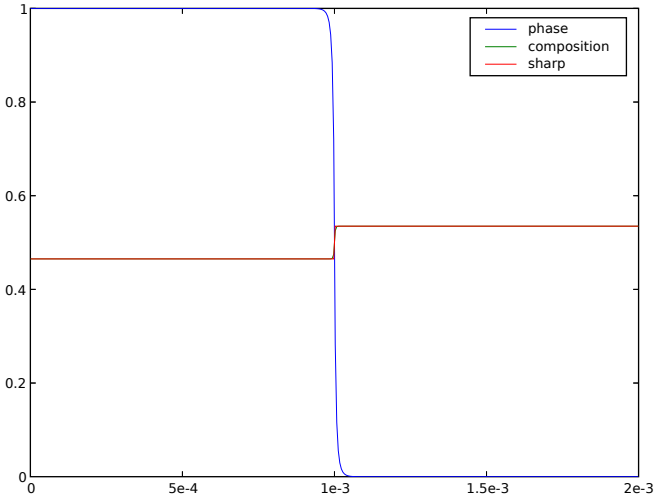
```
>>> dt = 1.e2
```

Because the phase field equation is coupled to the composition through `enthalpy` and `W` and the diffusion equation is coupled to the phase field through `phaseTransformationVelocity`, it is necessary sweep this non-linear problem to convergence. We use the "residual" of the equations (a measure of how well they think they have solved the given set of linear equations) as a test for how long to sweep. Because of the `ConvectionTerm`, the solution matrix for `diffusionEq` is asymmetric and cannot be solved by the default `LinearPCGSolver`. Therefore, we use a `LinearLUSolver` for this equation.

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

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

```
>>> phase.updateOld()
>>> C.updateOld()
>>> phaseRes = 1e+10
>>> diffRes = 1e+10
>>> while phaseRes > 1e-3 or diffRes > 1e-3:
...     phaseRes = phaseEq.sweep(var=phase, dt=dt)
...     diffRes = diffusionEq.sweep(var=C, dt=dt, solver=solver)
>>> if __name__ == '__main__':
...     viewer.plot()
...     raw_input("stationary phase field")
```



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

```
>>> print Cs.allclose(C[0], atol=2e-4)
1
>>> print Cl.allclose(C[nx-1], atol=2e-4)
1
```

and that the phase fraction remains unchanged

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

while conserving mass overall

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

We now quench by ten degrees

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

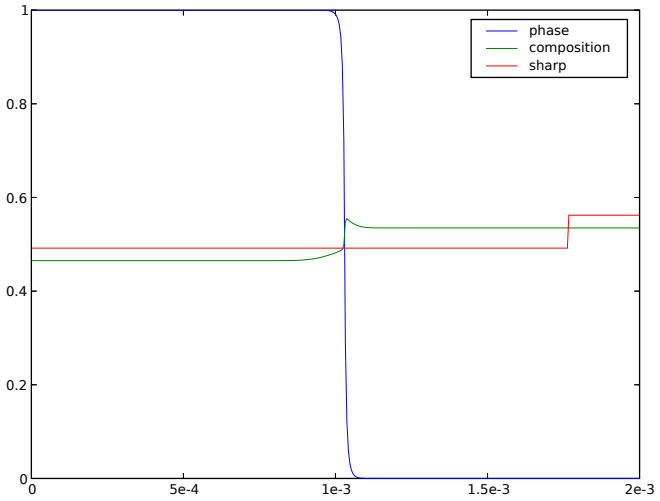
```
>>> sharp.setValue(Cs, where=x < L * fraction)
>>> sharp.setValue(Cl, where=x >= L * fraction)
```

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

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

>>> for i in range(100):
...     phase.updateOld()
...     C.updateOld()
...     phaseRes = 1e+10
...     diffRes = 1e+10
...     while phaseRes > 1e-3 or diffRes > 1e-3:
...         phaseRes = phaseEq.sweep(var=phase, dt=dt)
...         diffRes = diffusionEq.sweep(var=C, dt=dt, solver=solver)
...     if __name__ == '__main__':
...         viewer.plot()

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



We see that the composition on either side of the interface approach the sharp-interface solidus and liquidus, but it will take a great many more timesteps to reach equilibrium. If we waited sufficiently long, we could again verify the final concentrations and phase fraction against the expected values.

8.3 Module examples.phase.quaternary

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

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

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

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

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

interstitial components $C_0 \ldots C_M$

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

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

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

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

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

and temperature T

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

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

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

where

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

is the gas constant. As in the binary case,

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

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

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

and the "double well" function

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

We consider a very simplified model that has partial molar volumes $\bar{V}_0 = \cdots = \bar{V}_M = 0$ for the "interstitials" and $\bar{V}_{M+1} = \cdots = \bar{V}_N = 1$ for the "substitutionals". This approximation has been used in a number of models where density effects are ignored, including the treatment of electrons in electrodeposition processes [27, 28]. Under these constraints

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

and

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

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

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

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

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

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

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

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

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


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

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

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

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

We create the phase equation

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

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

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

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

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

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

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

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

The canonical form of the substitutional diffusion equations is

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

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

We start with a sharp phase boundary

$$\xi = \begin{cases} 1 & \text{for } x \leq L/2, \\ 0 & \text{for } x > L/2, \end{cases}$$

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

and with uniform concentration fields, initially equal to the average of the solidus and liquidus concentrations

```
>>> for Cj in interstitials + substitutionals:
...     Cj.setValue((Cj.S + Cj.L) / 2.)
```

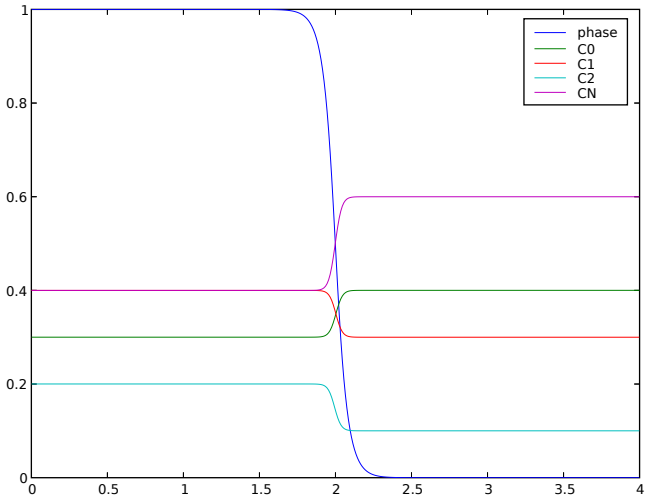
If we're running interactively, we create a viewer

```
>>> if __name__ == '__main__':
...     viewer = Viewer(vars=([phase]
...                           + interstitials + substitutionals
...                           + [solvent]),
...                     datamin=0, datamax=1)
...     viewer.plot()
```

and again iterate to equilibrium

```
>>> solver = LinearLUSolver(tolerance=1e-3)
```

```
>>> dt = 10000
>>> for i in range(5):
...     for field in [phase] + substitutionals + interstitials:
...         field.updateOld()
...     phase.equation.solve(var = phase, dt = dt)
...     for field in substitutionals + interstitials:
...         field.equation.solve(var = field,
...                              dt = dt,
...                              solver = solver)
...     if __name__ == '__main__':
...         viewer.plot()
```



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

```
>>> ends = take(phase, (0,-1))
>>> allclose(ends, (1.0, 0.0), rtol = 1e-5, atol = 1e-5)
1
```

and that the concentration fields have appropriately segregated into their equilibrium values in each phase

```
>>> equilibrium = True
>>> for Cj in interstitials + substitutionals:
...     ends = take(Cj, (0,-1))
...     equilibrium &= allclose(ends, (Cj.S, Cj.L), rtol = 3e-3, atol = 3e-3)
>>> print equilibrium
1
```

8.4 Module examples.phase.anisotropy

To convert a liquid material to a solid, it must be cooled to a temperature below its melting point (known as "undercooling" or "supercooling"). The rate of solidification is often assumed (and experimentally found) to be proportional to the undercooling. Under the right circumstances, the solidification front can become unstable, leading to dendritic patterns. Warren, Kobayashi, Lobkovsky and Carter [29] have described a phase field model ("Allen-Cahn", "non-conserved Ginsberg-Landau", or "model A" of Hohenberg & Halperin) of such a system, including the effects of discrete crystalline orientations (anisotropy).

We start with a regular 2D Cartesian mesh

```
>>> from fipy import *
>>> dx = dy = 0.025
>>> if __name__ == '__main__':
...     nx = ny = 500
... else:
...     nx = ny = 20
>>> mesh = Grid2D(dx=dx, dy=dy, nx=nx, ny=ny)
```

and we'll take fixed timesteps

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

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

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

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

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

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

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

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

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

The governing equation for the phase field is

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

where

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

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

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

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

```
>>> alpha = 0.015
>>> c = 0.02
>>> N = 6.
>>> theta = pi / 8.
>>> psi = theta + arctan2(phase.getFaceGrad()[1],
...                       phase.getFaceGrad()[0])
>>> Phi = tan(N * psi / 2)
>>> PhiSq = Phi**2
>>> beta = (1. - PhiSq) / (1. + PhiSq)
>>> DbetaDpsi = -N * 2 * Phi / (1 + PhiSq)
>>> Ddia = (1.+ c * beta)
>>> Doff = c * DbetaDpsi
>>> D = alpha**2 * (1.+ c * beta) * (Ddia * (( 1, 0),
...                                          ( 0, 1)) + Doff * (( 0,-1),
...                                                            ( 1, 0)))
```

With these expressions defined, we can construct the phase field equation as

```
>>> tau = 3e-4
>>> kappa1 = 0.9
>>> kappa2 = 20.
>>> phaseEq = (TransientTerm(tau)
...           == DiffusionTerm(D)
...           + ImplicitSourceTerm((phase - 0.5 - kappa1 / pi * arctan(kappa2 * dT))
...                                * (1 - phase)))
```

We seed a circular solidified region in the center

```
>>> radius = dx * 5.
>>> C = (nx * dx / 2, ny * dy / 2)
>>> x, y = mesh.getCellCenters()
>>> phase.setValue(1., where=((x - C[0])**2 + (y - C[1])**2) < radius**2)
```

and quench the entire simulation domain below the melting point

```
>>> dT.setValue(-0.5)
```

In a real solidification process, dendritic branching is induced by small thermal fluctuations along an otherwise smooth surface, but the granularity of the `Mesh` is enough "noise" in this case, so we don't need to explicitly introduce randomness, the way we did in the Cahn-Hilliard problem.

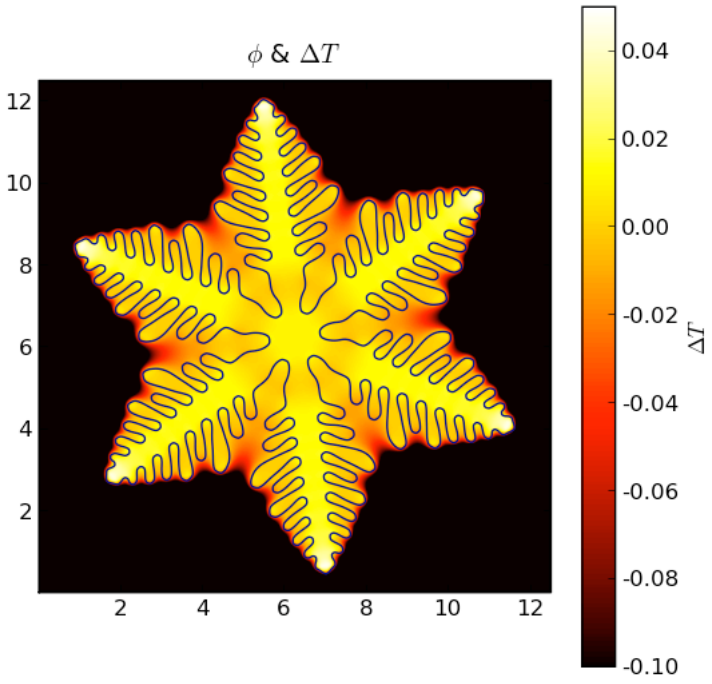
FiPy's viewers are utilitarian, striving to let the user see *something*, regardless of their operating system or installed packages, so you won't be able to simultaneously view two fields "out of the box", but, because all of Python is accessible and FiPy is object oriented, it is not hard to adapt one of the existing viewers to create a specialized display:

```
>>> if __name__ == "__main__":
...     try:
...         import pylab
...         class DendriteViewer(Matplotlib2DGridViewer):
...             def __init__(self, phase, dT, title=None, limits={}, **kwlimits):
...                 self.phase = phase
...                 self.contour = None
...                 Matplotlib2DGridViewer.__init__(self, vars=(dT,), title=title,
...                                                 cmap=pylab.cm.hot,
...                                                 limits=limits, **kwlimits)
...
...             def _plot(self):
...                 Matplotlib2DGridViewer._plot(self)
...
...                 if self.contour is not None:
...                     for c in self.contour.collections:
...                         c.remove()
...
...                 mesh = self.phase.getMesh()
...                 shape = mesh.getShape()
...                 x, y = mesh.getCellCenters()
...                 z = self.phase.getValue()
...                 x, y, z = [a.reshape(shape, order="FORTRAN") for a in (x, y, z)]
...
...                 self.contour = pylab.contour(x, y, z, (0.5,))
...
...         viewer = DendriteViewer(phase=phase, dT=dT,
...                                 title=r"%s & %s" % (phase.name, dT.name),
...                                 datamin=-0.1, datamax=0.05)
```

```
...     except ImportError:
...         viewer = MultiViewer(viewers=(Viewer(vars=phase),
...                                       Viewer(vars=dT,
...                                              datamin=-0.5,
...                                              datamax=0.5)))
```

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

```
>>> if __name__ == '__main__':
...     steps = 10000
... else:
...     steps = 10
>>> for i in range(steps):
...     phase.updateOld()
...     dT.updateOld()
...     phaseEq.solve(phase, dt=dt)
...     heatEq.solve(dT, dt=dt)
...     if __name__ == "__main__" and (i % 10 == 0):
...         viewer.plot()
```



The non-uniform temperature results from the release of latent heat at the solidifying interface. The dendrite arms grow fastest where the temperature gradient is steepest.

We note that this FiPy simulation is written in about 50 lines of code (excluding the custom viewer), compared with over 800 lines of (fairly lucid) FORTRAN code used for the figures in [29].

