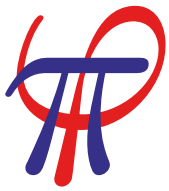


FiPy Manual

Release 0+unknown

Jonathan E. Guyer
Daniel Wheeler
James A. Warren

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

Aug 08, 2023

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

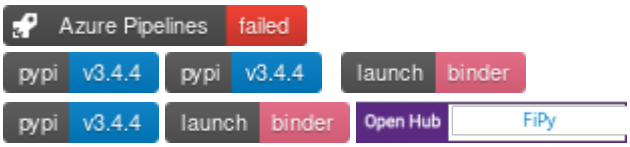
Contents

Part I

Introduction

Chapter 1

Overview



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

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

The *FiPy* framework includes terms for transient diffusion, convection and standard sources, enabling the solution of arbitrary combinations of coupled elliptic, hyperbolic and parabolic PDEs. Currently implemented models include phase field [1] [2] [3] treatments of polycrystalline, dendritic, and electrochemical phase transformations, as well as drug eluting stents [4], reactive wetting [5], photovoltaics [6] and a level set treatment of the electrodeposition process [7].

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

See the latest updates in the *Change Log*.

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

...please read *Installation*, *Using FiPy* and *Frequently Asked Questions*, as well as `examples.diffusion.mesh1D`.

1.2 Download and Installation

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

1.3 Support

We offer several modes to communicate with the *FiPy* developers and with other users.

1.3.1 Contact

In order to discuss