8.5 Module examples.phase.impingement.mesh40x1

In this example we solve a coupled phase and orientation equation on a one dimensional grid. This is another aspect of the model of Warren, Kobayashi, Lobkovsky and Carter [29]

```
>>> from fipy import *

>>> nx = 40
>>> Lx = 2.5 * nx / 100.
>>> dx = Lx / nx
>>> mesh = Grid1D(dx=dx, nx=nx)
```

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

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

where

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

and the orientation equation is given by

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

where

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

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

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

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

The parameters for these equations are

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

The system is held isothermal at

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

and is initially solid everywhere

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

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

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

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

```
>>> theta.setValue(0., where=mesh.getCellCenters()[0] > Lx / 2.)
```

The `phase` equation is built in the following way.

```
>>> mPhiVar = phase - 0.5 + temperature * phase * (1 - phase)
```

The source term is linearized in the manner demonstrated in `examples.phase.simple.input` (Kobayashi, semi-implicit).

```
>>> thetaMag = theta.getOld().getGrad().getMag()
>>> implicitSource = mPhiVar * (phase - (mPhiVar < 0))
>>> implicitSource += (2 * s + epsilon**2 * thetaMag) * thetaMag
```

The `phase` equation is constructed.

```
>>> phaseEq = TransientTerm(phaseTransientCoeff) \
...   == ExplicitDiffusionTerm(alpha**2) \
...      - ImplicitSourceTerm(implicitSource) \
...      + (mPhiVar > 0) * mPhiVar * phase
```

The `theta` equation is built in the following way. The details for this equation are fairly involved, see J.A. Warren *et al.*. The main detail is that a source must be added to correct for the discretization of `theta` on the circle.

```
>>> phaseMod = phase + ( phase < thetaSmallValue ) * thetaSmallValue
>>> phaseModSq = phaseMod * phaseMod
>>> expo = epsilon * beta * theta.getGrad().getMag()
>>> expo = (expo < 100.) * (expo - 100.) + 100.
>>> pFunc = 1. + exp(-expo) * (mu / epsilon - 1.)

>>> phaseFace = phase.getArithmeticFaceValue()
>>> phaseSq = phaseFace * phaseFace
>>> gradMag = theta.getFaceGrad().getMag()
>>> eps = 1. / gamma / 10.
>>> gradMag += (gradMag < eps) * eps
>>> IGamma = (gradMag > 1. / gamma) * (1 / gradMag - gamma) + gamma
>>> diffusionCoeff = phaseSq * (s * IGamma + epsilon**2)
```

The source term requires the evaluation of the face gradient without the modular operator. A method of `ModularVariable`, `getFaceGradNoMod()`, evelautes the gradient without modular arithmetic.

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

Finally the `theta` equation can be constructed.

```
>>> thetaEq = TransientTerm(thetaTransientCoeff * phaseModSq * pFunc) == \
...           ImplicitDiffusionTerm(diffusionCoeff) \
...           + sourceCoeff
```

If the example is run interactively, we create viewers for the phase and orientation variables.

```
>>> if __name__ == '__main__':
...     phaseViewer = Viewer(vars=phase, datamin=0., datamax=1.)
...     thetaProductViewer = Viewer(vars=theta,
...                                 datamin=-pi, datamax=pi)
...     phaseViewer.plot()
...     thetaProductViewer.plot()
```

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

```
>>> steps = 10
>>> for i in range(steps):
...     theta.updateOld()
...     phase.updateOld()
...     thetaEq.solve(theta, dt = timeStepDuration)
...     phaseEq.solve(phase, dt = timeStepDuration)
```

```
...     if __name__ == '__main__':
...         phaseViewer.plot()
...         thetaProductViewer.plot()
```

The solution is compared with test data. The test data was created with `steps = 10` with a FORTRAN code written by Ryo Kobayashi for phase field modeling. The following code opens the file `mesh40x1.gz` extracts the data and compares it with the `theta` variable.

```
>>> import os
>>> testData = loadtxt(os.path.splitext(__file__)[0] + '.gz')
>>> print theta.allclose(testData)
1
```

8.6 Module examples.phase.impingement.mesh20x20

In the following examples, we solve the same set of equations as in:

```
$ examples/phase/impingement/mesh40x1/input.py
```

with different initial conditions and a 2D mesh:

```
>>> from fipy.tools.parser import parse

>>> numberOfElements = parse('--numberOfElements', action = 'store',
...                          type = 'int', default = 400)
>>> numberOfSteps = parse('--numberOfSteps', action = 'store',
...                       type = 'int', default = 10)


>>> from fipy import *

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

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

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

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

The parameters for this example are

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

The system is held isothermal at

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

and is initialized to liquid everywhere

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

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

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

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

```
>>> x, y = mesh.getCellCenters()
>>> for a, b, thetaValue in ((0., 0.,  2. * pi / 3.),
...                          (L, 0., -2. * pi / 3.),
...                          (0., L, -2. * pi / 3. + 0.3),
...                          (L, L,  2. * pi / 3.)):
...     segment = (x - a)**2 + (y - b)**2 < (L / 2.)**2
...     phase.setValue(1., where=segment)
...     theta.setValue(thetaValue, where=segment)
```

The `phase` equation is built in the following way. The source term is linearized in the manner demonstrated in `examples.phase.simple.input` (Kobayashi, semi-implicit). Here we use a function to build the equation, so that it can be reused later.

```
>>> def buildPhaseEquation(phase, theta):
...
...     mPhiVar = phase - 0.5 + temperature * phase * (1 - phase)
...     thetaMag = theta.getOld().getGrad().getMag()
...     implicitSource = mPhiVar * (phase - (mPhiVar < 0))
...     implicitSource += (2 * s + epsilon**2 * thetaMag) * thetaMag
...
...     return TransientTerm(phaseTransientCoeff) == \
...                ExplicitDiffusionTerm(alpha**2) \
...                - ImplicitSourceTerm(implicitSource) \
...                + (mPhiVar > 0) * mPhiVar * phase

>>> phaseEq = buildPhaseEquation(phase, theta)
```

The `theta` equation is built in the following way. The details for this equation are fairly involved, see J.A. Warren *et al.*. The main detail is that a source must be added to correct for the discretization of `theta` on the circle. The source term requires the evaluation of the face gradient without the modular operators.

```
>>> def buildThetaEquation(phase, theta):
...
...     phaseMod = phase + ( phase < thetaSmallValue ) * thetaSmallValue
...     phaseModSq = phaseMod * phaseMod
...     expo = epsilon * beta * theta.getGrad().getMag()
...     expo = (expo < 100.) * (expo - 100.) + 100.
...     pFunc = 1. + exp(-expo) * (mu / epsilon - 1.)
...
...     phaseFace = phase.getArithmeticFaceValue()
...     phaseSq = phaseFace * phaseFace
...     gradMag = theta.getFaceGrad().getMag()
...     eps = 1. / gamma / 10.
...     gradMag += (gradMag < eps) * eps
...     IGamma = (gradMag > 1. / gamma) * (1 / gradMag - gamma) + gamma
...     diffusionCoeff = phaseSq * (s * IGamma + epsilon**2)
...
...     thetaGradDiff = theta.getFaceGrad() - theta.getFaceGradNoMod()
...     sourceCoeff = (diffusionCoeff * thetaGradDiff).getDivergence()
...
...     return TransientTerm(thetaTransientCoeff * phaseModSq * pFunc) == \
...                ImplicitDiffusionTerm(diffusionCoeff) \
...                + sourceCoeff


>>> thetaEq = buildThetaEquation(phase, theta)
```

If the example is run interactively, we create viewers for the phase and orientation variables. Rather than viewing the raw orientation, which is not meaningful in the liquid phase, we weight the orientation by the phase

```
>>> if __name__ == '__main__':
...     phaseViewer = Viewer(vars=phase, datamin=0., datamax=1.)
...     thetaProd = -pi + phase * (theta + pi)
...     thetaProductViewer = Viewer(vars=thetaProd,
...                                 datamin=-pi, datamax=pi)
...     phaseViewer.plot()
...     thetaProductViewer.plot()
```

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

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


>>> testData = resize(testData, (mesh.getNumberOfCells(),))
```

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

```
>>> for i in range(steps):
...     theta.updateOld()
...     phase.updateOld()
...     thetaEq.solve(theta, dt=timeStepDuration)
...     phaseEq.solve(phase, dt=timeStepDuration)
...     if __name__ == '__main__':
...         phaseViewer.plot()
...         thetaProductViewer.plot()
```

The solution is compared against Ryo Kobayashi's test data

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

The following code shows how to restart a simulation from some saved data. First, reset the variables to their original values.

```
>>> phase.setValue(0)
>>> theta.setValue(-pi + 0.0001)
>>> x, y = mesh.getCellCenters()
>>> for a, b, thetaValue in ((0., 0.,  2. * pi / 3.),
...                          (L, 0., -2. * pi / 3.),
...                          (0., L, -2. * pi / 3. + 0.3),
...                          (L, L,  2. * pi / 3.)):
...     segment = (x - a)**2 + (y - b)**2 < (L / 2.)**2
...     phase.setValue(1., where=segment)
...     theta.setValue(thetaValue, where=segment)
```

Step through half the time steps.

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

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

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

We save the variables to disk.

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

and then recall them to test the data pickling mechanism

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

and finish the iterations,

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

The solution is compared against Ryo Kobayashi's test data

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

Level Set Examples

The Level Set Method (LSM) is a popular interface tracking method. Further details of the LSM and descriptions of the algorithms used in FiPy can be found in Sethian's Level Set book [30].

9.1 Module examples.levelSet.distanceFunction.mesh1D

Here we create a level set variable in one dimension. The level set variable calculates its value over the domain to be the distance from the zero level set. This can be represented succinctly in the following equation with a boundary condition at the zero level set such that,

$$\frac{\partial \phi}{\partial x} = 1$$

with the boundary condition, $\phi = 0$ at $x = L/2$. The solution to this problem will be demonstrated in the following script. Firstly, setup the parameters.

```
>>> from fipy import *

>>> dx = 0.5
>>> nx = 10
```

Construct the mesh.

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

Construct a `distanceVariable` object.

```
>>> var = DistanceVariable(name='level set variable',
...                        mesh=mesh,
...                        value=-1,
...                        hasOld=1)
>>> x = mesh.getCellCenters()[0]
>>> var.setValue(1, where=x > dx * nx / 2)
```

Once the initial positive and negative regions have been initialized the `calcDistanceFunction()` method can be used to recalculate `var` as a distance function from the zero level set.

```
>>> var.calcDistanceFunction()
```

The problem can then be solved by executing the `solve()` method of the equation.

```
>>> if __name__ == '__main__':
...     viewer = Viewer(vars=var, datamin=-5., datamax=5.)
...     viewer.plot()
```

The result can be tested with the following commands.

```
>>> print allclose(var, x - dx * nx / 2)
1
```

9.2 Module examples.levelSet.distanceFunction.circle

Here we solve the level set equation in two dimensions for a circle. The 2D level set equation can be written,

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

and the boundary condition for a circle is given by, $\phi = 0$ at $(x - L/2)^2 + (y - L/2)^2 = (L/4)^2$. The solution to this problem will be demonstrated in the following script. Firstly, setup the parameters.

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

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

Construct the mesh.

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

Construct a `distanceVariable` object.

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

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

```
>>> var.calcDistanceFunction()


>>> if __name__ == '__main__':
...     viewer = Viewer(vars=var, datamin=-5., datamax=5.)
...     viewer.plot()
```

The result can be tested with the following commands.

```
>>> dY = dy / 2.
>>> dX = dx / 2.
>>> mm = min (dX, dY)
>>> m1 = dY * dX / sqrt(dY**2 + dX**2)
>>> def evalCell(phix, phiy, dx, dy):
...     aa = dy**2 + dx**2
...     bb = -2 * ( phix * dy**2 + phiy * dx**2)
...     cc = dy**2 * phix**2 + dx**2 * phiy**2 - dx**2 * dy**2
...     sqr = sqrt(bb**2 - 4. * aa * cc)
...     return ((-bb - sqr) / 2. / aa,  (-bb + sqr) / 2. / aa)
>>> v1 = evalCell(-dY, -m1, dx, dy)[0]
>>> v2 = evalCell(-m1, -dX, dx, dy)[0]
>>> v3 = evalCell(m1,  m1,  dx, dy)[1]
>>> v4 = evalCell(v3, dY, dx, dy)[1]
>>> v5 = evalCell(dX, v3, dx, dy)[1]
>>> MASK = -1000
>>> trialValues = MA.masked_values((
...     MASK,  MASK, MASK, MASK, MASK, MASK, MASK, MASK, MASK, MASK, MASK,
...     MASK,  MASK, MASK, MASK,-3*dY,-3*dY,-3*dY, MASK, MASK, MASK, MASK,
...     MASK,  MASK, MASK,   v1,  -dY,  -dY,  -dY,   v1, MASK, MASK, MASK,
...     MASK,  MASK,   v2,  -m1,   m1,   dY,   m1,  -m1,   v2, MASK, MASK,
...     MASK, -dX*3,  -dX,   m1,   v3,   v4,   v3,   m1,  -dX,-dX*3, MASK,
...     MASK, -dX*3,  -dX,   dX,   v5, MASK,   v5,   dX,  -dX,-dX*3, MASK,
...     MASK, -dX*3,  -dX,   m1,   v3,   v4,   v3,   m1,  -dX,-dX*3, MASK,
...     MASK,  MASK,   v2,  -m1,   m1,   dY,   m1,  -m1,   v2, MASK, MASK,
...     MASK,  MASK, MASK,   v1,  -dY,  -dY,  -dY,   v1, MASK, MASK, MASK,
...     MASK,  MASK, MASK, MASK,-3*dY,-3*dY,-3*dY, MASK, MASK, MASK, MASK,
...     MASK,  MASK, MASK, MASK, MASK, MASK, MASK, MASK, MASK, MASK, MASK),
...     MASK)
>>> print var.allclose(trialValues)
1
```

9.3 Module examples.levelSet.advection.mesh1D

This example first solves the distance function equation in one dimension:

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

with $\phi = 0$ at $x = L/5$. The variable is then advected with,

$$\frac{\partial \phi}{\partial t} + \vec{u} \cdot \nabla \phi = 0$$

The scheme used in the `AdvectionTerm` preserves the `var` as a distance function.

The solution to this problem will be demonstrated in the following script. Firstly, setup the parameters.

```
>>> from fipy import *

>>> velocity = 1.
>>> dx = 1.
>>> nx = 10
>>> timeStepDuration = 1.
>>> steps = 2
>>> L = nx * dx
>>> interfacePosition = L / 5.
```

Construct the mesh.

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

Construct a `distanceVariable` object.

```
>>> var = DistanceVariable(name='level set variable',
...                        mesh=mesh,
...                        value=-1.,
...                        hasOld=1)
>>> var.setValue(1., where=mesh.getCellCenters()[0] > interfacePosition)
>>> var.calcDistanceFunction()
```

The `advectionEquation` is constructed.

```
>>> advEqn = buildAdvectionEquation(advectionCoeff=velocity)
```

The problem can then be solved by executing a serious of time steps.

```
>>> if __name__ == '__main__':
...     viewer = Viewer(vars=var, datamin=-10., datamax=10.)
...     viewer.plot()
...     for step in range(steps):
...         var.updateOld()
...         advEqn.solve(var, dt=timeStepDuration)
...         viewer.plot()
```

The result can be tested with the following code:

```
>>> for step in range(steps):
...     var.updateOld()
...     advEqn.solve(var, dt=timeStepDuration)
>>> x = mesh.getCellCenters()[0]
>>> distanceTravelled = timeStepDuration * steps * velocity
>>> answer = x - interfacePosition - timeStepDuration * steps * velocity
>>> answer = where(x < distanceTravelled,
...                x[0] - interfacePosition, answer)
>>> print var.allclose(answer)
1
```

9.4 Module examples.levelSet.advection.circle

This example first imposes a circular distance function:

$$\phi(x, y) = \left[\left(x - \frac{L}{2}\right)^2 + \left(y - \frac{L}{2}\right)^2\right]^{1/2} - \frac{L}{4}$$

The variable is advected with,

$$\frac{\partial \phi}{\partial t} + \vec{u} \cdot \nabla \phi = 0$$

The scheme used in the `_AdvectionTerm` preserves the `var` as a distance function. The solution to this problem will be demonstrated in the following script. Firstly, setup the parameters.

```
>>> from fipy import *

>>> L = 1.
>>> N = 25
>>> velocity = 1.
>>> cfl = 0.1
>>> velocity = 1.
>>> distanceToTravel = L / 10.
>>> radius = L / 4.
>>> dL = L / N
>>> timeStepDuration = cfl * dL / velocity
>>> steps = int(distanceToTravel / dL / cfl)
```

Construct the mesh.

```
>>> mesh = Grid2D(dx=dL, dy=dL, nx=N, ny=N)
```

Construct a `distanceVariable` object.

```
>>> var = DistanceVariable(
...     name = 'level set variable',
...     mesh = mesh,
```

```
...     value = 1.,
...     hasOld = 1)
```

Initialise the `distanceVariable` to be a circular distance function.

```
>>> x, y = mesh.getCellCenters()
>>> initialArray = sqrt((x - L / 2.)**2 + (y - L / 2.)**2) - radius
>>> var.setValue(initialArray)
```

The `advectionEquation` is constructed.

```
>>> advEqn = buildAdvectionEquation(
...     advectionCoeff=velocity)
```

The problem can then be solved by executing a serious of time steps.

```
>>> if __name__ == '__main__':
...     viewer = Viewer(vars=var, datamin=-radius, datamax=radius)
...     viewer.plot()
...     for step in range(steps):
...         var.updateOld()
...         advEqn.solve(var, dt=timeStepDuration)
...         viewer.plot()
```

The result can be tested with the following commands.

```
>>> for step in range(steps):
...     var.updateOld()
...     advEqn.solve(var, dt=timeStepDuration)
>>> x = array(mesh.getCellCenters()[0])
>>> distanceTravelled = timeStepDuration * steps * velocity
>>> answer = initialArray - distanceTravelled
>>> answer = where(answer < 0., -1001., answer)
>>> solution = where(answer < 0., -1001., array(var))
>>> allclose(answer, solution, atol=4.7e-3)
1
```

If the `AdvectionEquation` is built with the `_HigherOrderAdvectionTerm` the result is more accurate,

```
>>> var.setValue(initialArray)
>>> advEqn = buildHigherOrderAdvectionEquation(
...     advectionCoeff = velocity)
>>> for step in range(steps):
...     var.updateOld()
...     advEqn.solve(var, dt=timeStepDuration)
>>> solution = where(answer < 0., -1001., array(var))
>>> allclose(answer, solution, atol=1.02e-3)
1
```

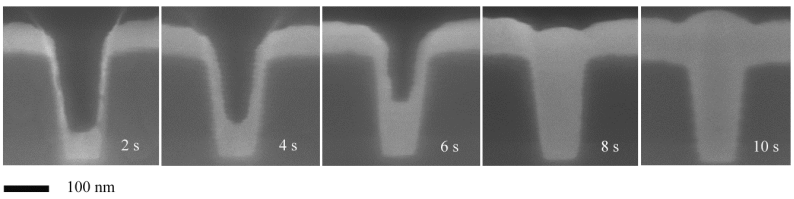
Superconformal Electrodeposition Examples

The Damascene Process

State of the art manufacturing of semiconductor devices involves the electrodeposition of copper for on-chip wiring of integrated circuits. In the Damascene process interconnects are fabricated by first patterning trenches in a dielectric medium and then filling by metal electrodeposition over the entire wafer surface. This metalization process, pioneered by IBM, depends on the use of electrolyte additives that effect the local metal deposition rate.

Superfill

The additives in the electrolyte affect the local deposition rate in such a way that bottom-up filling occurs in trenches or vias. This process, known as superconformal electrodeposition or superfill, is demonstrated in the following figure. The figure shows sequential images of bottom-up superfilling of submicrometer trenches by copper deposition from an electrolyte containing PEG-SPS-Cl. Preferential metal deposition at the bottom of the trenches followed by bump formation above the filled trenches is evident.



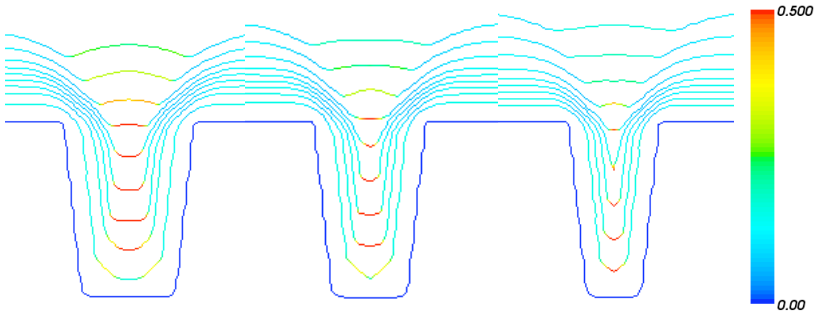
The CEAC Mechanism

This process has been demonstrated to depend critically on the inclusion of additives in the electrolyte. Recent publications propose Curvature Enhanced Accelerator Coverage (CEAC) as the mechanism behind the superfilling process [5]. In this mechanism, molecules that accelerate local metal deposition displace molecules that inhibit local metal deposition on the metal/electrolyte interface. For electrolytes that yield superconformal filling of fine features, this buildup happens relatively slowly because the concentration of accelerator species is much more dilute compared to the inhibitor species in the electrolyte. The mechanism that leads to the increased rate of metal deposition along the bottom of the filling trench is the concurrent local increase of the accelerator coverage due to decreasing local surface area, which scales with the local curvature (hence the name of the mechanism). A good overview of this mechanism can be found in [31].

Using FiPy to model Superfill

Example 9.5 provides a simple way to use FiPy to model the superfill process. The example includes a detailed description of the governing equations and feature geometry. It requires the user to import and execute a function at the python prompt. The model parameters can be passed as arguments to this function. In future all superfill examples will be provided with this type of interface. Example 9.8 has the same functionality as 9.5 but demonstrates how to write a new script in the case where the existing suite of scripts do not meet the required needs.

In general it is a good idea to obtain the Mayavi plotting package for which a specialized superfill viewer class has been created, see Chapter 2 "Installation and Usage". The other standard viewers mentioned in Chapter 2 "Installation and Usage" are still adequate although they do not give such clear images that are tailored for the superfill problem. The images below demonstrate the Mayavi viewing capability. Each contour represents sequential positions of the interface and the color represents the concentration of accelerator as a surfactant. The areas of high surfactant concentration have an increased deposition rate.



9.5 Module examples.levelSet.electroChem.simpleTrenchSystem

This input file is a demonstration of the use of FiPy for modeling electrodeposition using the CEAC mechanism. The material properties and experimental parameters used are roughly those that have been previously published [32]. To run this example from the base fipy directory type:

```
$ examples/levelSet/electroChem/simpleTrenchSystem.py
```

at the command line. The results of the simulation will be displayed and the word `finished` in the terminal at the end of the simulation. In order to alter the number of timesteps, the python function that encapsulates the system of equations must first be imported (at the python command line),

```
>>> from examples.levelSet.electroChem.simpleTrenchSystem import runSimpleTrenchSystem
```

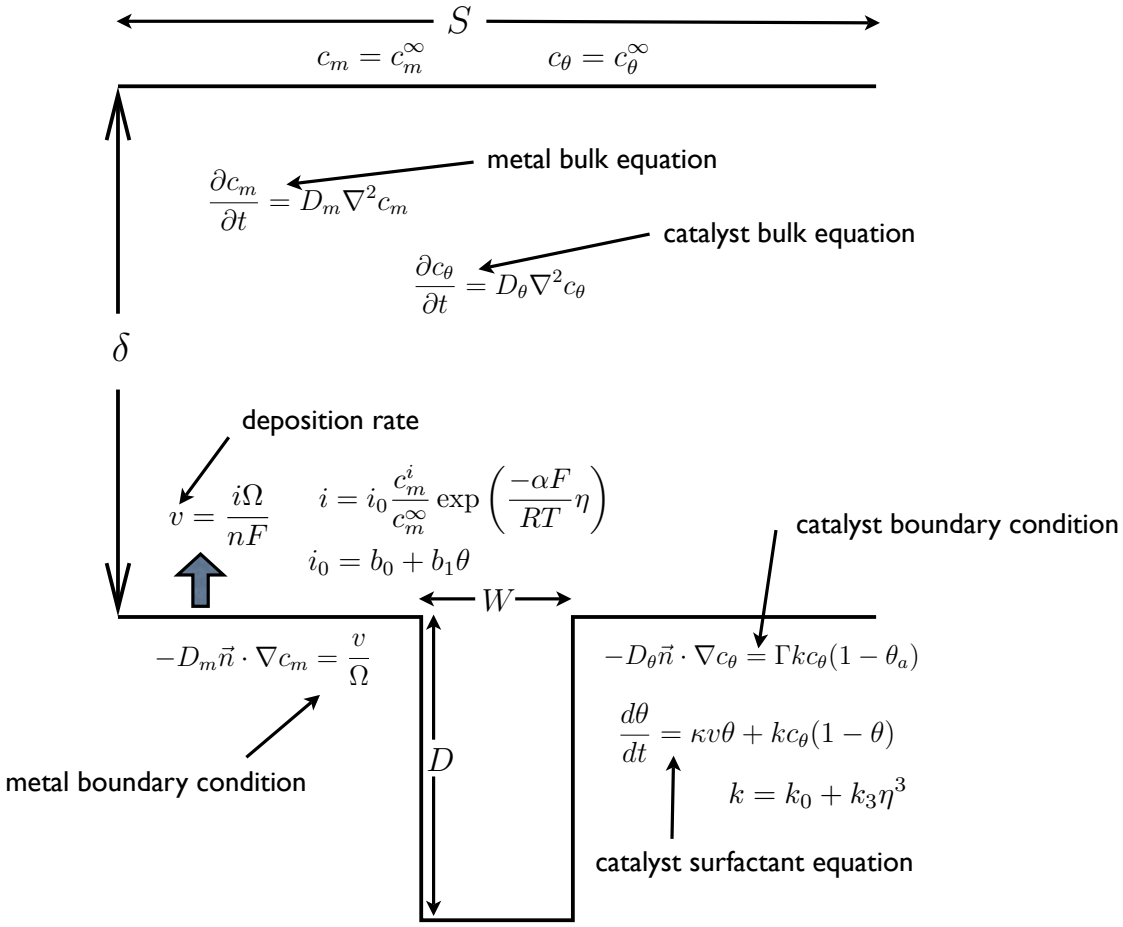
and then the function can be run with a different number of time steps with the `numberOfSteps` argument as follows,

```
>>> runSimpleTrenchSystem(numberOfSteps=5, displayViewers=False)
1
```

Change the `displayViewers` argument to `True` if you wish to see the results displayed on the screen. Example 9.8 gives explanation for writing new scripts or modifying existing scripts that are encapsulated by functions. Any argument parameter can be changed. For example if the initial catalyst coverage is not 0, then it can be reset,

```
>>> runSimpleTrenchSystem(catalystCoverage=0.1, displayViewers=False)
0
```

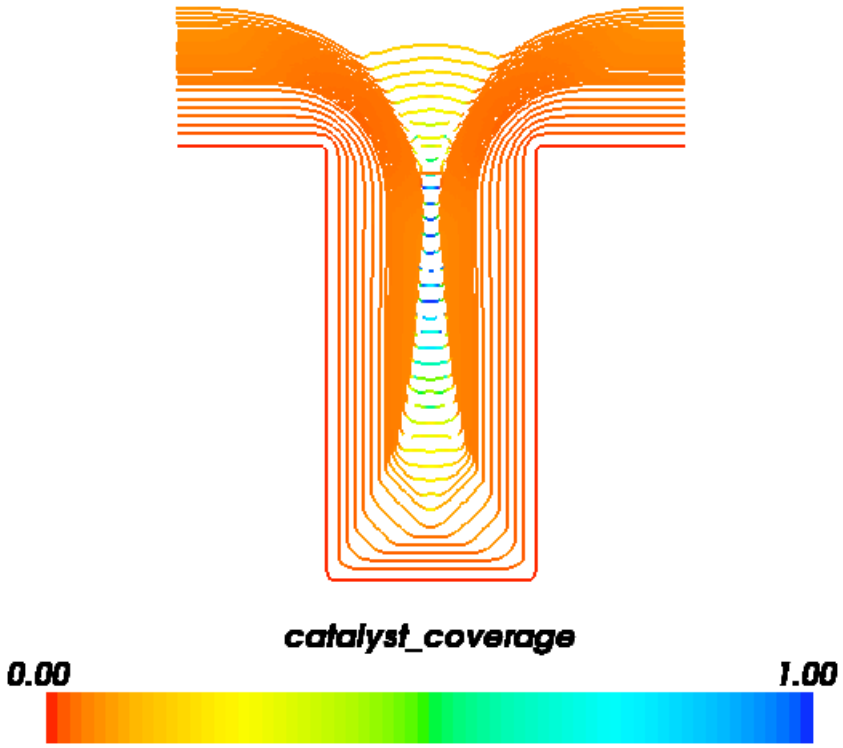
The following image shows a schematic of a trench geometry along with the governing equations for modeling electrodeposition with the CEAC mechanism. All of the given equations are implemented in the `runSimpleTrenchSystem` function. As stated above, all the parameters in the equations can be changed with function arguments.



The following table shows the symbols used in the governing equations and their corresponding arguments to the `runSimpleTrenchSystem` function. The boundary layer depth is intentionally small in this example in order not to complicate the mesh. Further examples will simulate more realistic boundary layer depths but will also have more complex meshes requiring the `gmsh` software.

Symbol	Description	Keyword Argument	Value	Unit
		Deposition Rate Parameters		
v	deposition rate			$m\ s^{-1}$
i	current density			$A\ m^{-2}$
Ω	molar volume	`molarVolume`	7.1×10^{-6}	$m^3\ mol^{-1}$
n	ion charge	`charge`	2	
F	Faraday's constant	`faradaysConstant`	9.6×10^{-4}	$C\ mol^{-1}$
i_0	exchange current density			$A\ m^{-2}$
α	transfer coefficient	`transferCoefficient`	0.5	
η	overpotential	`overpotential`	-0.3	V
R	gas constant	`gasConstant`	8.314	$J\ K^{-1}\ mol^{-1}$
T	temperature	`temperature`	298.0	K
b_0	current density for θ^0	`currentDensity0`	0.26	$A\ m^{-2}$
b_1	current density for θ	`currentDensity1`	45.0	$A\ m^{-2}$
		Metal Ion Parameters		
c_m	metal ion concentration	`metalConcentration`	250.0	$mol\ m^{-3}$
c_m^∞	far field metal ion concentration	`metalConcentration`	250.0	$mol\ m^{-3}$
D_m	metal ion diffusion coefficient	`metalDiffusion`	5.6×10^{-10}	$m^2\ s^{-1}$
		Catalyst Parameters		
θ	catalyst surfactant concentration	`catalystCoverage`	0.0	
c_θ	bulk catalyst concentration	`catalystConcentration`	5.0×10^{-3}	$mol\ m^{-3}$
c_θ^∞	far field catalyst concentration	`catalystConcentration`	5.0×10^{-3}	$mol\ m^{-3}$
D_θ	catalyst diffusion coefficient	`catalystDiffusion`	1.0×10^{-9}	$m^2\ s^{-1}$
Γ	catalyst site density	`siteDensity`	9.8×10^{-6}	$mol\ m^{-2}$
k	rate constant			$m^3\ mol^{-1}\ s^{-1}$
k_0	rate constant for η^0	`rateConstant0`	1.76	$m^3\ mol^{-1}\ s^{-1}$
k_3	rate constant for η^3	`rateConstant3`	-245.0×10^{-6}	$m^3\ mol^{-1}\ s^{-1}\ V^{-3}$
		Geometry Parameters		
D	trench depth	`trenchDepth`	0.5×10^{-6}	m
D/W	trench aspect ratio	`aspectRatio`	2.0	
S	trench spacing	`trenchSpacing`	0.6×10^{-6}	m
δ	boundary layer depth	`boundaryLayerDepth`	0.3×10^{-6}	m
		Simulation Control Parameters		
	computational cell size	`cellSize`	0.1×10^{-7}	m
	number of time steps	`numberOfSteps`	5	
	whether to display the viewers	`displayViewers`	`True`	

If the MayaVi plotting software is installed (see Chapter 2) then a plot should appear that is updated every 20 time steps and will eventually resemble the image below.



Functions

`runSimpleTrenchSystem(`*faradaysConstant*=`96000.0`, *gasConstant*=`8.314`, *transferCoefficient*=`0.5`, *rateConstant0*=`1.76`, *rateConstant3*=`-0.000245`, *catalystDiffusion*=`1e-09`, *siteDensity*=`9.8e-06`, *molarVolume*=`7.1e-06`, *charge*=`2`, *metalDiffusion*=`5.6e-10`, *temperature*=`298.0`, *overpotential*=`-0.3`, *metalConcentration*=`250.0`, *catalystConcentration*=`0.005`, *catalystCoverage*=`0.0`, *currentDensity0*=`0.26`, *currentDensity1*=`45.0`, *cellSize*=`1e-08`, *trenchDepth*=`5e-07`, *aspectRatio*=`2.0`, *trenchSpacing*=`6e-07`, *boundaryLayerDepth*=`3e-07`, *numberOfSteps*=`5`, *displayViewers*=`True`)

9.6 Module examples.levelSet.electroChem.gold

This input file is a demonstration of the use of FiPy for modeling gold superfill. The material properties and experimental parameters used are roughly those that have been previously published [33]. To run this example from the base fipy directory type:

```
$ examples/levelSet/electroChem/gold.py
```

at the command line. The results of the simulation will be displayed and the word `finished` in the terminal at the end of the simulation. The simulation will only run for 10 time steps. In order to alter the number of timesteps, the python function that encapsulates the system of equations must first be imported (at the python command line),

```
>>> from examples.levelSet.electroChem.gold import runGold
```

and then the function can be run with a different number of time steps with the `numberOfSteps` argument as follows,

```
>>> runGold(numberOfSteps=10, displayViewers=False)
1
```

Change the `displayViewers` argument to `True` if you wish to see the results displayed on the screen. This example has a more realistic default boundary layer depth and thus requires `gmsh` to construct a more complex mesh.

There are a few differences between the gold superfill model presented in this example and Example 9.5. Most default values have changed to account for a different metal ion (gold) and catalyst (lead). In this system the catalyst is not present in the electrolyte but instead has a non-zero initial coverage. Thus quantities associated with bulk catalyst and catalyst accumulation are not defined. The current density is given by,

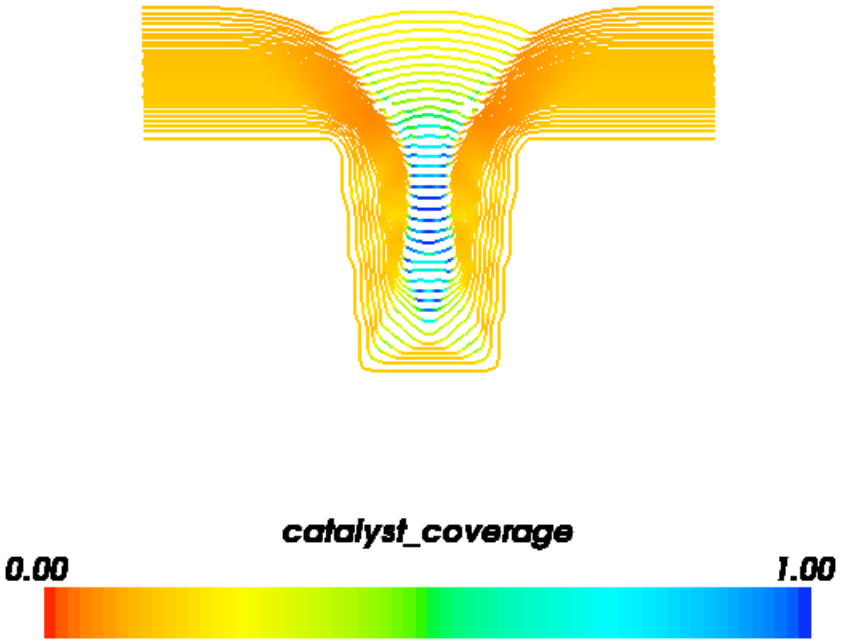
$$i = \frac{c_m}{c_m^\infty}\left(b_0 + b_1\theta\right).$$

The more common representation of the current density includes an exponential part. Here it is buried in b_0 and b_1. The governing equation for catalyst evolution includes a term for catalyst consumption on the interface and is given by

$$\dot{\theta} = Jv\theta - k_c v\theta$$

where k_c is the consumption coefficient (`consumptionRateConstant`). The trench geometry is also given a slight taper, given by `taperAngle`.

If the MayaVi plotting software is installed (see Chapter 2) then a plot should appear that is updated every 10 time steps and will eventually resemble the image below.



Functions

`runGold(`*faradaysConstant*`=96000.0`, *consumptionRateConstant*`=2600000.0`, *molarVolume*`=1.021e-05`, *charge*`=1.0`, *metalDiffusion*`=1.7e-09`, *metalConcentration*`=20.0`, *catalystCoverage*`=0.15`, *currentDensity0*`=0.48`, *currentDensity1*`=10.4`, *cellSize*`=1e-08`, *trenchDepth*`=2e-07`, *aspectRatio*`=1.47`, *trenchSpacing*`=5e-07`, *boundaryLayerDepth*`=9e-05`, *numberOfSteps*`=10`, *taperAngle*`=6.0`, *displayViewers*`=True)`

9.7 Module examples.levelSet.electroChem.leveler

This input file is a demonstration of the use of FiPy for modeling copper superfill with leveler and accelerator additives. The material properties and experimental parameters used are roughly those that have been previously published [34]. To run this example from the base fipy directory type:

```
$ examples/levelSet/electroChem/leveler.py
```

at the command line. The results of the simulation will be displayed and the word `finished` in the terminal at the end of the simulation. The simulation will only run for 200 time steps. In order to alter the number

of timesteps, the python function that encapsulates the system of equations must first be imported (at the python command line),

```
>>> from examples.levelSet.electroChem.leveler import runLeveler
```

and then the function can be executed with a different number of time steps by changing the `numberOfSteps` argument as follows,

```
>>> runLeveler(numberOfSteps=10, displayViewers=False, cellSize=0.25e-7)
1
```

Change the `displayViewers` argument to `True` if you wish to see the results displayed on the screen. This example requires `gmsh` to construct the mesh.

This example models the case when suppressor, accelerator and leveler additives are present in the electrolyte. The suppressor is assumed to absorb quickly compared with the other additives. Any unoccupied surface sites are immediately covered with suppressor. The accelerator additive has more surface affinity than suppressor and is thus preferential adsorbed. The accelerator can also remove suppressor when the surface reaches full coverage. Similarly, the leveler additive has more surface affinity than both the suppressor and accelerator. This forms a simple set of assumptions for understanding the behavior of these additives.

The following is a complete description of the equations for the model described here. Any equations that have been omitted are the same as those given in Example 9.5. The current density is governed by

$$i = \frac{c_m}{c_m^\infty} \sum_j \left[i_j \theta_j \left(\exp \frac{-\alpha_j F \eta}{RT} - \exp \frac{(1-\alpha_j) F \eta}{RT} \right) \right]$$

where j represents S for suppressor, A for accelerator, L for leveler and V for vacant. This model assumes a linear interpolation between the three cases of complete coverage for each additive or vacant substrate. The governing equations for the surfactants are given by,

$$\dot{\theta_L} = \kappa v \theta_L + k_l^+ c_L \left(1 - \theta_L\right) - k_L^- v \theta_L,$$

$$\dot{\theta_a} = \kappa v \theta_A + k_A^+ c_A \left(1 - \theta_A - \theta_L\right) - k_L c_L \theta_A - k_A^- \theta_A^{q-1},$$

$$\theta_S = 1 - \theta_A - \theta_L$$

and

$$\theta_V = 0.$$

It has been found experimentally that $i_L = i_S$.

If the surface reaches full coverage, the equations do not naturally prevent the coverage rising above full coverage due to the curvature terms. Thus, when $\theta_L + \theta_A = 1$ then the equation for accelerator becomes $\dot{\theta_A} = -\dot{\theta_L}$ and when $\theta_L = 1$, the equation for leveler becomes $\dot{\theta_L} = -k_L^- v \theta_L$.

The parameters k_A^+, k_A^- and q are both functions of η given by,

$$k_A^+ = k_{A0}^+ \exp \frac{-\alpha_k F \eta}{RT},$$

and

$$k_A^- = B_d + \frac{A}{\exp\left(B_a \left(\eta + V_d\right)\right)} + \exp\left(B_b \left(\eta + V_d\right)\right)$$

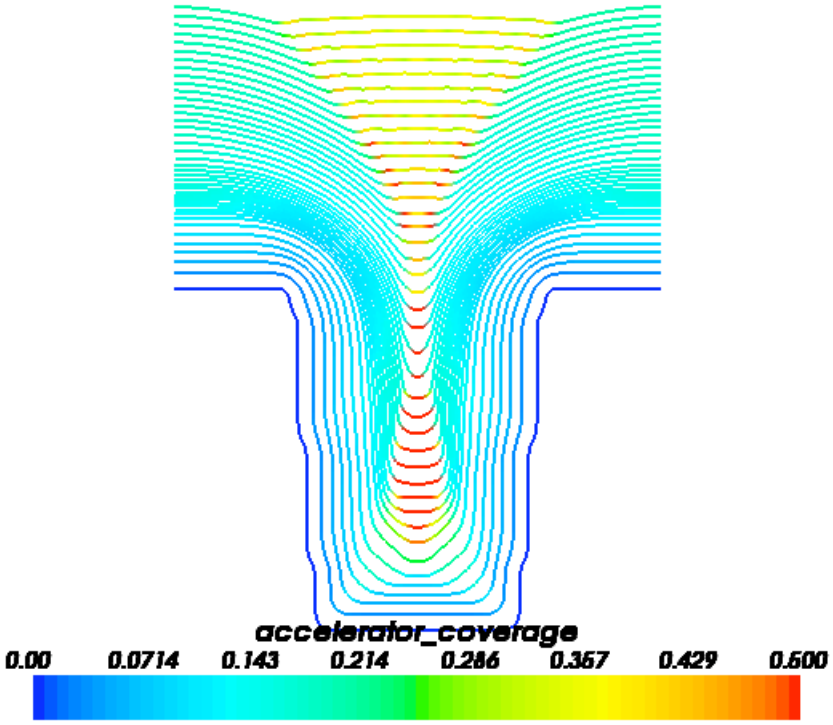
and

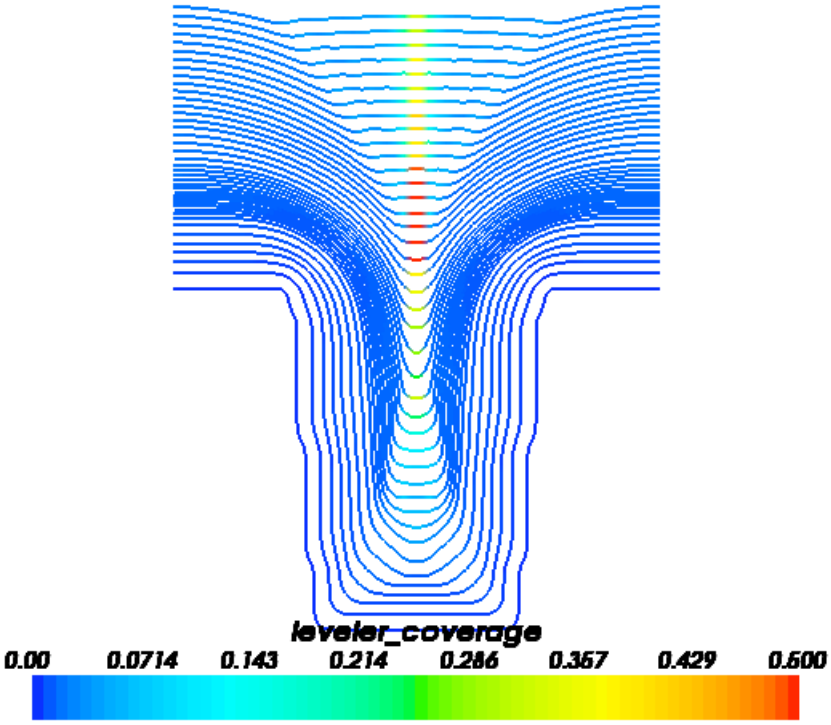
$$q = m * \eta + b.$$

The following table shows the symbols used in the governing equations and their corresponding arguments for the `runLeveler` function.

Symbol	Description	Keyword Argument	Value	Unit
		Deposition Rate Parameters		
v	deposition rate			m s^{-1}
i_A	accelerator current density	`i0Accelerator`		A m^{-2}
i_L	leveler current density	`i0Leveler`		A m^{-2}
Ω	molar volume	`molarVolume`	7.1×10^{-6}	m^3 mol^{-1}
n	ion charge	`charge`	2	
F	Faraday's constant	`faradaysConstant`	9.6×10^{-4}	C mol^{-1}
i_0	exchange current density			A m^{-2}
α_A	accelerator transfer coefficient	`alphaAccelerator`	0.4	
α_S	leveler transfer coefficient	`alphaLeveler`	0.5	
η	overpotential	`overpotential`	-0.3	V
R	gas constant	`gasConstant`	8.314	J K mol^{-1}
T	temperature	`temperature`	298.0	K
		Ion Parameters		
c_I	ion concentration	`ionConcentration`	250.0	mol m^{-3}
c_I^∞	far field ion concentration	`ionConcentration`	250.0	mol m^{-3}
D_I	ion diffusion coefficient	`ionDiffusion`	5.6×10^{-10}	m^2 s^{-1}
		Accelerator Parameters		
θ_A	accelerator coverage	`acceleratorCoverage`	0.0	
c_A	accelerator concentartion	`acceleratorConcentration`	5.0×10^{-3}	mol m^{-3}
c_A^∞	far field accelerator concentration	`acceleratorConcentration`	5.0×10^{-3}	mol m^{-3}
D_A	catalyst diffusion coefficient	`catalystDiffusion`	1.0×10^{-9}	m^2 s^{-1}
Γ_A	accelerator site density	`siteDensity`	9.8×10^{-6}	mol m^{-2}
k_A^+	accelerator adsorption			m^3 mol^{-1} s^{-1}
k_{A0}^+	accelerator adsorption coeff	`kAccelerator0`	$2.6times10^{-4}$	m^3 mol^{-1} s^{-1}
α_k	accelerator adsorption coeff	`alphaAdsorption`	0.62	
k_A^-	accelerator consumption coeff			
B_a	experimental parameter	`Bd`	-40.0	
B_b	experimental parameter	`Bd`	60.0	
V_d	experimental parameter	`Bd`	9.8×10^{-2}	
B_d	experimental parameter	`Bd`	8.0×10^{-4}	
		Geometry Parameters		
D	trench depth	`trenchDepth`	0.5×10^{-6}	m
D/W	trench aspect ratio	`aspectRatio`	2.0	
S	trench spacing	`trenchSpacing`	0.6×10^{-6}	m
δ	boundary layer depth	`boundaryLayerDepth`	0.3×10^{-6}	m
		Simulation Control Parameters		
	computational cell size	`cellSize`	0.1×10^{-7}	m
	number of time steps	`numberOfSteps`	5	
	whether to display the viewers	`displayViewers`	`True`	

The following images show accelerator and leveler contour plots that can be obtained by running this example.


accelerator_coverage
0.00
0.0714
0.143
0.214
0.286
0.357
0.429
0.500



Functions

`runLeveler(`*kLeveler*=`0.018`, *bulkLevelerConcentration*=`0.02`, *cellSize*=`1e-08`, *rateConstant*=`0.00026`, *initialAcceleratorCoverage*=`0.0`, *levelerDiffusionCoefficient*=`5e-10`, *numberOfSteps*=`400`, *displayRate*=`10`, *displayViewers*=`True`)

9.8 Module examples.levelSet.electroChem.howToWriteAScript

This input file demonstrates how to create a new superfill script if the existing suite of scripts do not meet the required needs. It provides the functionality of Example 9.5. To run this example from the base fipy directory type:

```
$ examples/levelSet/electroChem/howToWriteAScript.py --numberOfElements=10000 --numberOfSteps=800
```

at the command line. The results of the simulation will be displayed and the word `finished` in the terminal at the end of the simulation. To obtain this example in a plain script file in order to edit and run type:

```
$ python setup.py copy_script --From examples/levelSet/electroChem/howToWriteAScript.py --To myScri
```

in the base FiPy directory. The file `myScript.py` will contain the script.

The following is an explicit explanation of the input commands required to set up and run the problem. At the top of the file all the parameter values are set. Their use will be explained during the instantiation of various objects and are the same as those explained in Example 9.5. The following parameters (all in S.I. units) represent,

physical constants,

```
>>> faradaysConstant = 9.6e4
>>> gasConstant = 8.314
>>> transferCoefficient = 0.5
```

properties associated with the catalyst species,

```
>>> rateConstant0 = 1.76
>>> rateConstant3 = -245e-6
>>> catalystDiffusion = 1e-9
>>> siteDensity = 9.8e-6
```

properties of the cupric ions,

```
>>> molarVolume = 7.1e-6
>>> charge = 2
>>> metalDiffusionCoefficient = 5.6e-10
```

parameters dependent on experimental constraints,

```
>>> temperature = 298.
>>> overpotential = -0.3
>>> bulkMetalConcentration = 250.
>>> catalystConcentration = 5e-3
>>> catalystCoverage = 0.
```

parameters obtained from experiments on flat copper electrodes,

```
>>> currentDensity0 = 0.26
>>> currentDensity1 = 45.
```

general simulation control parameters,

```
>>> cflNumber = 0.2
>>> numberOfCellsInNarrowBand = 10
>>> cellsBelowTrench = 10
>>> cellSize = 0.1e-7
```

parameters required for a trench geometry,

```
>>> trenchDepth = 0.5e-6
>>> aspectRatio = 2.
>>> trenchSpacing = 0.6e-6
>>> boundaryLayerDepth = 0.3e-6
```

The hydrodynamic boundary layer depth (`boundaryLayerDepth`) is intentionally small in this example to keep the mesh at a reasonable size.

Build the mesh:

```
>>> from fipy.tools.parser import parse
>>> numberOfElements = parse('--numberOfElements', action='store',
...     type='int', default=-1)
>>> numberOfSteps = parse('--numberOfSteps', action='store',
...     type='int', default=5)


>>> if numberOfElements != -1:
...     pos = trenchSpacing * cellsBelowTrench / 4 / numberOfElements
...     sqr = trenchSpacing * (trenchDepth + boundaryLayerDepth) \
...           / (2 * numberOfElements)
...     cellSize = pos + sqrt(pos**2 + sqr)
... else:
...     cellSize = 0.1e-7

>>> yCells = cellsBelowTrench \
...          + int((trenchDepth + boundaryLayerDepth) / cellSize)
>>> xCells = int(trenchSpacing / 2 / cellSize)


>>> from fipy import *
>>> mesh = Grid2D(dx=cellSize,
...               dy=cellSize,
...               nx=xCells,
...               ny=yCells)
```

A `distanceVariable` object, ϕ, is required to store the position of the interface . The `distanceVariable` calculates its value so that it is a distance function (*i.e.* holds the distance at any point in the mesh from the electrolyte/metal interface at $\phi = 0$) and $|\nabla\phi| = 1$.

First, create the ϕ variable, which is initially set to -1 everywhere. Create an initial variable,

```
>>> narrowBandWidth = numberOfCellsInNarrowBand * cellSize
>>> distanceVar = DistanceVariable(
...     name='distance variable',
...     mesh= mesh,
...     value=-1.,
...     narrowBandWidth=narrowBandWidth,
...     hasOld=1)
```

The electrolyte region will be the positive region of the domain while the metal region will be negative.

```
>>> bottomHeight = cellsBelowTrench * cellSize
>>> trenchHeight = bottomHeight + trenchDepth
>>> trenchWidth = trenchDepth / aspectRatio
>>> sideWidth = (trenchSpacing - trenchWidth) / 2
```

```
>>> x, y = mesh.getCellCenters()
>>> distanceVar.setValue(1., where=(y > trenchHeight)
...                                 | ((y > bottomHeight)
...                                    & (x < xCells * cellSize - sideWidth)))
```

```
>>> distanceVar.calcDistanceFunction(narrowBandWidth=1e10)
```

The `distanceVariable` has now been created to mark the interface. Some other variables need to be created that govern the concentrations of various species. Create the catalyst surfactant coverage, θ, variable. This variable influences the deposition rate.

..

¿¿¿ catalystVar = SurfactantVariable(... name="catalyst variable", ... value=catalystCoverage, ... distanceVar=distanceVar)Create the bulk catalyst concentration, c_θ, in the electrolyte,

```
>>> bulkCatalystVar = CellVariable(
...     name='bulk catalyst variable',
...     mesh=mesh,
...     value=catalystConcentration)
```

Create the bulk metal ion concentration, c_m, in the electrolyte.

```
>>> metalVar = CellVariable(
...     name='metal variable',
...     mesh=mesh,
...     value=bulkMetalConcentration)
```

The following commands build the `depositionRateVariable`, v. The `depositionRateVariable` is given by the following equation.

$$v = \frac{i\Omega}{nF}$$

where Ω is the metal molar volume, n is the metal ion charge and F is Faraday's constant. The current density is given by

$$i = i_0 \frac{c_m^i}{c_m^\infty} \exp\left(\frac{-\alpha F}{RT}\eta\right)$$

where c_m^i is the metal ion concentration in the bulk at the interface, c_m^∞ is the far-field bulk concentration of metal ions, α is the transfer coefficient, R is the gas constant, T is the temperature and η is the overpotential. The exchange current density is an empirical function of catalyst coverage,

$$i_0(\theta) = b_0 + b_1\theta$$

The commands needed to build this equation are,

```
>>> expoConstant = -transferCoefficient * faradaysConstant \
...                / (gasConstant * temperature)
>>> tmp = currentDensity1 \
...       * catalystVar.getInterfaceVar()
>>> exchangeCurrentDensity = currentDensity0 + tmp
>>> expo = exp(expoConstant * overpotential)
>>> currentDensity = expo * exchangeCurrentDensity * metalVar \
...                  / bulkMetalConcentration
>>> depositionRateVariable = currentDensity * molarVolume \
...                           / (charge * faradaysConstant)
```

Build the extension velocity variable v_{ext}. The extension velocity uses the `extensionEquation` to spread the velocity at the interface to the rest of the domain.

```
>>> extensionVelocityVariable = CellVariable(
...     name='extension velocity',
...     mesh=mesh,
...     value=depositionRateVariable)
```

Using the variables created above the governing equations will be built. The governing equation for surfactant conservation is given by,

$$\dot{\theta} = Jv\theta + kc_\theta^i(1-\theta)$$

where θ is the coverage of catalyst at the interface, J is the curvature of the interface, v is the normal velocity of the interface, c_θ^i is the concentration of catalyst in the bulk at the interface. The value k is given by an empirical function of overpotential,

$$k = k_0 + k_3\eta^3$$

The above equation is represented by the `AdsorbingSurfactantEquation` in FiPy:

```
>>> surfactantEquation = AdsorbingSurfactantEquation(
...     surfactantVar=catalystVar,
...     distanceVar=distanceVar,
...     bulkVar=bulkCatalystVar,
```

```
...     rateConstant=rateConstant0 \
...                     + rateConstant3 * overpotential**3)
```

The variable ϕ is advected by the `advectionEquation` given by,

$$\frac{\partial \phi}{\partial t} + v_{\text{ext}}|\nabla \phi| = 0$$

and is set up with the following commands:

```
>>> advectionEquation = buildHigherOrderAdvectionEquation(
...     advectionCoeff=extensionVelocityVariable)
```

The diffusion of metal ions from the far field to the interface is governed by,

$$\frac{\partial c_m}{\partial t} = \nabla \cdot D \nabla c_m$$

where,

$$D = \begin{cases} D_m & \text{when } \phi > 0, \\ 0 & \text{when } \phi \leq 0 \end{cases}$$

The following boundary condition applies at $\phi = 0$,

$$D\hat{n} \cdot \nabla c = \frac{v}{\Omega}.$$

The `MetalIonDiffusionEquation` is set up with the following commands.

```
>>> metalEquation = buildMetalIonDiffusionEquation(
...     ionVar=metalVar,
...     distanceVar=distanceVar,
...     depositionRate=depositionRateVariable,
...     diffusionCoeff=metalDiffusionCoefficient,
...     metalIonMolarVolume=molarVolume,
... )
```

```
>>> metalEquationBCs = FixedValue(faces=mesh.getFacesTop(), value=bulkMetalConcentration)
```

The `SurfactantBulkDiffusionEquation` solves the bulk diffusion of a species with a source term for the jump from the bulk to an interface. The governing equation is given by,

$$\frac{\partial c}{\partial t} = \nabla \cdot D \nabla c$$

where,

$$D = \begin{cases} D_\theta & \text{when } \phi > 0 \\ 0 & \text{when } \phi \leq 0 \end{cases}$$

The jump condition at the interface is defined by Langmuir adsorption. Langmuir adsorption essentially states that the ability for a species to jump from an electrolyte to an interface is proportional to the concentration in the electrolyte, available site density and a jump coefficient. The boundary condition at $\phi = 0$ is given by,

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

The `SurfactantBulkDiffusionEquation` is set up with the following commands.

```
>>> bulkCatalystEquation = buildSurfactantBulkDiffusionEquation(
...     bulkVar=bulkCatalystVar,
...     distanceVar=distanceVar,
...     surfactantVar=catalystVar,
...     diffusionCoeff=catalystDiffusion,
...     rateConstant=rateConstant0 * siteDensity
... )

>>> catalystBCs = FixedValue(faces=mesh.getFacesTop(), value=catalystConcentration)
```

If running interactively, create viewers.

```
>>> if __name__ == '__main__':
...     try:
...         viewer = MayaviSurfactantViewer(distanceVar,
...                                         catalystVar.getInterfaceVar(),
...                                         zoomFactor=1e6,
...                                         datamax=1.0,
...                                         datamin=0.0,
...                                         smooth=1)
...     except:
...         viewer = MultiViewer(viewers=(
...             Viewer(distanceVar, datamin=-1e-9, datamax=1e-9),
...             Viewer(catalystVar.getInterfaceVar())))
... else:
...     viewer = None
```

The `levelSetUpdateFrequency` defines how often to call the `distanceEquation` to reinitialize the `distanceVariable` to a distance function.

```
>>> levelSetUpdateFrequency = int(0.8 * narrowBandWidth \
...                               / (cellSize * cflNumber * 2))
```

The following loop runs for `numberOfSteps` time steps. The time step is calculated with the CFL number and the maximum extension velocity. v to v_{ext} throughout the whole domain using $\nabla\phi \cdot \nabla v_{\text{ext}} = 0$.

```
>>> for step in range(numberOfSteps):
...
...     if viewer is not None:
...         viewer.plot()
```

```
...
...     if step % levelSetUpdateFrequency == 0:
...         distanceVar.calcDistanceFunction()
...
...     extensionVelocityVariable.setValue(depositionRateVariable())
...
...     distanceVar.updateOld()
...     catalystVar.updateOld()
...     metalVar.updateOld()
...     bulkCatalystVar.updateOld()
...     distanceVar.extendVariable(extensionVelocityVariable)
...     dt = cflNumber * cellSize / extensionVelocityVariable.max()
...     advectionEquation.solve(distanceVar, dt=dt)
...     surfactantEquation.solve(catalystVar, dt=dt)
...     metalEquation.solve(var=metalVar, dt=dt,
...                         boundaryConditions=metalEquationBCs)
...     bulkCatalystEquation.solve(var=bulkCatalystVar, dt=dt,
...                                boundaryConditions=catalystBCs)
```

The following is a short test case. It uses saved data from a simulation with 5 time steps. It is not a test for accuracy but a way to tell if something has changed or been broken.

```
>>> import os
>>> filepath = os.path.join(os.path.split(__file__)[0],
...                         "simpleTrenchSystem.gz")
>>> print catalystVar.allclose(loadtxt(filepath), rtol=1e-4)
1

>>> if __name__ == '__main__':
...     raw_input('finished')
```

Cahn-Hilliard Examples

10.1 Module examples.cahnHilliard.mesh2D

The spinodal decomposition phenomenon is a spontaneous separation of an initially homogenous mixture into two distinct regions of different properties (spin-up/spin-down, component A/component B). It is a "barrierless" phase separation process, such that under the right thermodynamic conditions, any fluctuation, no matter how small, will tend to grow. This is in contrast to nucleation, where a fluctuation must exceed some critical magnitude before it will survive and grow. Spinodal decomposition can be described by the "Cahn-Hilliard" equation (also known as "conserved Ginsberg-Landau" or "model B" of Hohenberg & Halperin)

$$\frac{\partial \phi}{\partial t} = \nabla \cdot D \nabla \left(\frac{\partial f}{\partial \phi} - \epsilon^2 \nabla^2 \phi \right).$$

where ϕ is a conserved order parameter, possibly representing alloy composition or spin. The double-well free energy function $f = (a^2/2)\phi^2(1-\phi)^2$ penalizes states with intermediate values of ϕ between 0 and 1. The gradient energy term $\epsilon^2\nabla^2\phi$, on the other hand, penalizes sharp changes of ϕ. These two competing effects result in the segregation of ϕ into domains of 0 and 1, separated by abrupt, but smooth, transitions. The parameters a and ϵ determine the relative weighting of the two effects and D is a rate constant. We can simulate this process in FiPy with a simple script:

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

(Note that all of the functionality of NumPy is imported along with FiPy, although much is augmented for FiPy's needs.)

```
>>> if __name__ == "__main__":
...     nx = ny = 1000
... else:
...     nx = ny = 10
>>> mesh = Grid2D(nx=nx, ny=ny, dx=0.25, dy=0.25)
>>> phi = CellVariable(name=r"$\phi$", mesh=mesh)
```

We start the problem with random fluctuations about \$phi = 1/2\$

```
>>> phi.setValue(GaussianNoiseVariable(mesh=mesh,
...                                    mean=0.5,
...                                    variance=0.01))
```

FiPy doesn't plot or output anything unless you tell it to:

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

For FiPy, we need to perform the partial derivative $\partial f/\partial\phi$ manually and then put the equation in the canonical form by decomposing the spatial derivatives so that each `Term` is of a single, even order:

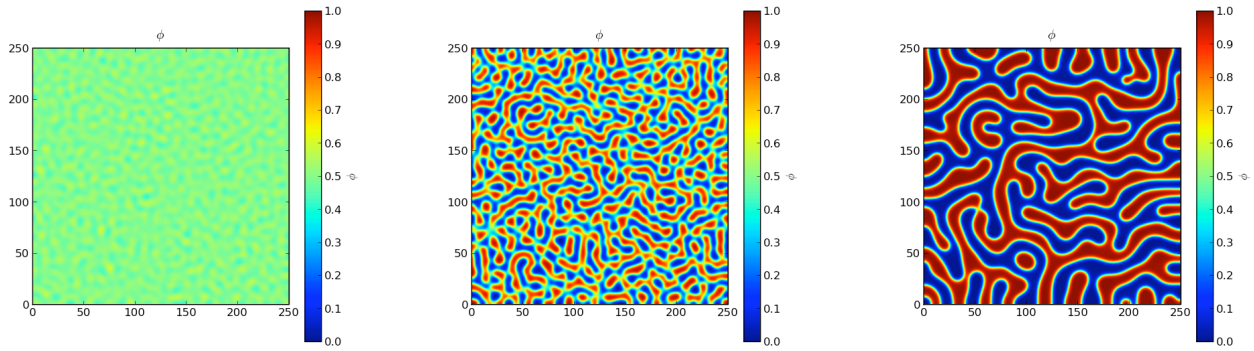
$$\frac{\partial \phi}{\partial t} = \nabla \cdot Da^2 \left[1 - 6\phi\left(1 - \phi\right)\right] \nabla\phi - \nabla \cdot D\nabla\epsilon^2\nabla^2\phi.$$

FiPy would automatically interpolate `D * a**2 * (1 - 6 * phi * (1 - phi))` onto the `Face`s, where the diffusive flux is calculated, but we obtain somewhat more accurate results by performing a linear interpolation from `phi` at `Cell` centers to `PHI` at `Face` centers. Some problems benefit from non-linear interpolations, such as harmonic or geometric means, and FiPy makes it easy to obtain these, too.

```
>>> PHI = phi.getArithmeticFaceValue()
>>> D = a = epsilon = 1.
>>> eq = (TransientTerm()
...       == DiffusionTerm(coeff=D * a**2 * (1 - 6 * PHI * (1 - PHI)))
...       - DiffusionTerm(coeff=(D, epsilon**2)))
```

Because the evolution of a spinodal microstructure slows with time, we use exponentially increasing time steps to keep the simulation "interesting". The FiPy user always has direct control over the evolution of their problem.

```
>>> dexp = -5
>>> elapsed = 0.
>>> if __name__ == "__main__":
...     duration = 1000.
... else:
...     duration = 1e-2
>>> while elapsed < duration:
...     dt = min(100, exp(dexp))
...     elapsed += dt
...     dexp += 0.01
...     eq.solve(phi, dt=dt)
...     if __name__ == "__main__":
...         viewer.plot()
```



(a) $t = 30$ (b) $t = 100$ (c) $t = 1000$

10.2 Module examples.cahnHilliard.sphere

Solves the Cahn-Hilliard problem on the surface of a sphere, such as may occur on vesicles (http://www.youtube.com/watch?v=kDsFP67_ZSE).

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

The only difference from `examples.cahnHilliard.mesh2D` is the declaration of `mesh`.

```
>>> mesh = GmshImporter2DIn3DSpace('''
...     radius = 5.0;
...     cellSize = 0.3;
...
...     // create inner 1/8 shell
...     Point(1) = {0, 0, 0, cellSize};
...     Point(2) = {-radius, 0, 0, cellSize};
...     Point(3) = {0, radius, 0, cellSize};
...     Point(4) = {0, 0, radius, cellSize};
...     Circle(1) = {2, 1, 3};
...     Circle(2) = {4, 1, 2};
...     Circle(3) = {4, 1, 3};
...     Line Loop(1) = {1, -3, 2} ;
...     Ruled Surface(1) = {1};
...
...     // create remaining 7/8 inner shells
...     t1[] = Rotate {{0,0,1},{0,0,0},Pi/2} {Duplicata{Surface{1};}};
...     t2[] = Rotate {{0,0,1},{0,0,0},Pi} {Duplicata{Surface{1};}};
...     t3[] = Rotate {{0,0,1},{0,0,0},Pi*3/2} {Duplicata{Surface{1};}};
...     t4[] = Rotate {{0,1,0},{0,0,0},-Pi/2} {Duplicata{Surface{1};}};
...     t5[] = Rotate {{0,0,1},{0,0,0},Pi/2} {Duplicata{Surface{t4[0]};}};
...     t6[] = Rotate {{0,0,1},{0,0,0},Pi} {Duplicata{Surface{t4[0]};}};
```

```
...     t7[] = Rotate {{0,0,1},{0,0,0},Pi*3/2} {Duplicata{Surface{t4[0]};}};
...
...     // create entire inner and outer shell
...     Surface Loop(100)={1,t1[0],t2[0],t3[0],t7[0],t4[0],t5[0],t6[0]};
... ''').extrude(extrudeFunc=lambda r: 1.1 * r)
>>> phi = CellVariable(name=r"$\phi$", mesh=mesh)
```

We start the problem with random fluctuations about $\phi = 1/2$

¿¿¿ phi.setValue(GaussianNoiseVariable(mesh=mesh, ... mean=0.5, ... variance=0.01)) FiPy doesn't plot or output anything unless you tell it to:

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

For FiPy, we need to perform the partial derivative $\partial f/\partial\phi$ manually and then put the equation in the canonical form by decomposing the spatial derivatives so that each `Term` is of a single, even order:

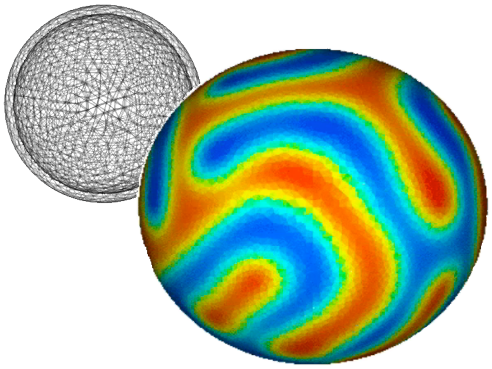
$$\frac{\partial \phi}{\partial t} = \nabla \cdot Da^2 \left[1 - 6\phi\left(1 - \phi\right)\right] \nabla\phi - \nabla \cdot D\nabla\epsilon^2\nabla^2\phi.$$

FiPy would automatically interpolate `D * a**2 * (1 - 6 * phi * (1 - phi))` onto the `Face`s, where the diffusive flux is calculated, but we obtain somewhat more accurate results by performing a linear interpolation from `phi` at `Cell` centers to `PHI` at `Face` centers. Some problems benefit from non-linear interpolations, such as harmonic or geometric means, and FiPy makes it easy to obtain these, too.

```
>>> PHI = phi.getArithmeticFaceValue()
>>> D = a = epsilon = 1.
>>> eq = (TransientTerm()
...       == DiffusionTerm(coeff=D * a**2 * (1 - 6 * PHI * (1 - PHI)))
...       - DiffusionTerm(coeff=(D, epsilon**2)))
```

Because the evolution of a spinodal microstructure slows with time, we use exponentially increasing time steps to keep the simulation "interesting". The FiPy user always has direct control over the evolution of their problem.

```
>>> dexp = -5
>>> elapsed = 0.
>>> if __name__ == "__main__":
...     duration = 1000.
... else:
...     duration = 1e-2
>>> while elapsed < duration:
...     dt = min(100, exp(dexp))
...     elapsed += dt
...     dexp += 0.01
...     eq.solve(phi, dt=dt)
...     if __name__ == "__main__":
...         viewer.plot()
```



Fluid Flow Examples

11.1 Module examples.flow.stokesCavity

This example is an implementation of a rudimentary Stokes solver. It solves the Navier-Stokes equation in the viscous limit,

$$\nabla\mu \cdot \nabla\vec{u} = \nabla p$$

and the continuity equation,

$$\nabla \cdot \vec{u} = 0$$

where $\vec{u}$ is the fluid velocity, p is the pressure and μ is the viscosity. The domain in this example is a square cavity of unit dimensions with a moving lid of unit speed. This example uses the SIMPLE algorithm with Rhie-Chow interpolation to solve the pressure-momentum coupling. Some of the details of the algorithm will be highlighted below but a good reference for this material is Ferziger and Perić [35]. The solution has a high degree of error close to the corners of the domain for the pressure but does a reasonable job of predicting the velocities away from the boundaries. A number of aspects of FiPy need to be improved to have a first class flow solver. These include, higher order spatial diffusion terms, proper wall boundary conditions, improved mass flux evaluation and extrapolation of cell values to the boundaries using gradients. In the table below a comparison is made with the Dolfyn open source code on a 100 by 100 grid. The table shows the frequency of values that fall within the given error confidence bands. Dolfyn has the added features described above. When these features are switched off the results of Dolfyn and FiPy are identical.

% frequency of cells	x-velocity error (%)	y-velocity error (%)	pressure error (%)
90	< 0.1	< 0.1	< 5
5	0.1 to 0.6	0.1 to 0.3	5 to 11
4	0.6 to 7	0.3 to 4	11 to 35
1	7 to 96	4 to 80	35 to 179
0	> 96	> 80	> 179

To start, some parameters are declared.

```
>>> from fipy import *

>>> L = 1.0
>>> N = 50
>>> dL = L / N
>>> viscosity = 1.
>>> pressureRelaxation = 0.2
>>> velocityRelaxation = 0.5
>>> if __name__ == '__main__':
```

```
... sweeps = 300
... else:
...     sweeps = 5
```

Build the mesh.

```
>>> mesh = Grid2D(nx=N, ny=N, dx=dL, dy=dL)
```

Declare the variables.

```
>>> pressure = CellVariable(mesh=mesh, name='pressure')
>>> pressureCorrection = CellVariable(mesh=mesh)
>>> xVelocity = CellVariable(mesh=mesh, name='X velocity')
>>> yVelocity = CellVariable(mesh=mesh, name='Y velocity')
```

The velocity is required as a rank-1 `FaceVariable` for calculating the mass flux. This is a somewhat clumsy aspect of the FiPy interface that needs improvement.

```
>>> velocity = FaceVariable(mesh=mesh, rank=1)
```

Build the Stokes equations.

```
>>> xVelocityEq = ImplicitDiffusionTerm(coeff=viscosity) - pressure.getGrad().dot([1,0])
>>> yVelocityEq = ImplicitDiffusionTerm(coeff=viscosity) - pressure.getGrad().dot([0,1])
```

In this example the SIMPLE algorithm is used to couple the pressure and momentum equations. Let us assume we have solved the discretized momentum equations using a guessed pressure field p^* to obtain a velocity field $\vec{u}^*$. We would like to somehow correct these initial fields to satisfy both the discretized momentum and continuity equations. We now try to correct these initial fields with a correction such that $\vec{u} = \vec{u}^* + \vec{u}'$ and $p = p^* + p'$, where $\vec{u}$ and p now satisfy the momentum and continuity equations. Substituting the exact solution into the equations we obtain,

$$\nabla\mu \cdot \nabla\vec{u}' = \vec{p}'$$

and

$$\nabla \cdot \vec{u}^* + \nabla \cdot \vec{u}' = 0$$

We now use the discretized form of the equations to write the velocity correction in terms of the pressure correction. The discretized form of the above equation is,

$$a_P \vec{u}'_P = \sum_f a_A \vec{u}'_A - V_P(\nabla p')_P$$

where notation from section 3.4 is used. The SIMPLE algorithm drops the second term in the above equation to leave,

$$\vec{u}'_P = -\frac{V_P(\nabla p')_P}{a_P}$$

By substituting the above expression into the continuity equations we obtain the pressure correction equation,

$$\nabla \frac{V_P}{a_P} \cdot \nabla p' = \nabla \cdot \vec{u}^*$$

In the discretized version of the above equation V_P/a_P is approximated at the face by $A_f d_{AP}/(a_P)_f$. In FiPy the pressure correction equation can be written as,

```
>>> ap = CellVariable(mesh=mesh)
>>> coeff = mesh._getFaceAreas() * mesh._getCellDistances() / ap.getArithmeticFaceValue()
>>> pressureCorrectionEq = ImplicitDiffusionTerm(coeff=coeff) - velocity.getDivergence()
```

Set up the no-slip boundary conditions

```
>>> bcs = (FixedValue(faces=mesh.getFacesLeft(), value=0),
...        FixedValue(faces=mesh.getFacesRight(), value=0),
...        FixedValue(faces=mesh.getFacesBottom(), value=0),)
>>> bcsX = bcs + (FixedValue(faces=mesh.getFacesTop(), value=1),)
>>> bcsY = bcs + (FixedValue(faces=mesh.getFacesTop(), value=0),)
```

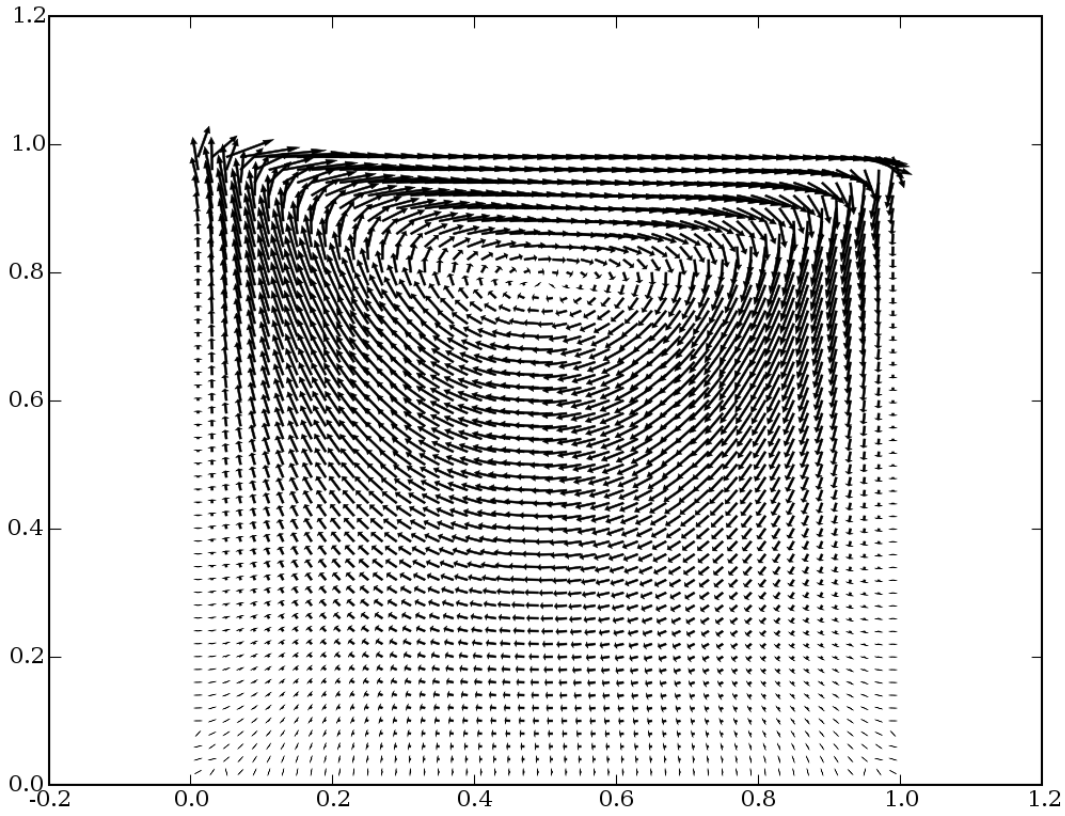
Set up the viewers,

```
>>> if __name__ == '__main__':
...     viewer = Viewer(vars=(pressure, xVelocity, yVelocity, velocity))
```

Below, we iterate for a set number of sweeps. We use the `sweep()` method instead of `solve()` because we require the residual for output. We also use the `cacheMatrix()`, `getMatrix()`, `cacheRHSvector()` and `getRHSvector()` because both the matrix and RHS vector are required by the SIMPLE algorithm. Additionally, the `sweep()` method is passed an `underRelaxation` factor to relax the solution. This argument cannot be passed to `solve()`.

```
>>> for sweep in range(sweeps):
...
...     ## solve the Stokes equations to get starred values
...     xVelocityEq.cacheMatrix()
...     xres = xVelocityEq.sweep(var=xVelocity,
...                              boundaryConditions=bcsX,
...                              underRelaxation=velocityRelaxation)
...     xmat = xVelocityEq.getMatrix()
...
...     yres = yVelocityEq.sweep(var=yVelocity,
...                              boundaryConditions=bcsY,
...                              underRelaxation=velocityRelaxation)
...
...     ## update the ap coefficient from the matrix diagonal
...     ap[:] = -xmat.takeDiagonal()
...
...     ## update the face velocities based on starred values
```

```
...     velocity[0] = xVelocity.getArithmeticFaceValue()
...     velocity[1] = yVelocity.getArithmeticFaceValue()
...     velocity[..., mesh.getExteriorFaces().getValue()] = 0.
...
...     ## solve the pressure correction equation
...     pressureCorrectionEq.cacheRHSvector()
...     pres = pressureCorrectionEq.sweep(var=pressureCorrection)
...     rhs = pressureCorrectionEq.getRHSvector()
...
...     ## update the pressure using the corrected value but hold one cell fixed
...     pressure.setValue(pressure + pressureRelaxation * \
...                                             (pressureCorrection - pressureCorrection[0]))
...     ## update the velocity using the corrected pressure
...     xVelocity.setValue(xVelocity - pressureCorrection.getGrad()[0] / \
...                                                 ap * mesh.getCellVolumes())
...     yVelocity.setValue(yVelocity - pressureCorrection.getGrad()[1] / \
...                                                 ap * mesh.getCellVolumes())
...
...     if __name__ == '__main__':
...         if sweep%1 == 0:
...             print 'sweep:',sweep,', x residual:',xres, \
...                                   ', y residual',yres, \
...                                   ', p residual:',pres, \
...                                   ', continuity:',max(abs(rhs))
...
...             viewer.plot()
```



Test values in the last cell.

```
>>> print pressure[...,-1].allclose(145.233883763)
1
>>> print xVelocity[...,-1].allclose(0.24964673696)
1
>>> print yVelocity[...,-1].allclose(-0.164498041783)
1
```

Converting from older versions of FiPy

12.1 Module examples.update1_0to2_0

FiPy 2.0 introduces several syntax changes from FiPy 1.0. We appreciate that this is very inconvenient for our users, but we hope you'll agree that the new syntax is easier to read and easier to use. We assure you that this is not something we do casually; it has been over three years since our last incompatible change (when FiPy 1.0 superceded FiPy 0.1).

All examples included with version 2.0 have been updated to use the new syntax, but any scripts you have written for FiPy 1.0 will need to be updated. A complete listing of the changes needed to take the FiPy examples scripts from version 1.0 to version 2.0 can be found at

http://www.matforge.org/fipy/wiki/upgrade1_0examplesTo2_0

but we summarize the necessary changes here. If these tips are not sufficient to make your scripts compatible with FiPy 2.0, please don't hesitate to ask for help on the mailing list.

The following items **must** be changed in your scripts

- The dimension axis of a `Variable` is now first, not last

  ```
  >>> x = mesh.getCellCenters()[0]
  ```

  instead of

  ```
  >>> x = mesh.getCellCenters()[...,0]
  ```

  This seemingly arbitrary change simplifies a great many things in FiPy, but the one most noticeable to the user is that you can now write

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

  instead of

  ```
  >>> x = mesh.getCellCenters()[...,0]
  >>> y = mesh.getCellCenters()[...,1]
  ```

  Unfortunately, we cannot reliably automate this conversion, but we find that searching for "`...,`" and "`:,`" finds almost everything. Please don't blindly "search & replace all" as that is almost bound to create more problems than it's worth.

Note

Any vector constants must be reoriented. For instance, in order to offset a `Mesh`, you must write

```
>>> mesh = Grid2D(...) + ((deltax,), (deltay,))
```

or

```
>>> mesh = Grid2D(...) + [[deltax], [deltay]]
```

instead of

```
>>> mesh = Grid2D(...) + (deltax, deltay)
```

- `VectorCellVariable` and `VectorFaceVariable` no longer exist. `CellVariable` and `FaceVariable` now both inherit from `MeshVariable`, and a `MeshVariable` can have arbitrary rank. A field of scalars (default) will have `rank=0`, a field of vectors will have `rank=1`, etc. You should write

  ```
  >>> vectorField = CellVariable(mesh=mesh, rank=1)
  ```

  instead of

  ```
  >>> vectorField = VectorCellVariable(mesh=mesh)
  ```

  Note

  Because vector fields are properly supported, use vector operations to manipulate them, such as

  ```
  >>> phase.getFaceGrad().dot((( 0, 1),
  ...                          (-1, 0)))
  ```

  instead of the hackish

  ```
  >>> phase.getFaceGrad()._take((1, 0), axis=1) * (-1, 1)
  ```

- For internal reasons, FiPy now supports `CellVariable` and `FaceVariable` objects that contain integers, but it is not meaningful to solve a PDE for an integer field (FiPy should issue a warning if you try). As a result, when given, initial values must be specified as floating-point values:

  ```
  >>> var = CellVariable(mesh=mesh, value=1.)
  ```

  where they used to be quietly accepted as integers

  ```
  >>> var = CellVariable(mesh=mesh, value=1)
  ```

  If the `value` argument is not supplied, the `CellVariable` will contain floats, as before.

- The `faces` argument to `BoundaryCondition` now takes a mask, instead of a list of `Face` IDs. Now you write

  ```
  >>> X, Y = mesh.getFaceCenters()
  >>> FixedValue(faces=mesh.getExteriorFaces() & (X**2 < 1e-6), value=...)
  ```

  instead of

  ```
  >>> exteriorFaces = mesh.getExteriorFaces()
  >>> X = exteriorFaces.getCenters()[...,0]
  >>> FixedValue(faces=exteriorFaces.where(X**2 < 1e-6), value=...)
  ```

With the old syntax, a different call to `getCenters()` had to be made for each set of `Face` objects. It was also extremely difficult to specify boundary conditions that depended both on position in space and on the current values of other `Variables`

```
>>> FixedValue(faces=(mesh.getExteriorFaces()
...                   & (((X**2 < 1e-6)
...                       & (Y > 3.))
...                      | (phi.getArithmeticFaceValue()
...                         < sin(gamma.getArithmeticFaceValue())))), value=...)
```

although it probably could have been done with a rather convoluted (and slow!) `filter` function passed to `where`. There no longer are any `filter` methods used in FiPy. You now would write

```
>>> x, y = mesh.getCellCenters()
>>> initialArray[(x < dx) | (x > (Lx - dx)) | (y < dy) | (y > (Ly - dy))] = 1.
```

instead of the **much** slower

```
>>> def cellFilter(cell):
...     return ((cell.getCenter()[0] < dx)
...             or (cell.getCenter()[0] > (Lx - dx))
...             or (cell.getCenter()[1] < dy)
...             or (cell.getCenter()[1] > (Ly - dy)))
>>> positiveCells = mesh.getCells(filter=cellFilter)
>>> for cell in positiveCells:
...     initialArray[cell.getID()] = 1.
```

Although they still exist, we find very lille cause to ever call `mesh.getCells()` or `mesh.getFaces()`.

- Some modules, such as `fipy.solvers`, have been significantly rearranged. For example, you need to change

  ```
  >>> from fipy.solvers.linearPCGSolver import LinearPCGSolver
  ```

  to either

  ```
  >>> from fipy import LinearPCGSolver
  ```

  or

  ```
  >>> from fipy.solvers.pysparse.linearPCGSolver import LinearPCGSolver
  ```

- The `numerix.max()` and `numerix.min()` functions no longer exist. Either call `max()` and `min()` or `var.max()` and `var.min()`.
- The `Numeric` module has not been supported for a long time. Be sure to use

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

  instead of

  ```
  >>> import Numeric
  ```

The remaining changes are not **required**, but they make scripts easier to read and we recommend them. FiPy may issue a `DeprecationWarning` for some cases, to indicate that we may not maintain the old syntax indefinitely.

- All of the most commonly used classes and functions in FiPy are directly accessible in the `fipy` namespace. For brevity, our examples now start with

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

instead of the explicit

```
>>> from fipy.meshes.grid1D import Grid1D
>>> from fipy.terms.powerLawConvectionTerm import PowerLawConvectionTerm
>>> from fipy.variables.cellVariable import CellVariable
```

imports that we used to use. Most of the explicit imports should continue to work, so you do not need to change them if you don't wish to, but we find our own scripts much easier to read without them.

All of the `numerix` module is now imported into the `fipy` namespace, so you can call `numerix` functions a number of different ways, including:

```
>>> from fipy import *
>>> y = exp(x)
```

or

```
>>> from fipy import numerix
>>> y = numerix.exp(x)
```

or

```
>>> from fipy.tools.numerix import exp
>>> y = exp(x)
```

We generally use the first, but you may see us use the others, and should feel free to use whichever form you find most comfortable.

Note

Internally, FiPy uses explicit imports, as is considered best Python practice, but we feel that clarity trumps orthodoxy when it comes to the examples.

- The function `fipy.viewers.make()` has been renamed to `fipy.viewers.Viewer()`. All of the `limits` can now be supplied as direct arguments, as well (although this is not required). The result is a more natural syntax:

```
>>> from fipy import Viewer
>>> viewer = Viewer(vars=(alpha, beta, gamma), datamin=0, datamax=1)
```

instead of

```
>>> from fipy import viewers
>>> viewer = viewers.make(vars=(alpha, beta, gamma),
...                       limits={'datamin': 0, 'datamax': 1})
```

With the old syntax, there was also a temptation to write

```
>>> from fipy.viewers import make
>>> viewer = make(vars=(alpha, beta, gamma))
```

which can be very hard to understand after the fact (`make`? `make` what?).

- A `ConvectionTerm` can now calculate its Peclet number automatically, so the `diffusionTerm` argument is no longer required

```
>>> eq = (TransientTerm()
...       == ImplicitDiffusionTerm(coeff=diffCoeff)
...       + PowerLawConvectionTerm(coeff=convCoeff))
```

instead of

```
>>> diffTerm = ImplicitDiffusionTerm(coeff=diffCoeff)
>>> eq = (TransientTerm()
...       == diffTerm
...       + PowerLawConvectionTerm(coeff=convCoeff, diffusionTerm=diffTerm))
```

- An `ImplicitSourceTerm` now "knows" how to partion itself onto the solution matrix, so you can write

  ```
  >>> S0 = mXi * phase * (1 - phase) - phase * S1
  >>> source = S0 + ImplicitSourceTerm(coeff=S1)
  ```

  instead of

  ```
  >>> S0 = mXi * phase * (1 - phase) - phase * S1 * (S1 < 0)
  >>> source = S0 + ImplicitSourceTerm(coeff=S1 * (S1 < 0))
  ```

  It is definitely still advantageous to hand-linearize your source terms, but it is no longer necessary to worry about putting the "wrong" sign on the diagonal of the matrix.

- To make clearer the distinction between iterations, timesteps, and sweeps (see FAQ 5.3) the `steps` argument to a `Solver` object has been renamed `iterations`.

- `ImplicitDiffusionTerm` has been renamed to `DiffusionTerm`.

12.2 Module examples.update0_1to1_0

It seems unlikely that many users are still running FiPy 0.1, but for those that are, the syntax of FiPy scripts changed considerably between version 0.1 and version 1.0. We incremented the full version-number to stress that previous scripts are incompatible. We strongly believe that these changes are for the better, resulting in easier code to write and read as well as slightly improved efficiency, but we realize that this represents an inconvenience to our users that have already written scripts of their own. We will strive to avoid any such incompatible changes in the future.

Any scripts you have written for FiPy 0.1 should be updated in two steps, first to work with FiPy 1.0, and then with FiPy 2.0. As a tutorial for updating your scripts, we will walk through updating the file `examples/convection/exponential1D/input.py` from FiPy 0.1. If you attempt to run that script with FiPy 1.0, the script will fail and you will see the errors shown below:

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

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

with coefficients $D = 1$ and $\vec{u} = (10, 0)$, or

```
>>> diffCoeff = 1.
>>> convCoeff = (10.,0.)
```

We define a 1D mesh

```
>>> L = 10.
>>> nx = 1000
```

```
>>> ny = 1
>>> from fipy.meshes.grid2D import Grid2D
>>> mesh = Grid2D(L / nx, L / ny, nx, ny)
```

and impose the boundary conditions

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

or

```
>>> valueLeft = 0.
>>> valueRight = 1.
>>> from fipy.boundaryConditions.fixedValue import FixedValue
>>> from fipy.boundaryConditions.fixedFlux import FixedFlux
>>> boundaryConditions = (
...     FixedValue(mesh.getFacesLeft(), valueLeft),
...     FixedValue(mesh.getFacesRight(), valueRight),
...     FixedFlux(mesh.getFacesTop(), 0.),
...     FixedFlux(mesh.getFacesBottom(), 0.)
...     )
```

The solution variable is initialized to `valueLeft`:

```
>>> from fipy.variables.cellVariable import CellVariable
>>> var = CellVariable(
...     name = "concentration",
...     mesh = mesh,
...     value = valueLeft)
```

The `SteadyConvectionDiffusionScEquation` object is used to create the equation. It needs to be passed a convection term instantiator as follows:

```
>>> from fipy.terms.exponentialConvectionTerm import ExponentialConvectionTerm
>>> from fipy.solvers import *
>>> from fipy.equations.stdyConvDiffScEquation import SteadyConvectionDiffusionScEquation
Traceback (most recent call last):
...
ImportError: No module named equations.stdyConvDiffScEquation
>>> eq = SteadyConvectionDiffusionScEquation(
...      var = var,
...      diffusionCoeff = diffCoeff,
...      convectionCoeff = convCoeff,
...      solver = LinearLUSolver(tolerance = 1.e-15, steps = 2000),
...      convectionScheme = ExponentialConvectionTerm,
...      boundaryConditions = boundaryConditions
...      )
Traceback (most recent call last):
...
NameError: name 'SteadyConvectionDiffusionScEquation' is not defined
```

More details of the benefits and drawbacks of each type of convection term can be found in the numerical section of the manual. Essentially the `ExponentialConvectionTerm` and `PowerLawConvectionTerm` will both handle most types of convection diffusion cases with the `PowerLawConvectionTerm` being more efficient.

We iterate to equilibrium

```
>>> from fipy.iterators.iterator import Iterator
>>> it = Iterator((eq,))
Traceback (most recent call last):
...
NameError: name 'eq' is not defined
>>> it.timestep()
Traceback (most recent call last):
...
NameError: name 'it' is not defined
```

and test the solution against the analytical result

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

or

```
>>> axis = 0
>>> x = mesh.getCellCenters()[:,axis]
>>> from fipy.tools import numerix
>>> CC = 1. - numerix.exp(-convCoeff[axis] * x / diffCoeff)
>>> DD = 1. - numerix.exp(-convCoeff[axis] * L / diffCoeff)
>>> analyticalArray = CC / DD
>>> numerix.allclose(analyticalArray, var, rtol = 1e-10, atol = 1e-10)
0
```

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

```
>>> if __name__ == '__main__':
...     from fipy.viewers.grid2DGistViewer import Grid2DGistViewer
Traceback (most recent call last):
...
ImportError: No module named grid2DGistViewer


...     viewer = Grid2DGistViewer(var)
...     viewer.plot()
```

We see that a number of errors are thrown:

- `ImportError: No module named equations.stdyConvDiffScEquation`
- `NameError: name 'SteadyConvectionDiffusionScEquation' is not defined`

- `NameError: name 'eq' is not defined`
- `NameError: name 'it' is not defined`
- `ImportError: No module named grid2DGistViewer`

As is usually the case with computer programming, many of these errors are caused by earlier errors. Let us update the script, section by section:

Although no error was generated by the use of `Grid2D`, FiPy 1.0 supports a true 1D mesh class, so we instantiate the mesh as

```
>>> L = 10.
>>> nx = 1000
>>> from fipy.meshes.grid1D import Grid1D
>>> mesh = Grid1D(dx = L / nx, nx = nx)
```

The `Grid2D` class with `ny = 1` still works perfectly well for 1D problems, but the `Grid1D` class is slightly more efficient, and it makes the code clearer when a 1D geometry is actually desired.

Because the mesh is now 1D, we must update the convection coefficient vector to be 1D as well

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

The `FixedValue` boundary conditions at the left and right are unchanged, but a `Grid1D` mesh does not even have top and bottom faces:

```
>>> valueLeft = 0.
>>> valueRight = 1.
>>> from fipy.boundaryConditions.fixedValue import FixedValue
>>> boundaryConditions = (
...     FixedValue(mesh.getFacesLeft(), valueLeft),
...     FixedValue(mesh.getFacesRight(), valueRight))
```

The creation of the solution variable is unchanged:

```
>>> from fipy.variables.cellVariable import CellVariable
>>> var = CellVariable(name = "concentration",
...                    mesh = mesh,
...                    value = valueLeft)
```

The biggest change between FiPy 0.1 and FiPy 1.0 is that `Equation` objects no longer exist at all. Instead, `Term` objects can be simply added, subtracted, and equated to assemble an equation. Where before the assembly of the equation occurred in the black-box of `SteadyConvectionDiffusionScEquation`, we now assemble it directly:

```
>>> from fipy.terms.implicitDiffusionTerm import ImplicitDiffusionTerm
>>> diffTerm = ImplicitDiffusionTerm(coeff = diffCoeff)
```

```
>>> from fipy.terms.exponentialConvectionTerm import ExponentialConvectionTerm
>>> eq = diffTerm + ExponentialConvectionTerm(coeff = convCoeff,
...                                           diffusionTerm = diffTerm)
```

One thing that `SteadyConvectionDiffusionScEquation` took care of automatically was that a `ConvectionTerm` must know about any `DiffusionTerm` in the equation in order to calculate a Peclet number. Now, the `DiffusionTerm` must be explicitly passed to the `ConvectionTerm` in the `diffusionTerm` parameter.

The `Iterator` class still exists, but it is no longer necessary. Instead, the solution to an implicit steady-state problem like this can simply be obtained by telling the equation to solve itself (with an appropriate `solver` if desired, although the default `LinearPCGSolver` is usually suitable):

```
>>> from fipy.solvers import *
>>> eq.solve(var = var,
...          solver = LinearLUSolver(tolerance = 1.e-15, steps = 2000),
...          boundaryConditions = boundaryConditions)
```

Note

In version 0.1, the `Equation` object had to be told about the `Variable`, `Solver`, and `BoundaryCondition` objects when it was created (and it, in turn, passed much of this information to the `Term` objects in order to create them). In version 1.0, the `Term` objects (and the equation assembled from them) are abstract. The `Variable`, `Solver`, and `BoundaryCondition` objects are only needed by the `solve()` method (and, in fact, the same equation could be used to solve different variables, with different solvers, subject to different boundary conditions, if desired).

The analytical solution is unchanged, and we can test as before

```
>>> numerix.allclose(analyticalArray, var, rtol = 1e-10, atol = 1e-10)
1
```

or we can use the slightly simpler syntax

```
>>> print var.allclose(analyticalArray, rtol = 1e-10, atol = 1e-10)
1
```

The `ImportError: No module named grid2DGistViewer` results because the `Viewer` classes have been moved and renamed. This error could be resolved by changing the `import` statement appropriately:

```
>>> if __name__ == '__main__':
...     from fipy.viewers.gistViewer.gist1DViewer import Gist1DViewer
...     viewer = Gist1DViewer(vars = var)
...     viewer.plot()
```

Instead, rather than instantiating a particular `Viewer` (which you can still do, if you desire), a generic "factory" method will return a `Viewer` appropriate for the supplied `Variable` object(s):

```
>>> if __name__ == '__main__':
...     import fipy.viewers
...     viewer = fipy.viewers.make(vars = var)
...     viewer.plot()
```

Please do not hesitate to contact us if this example does not help you convert your existing scripts to FiPy 1.0.

Bibliography

[1] The Python Programming Language, URL http://www.python.org/. 11

[2] W. J. Boettinger, J. A. Warren, C. Beckermann, and A. Karma, "Phase-field simulation of solidification". *Annual Review of Materials Research*, **32**, (2002) 163–194, URL http://arjournals.annualreviews.org/doi/abs/10.1146/annurev.matsci.32.101901.155803. 11, 103, 112

[3] L. Q. Chen, "Phase-field mdoels for microstructure evolution". *Annual Review of Materials Research*, **32**, (2002) 113–140, URL http://arjournals.annualreviews.org/doi/pdf/10.1146/annurev.matsci.32.112001.132041. 11, 103

[4] G. B. McFadden, "Phase-field models of solidification". *Contemporary Mathematics*, **306**, (2002) 107–145. 11, 103, 112

[5] D. Josell, D. Wheeler, W. H. Huber, and T. P. Moffat, "Superconformal Electrodeposition in Submicron Features". *Physical Review Letters*, **87**(1), (2001) 016102, URL http://link.aps.org/abstract/PRL/v87/e016102. 11, 147

[6] FiPy mailing list, URL http://www.ctcms.nist.gov/fipy/mail.html. 12, 61

[7] FiPy bug tracker, URL http://matforge.org/fipy/report. 12, 61

[8] Greg Ward, *Installing Python Modules*. URL http://docs.python.org/inst/. 16

[9] Python download page, URL http://www.python.org/download/. 17

[10] Matplotlib download page, URL http://sourceforge.net/project/showfiles.php?group_id=80706&package_id=82474. 18

[11] FiPy download page, URL http://www.ctcms.nist.gov/fipy/download/. 20

[12] Ben Collins-Sussman, Brian W. Fitzpatrick, and C. Michael Pilato, *Version Control with Subversion*. O'Reilly Media, 2004, URL http://svnbook.red-bean.com. 20, 36

[13] 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. 20

[14] Guido van Rossum, *Python Tutorial*. URL http://docs.python.org/tut/. 22

[15] Mark Pilgrim, *Dive Into Python*. Apress, 2004, ISBN 1590593561, URL http://diveintopython.org. 22

[16] SciPy download page, URL http://www.scipy.org/Download. 22

[17] T. N. Croft, *Unstructured Mesh - Finite Volume Algorithms for Swirling, Turbulent Reacting Flows*. Ph.D. thesis, University of Greenwich, 1998, URL http://www.gre.ac.uk/~ct02/research/thesis/main.html. 37, 40

[18] S. V. Patanker, *Numerical Heat Transfer and Fluid Flow*. Taylor and Francis, 1980. 37

[19] H. K. Versteeg and W. Malalasekera, *An Introduction to Computational Fluid Dynamics*. Longman Scientific and Technical, 1995. 37

[20] C. Mattiussi, "An analysis of finite volume, finite element, and finite difference methods using some concepts from algebraic topology". *Journal of Computational Physics*, **133**, (1997) 289–309, URL http://lis.epfl.ch/publications/JCP1997.pdf. 37

[21] John W. Cahn and John E. Hilliard, "Free energy of a nonuniform system. I. Interfacial free energy". *Journal of Computational Physics*, **28**(2), (1958) 258–267. 38

[22] John W. Cahn, "Free energy of a nonuniform system. II. Thermodynamic basis". *Journal of Computational Physics*, **30**(5), (1959) 1121–1124. 38

[23] John W. Cahn and John E. Hilliard, "Free energy of a nonuniform system. III. Nucleation in a two-component incompressible fluid". *Journal of Computational Physics*, **31**(3), (1959) 688–699. 38

[24] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery, *Numerical Recipes in C: the Art of Scientific Computing*. Cambridge University Press, 2nd edition, 1999. 73

[25] J. A. Warren and W. J. Boettinger, "Prediction of Dendritic Growth and Microsegregation in a Binary Alloy using the Phase Field Method". *Acta Metallurgica et Materialia*, **43**(2), (1995) 689–703. 109

[26] A. A. Wheeler, W. J. Boettinger, and G. B. McFadden, "Phase-field model for isothermal phase transitions in binary alloys". *Physical Review A*, **45**(10), (1992) 7424–7439. 112

[27] J. E. Guyer, W. J. Boettinger, J. A. Warren, and G. B. McFadden, "Phase field modeling of electrochemistry I: Equilibrium". *Physical Review E*, **69**, (2004) 021603, cond-mat/0308173, URL http://link.aps.org/abstract/PRE/v69/e021603. 123

[28] J. E. Guyer, W. J. Boettinger, J. A. Warren, and G. B. McFadden, "Phase field modeling of electrochemistry II: Kinetics". *Physical Review E*, **69**, (2004) 021604, cond-mat/0308179, URL http://link.aps.org/abstract/PRE/v69/e021604. 123

[29] James A. Warren, Ryo Kobayashi, Alexander E. Lobkovsky, and W. Craig Carter, "Extending Phase Field Models of Solidification to Polycrystalline Materials". *Acta Materialia*, **51**(20), (2003) 6035–6058, URL http://dx.doi.org/10.1016/S1359-6454(03)00388-4. 128, 132

[30] J. A. Sethian, *Level Set Methods and Fast Marching Methods*. Cambridge University Press, 1996. 141

[31] T. P. Moffat, D. Wheeler, and D. Josell, "Superfilling and the Curvature Enhanced Accelerator Coverage Mechanism". *The Electrochemical Society, Interface*, **13**(4), (2004) 46–52, URL http://www.electrochem.org/publications/interface/winter2004/IF12-04-Pg46.pdf. 147

[32] D. Wheeler, D. Josell, and T. P. Moffat, "Modeling Superconformal Electrodeposition Using The Level Set Method". *Journal of The Electrochemical Society*, **150**(5), (2003) C302–C310. 148

[33] D. Josell, D. Wheeler, and T. P. Moffat, "Gold superfill in submicrometer trenches: Experiment and prediction". *Journal of The Electrochemical Society*, **153**(1), (2006) C11–C18. 152

[34] T. P. Moffat, D. Wheeler, S. K. Kim, and D. Josell, "Curvature enhanced adsorbate coverage model for electrodeposition". *Journal of The Electrochemical Society*, **153**(2), (2006) C127–C132. 153

[35] J. H. Ferziger and M. Perić, *Computational Methods for Fluid Dynamics*. Springer, 1996. 171

Index

Contributors

Jon Guyer is a member of the research staff of the Metallurgy Division in the Materials Science and Engineering Laboratory at the National Institute of Standards and Technology. Jon's computational interests are in object-oriented design and in phase field modeling of electrochemistry.

Daniel Wheeler is a caveman. Daniel's interests are in numerical modeling, finite volume techniques, and level set treatments.

Jim Warren is the group leader of the Themodynamics and Kinetics group in the Metallurgy Division and Director of the Center for Theoretical and Computational Materials Science of the Materials Science and Engineering Laboratory at the National Institute of Standards and Technology. Jim is interested in a variety of problems, including the phase field modeling of solidification, polycrystalline solids, and the electrochemical interface.

Alex Mont developed the *PyxViewer* and the *Gmsh* import and export modules while he was a student at Montgomery Blair High School.

Katie Travis developed the automated `--inline` optimization code for `Variable` objects while she was a SURF student from Smith College.

Max Gibiansky added support for the Trilinos solvers while he was a SURF student from Harvey Mudd College

Andrew Reeve added support for anistropic diffusion coefficients while he was on sabbatical from the University of Maine.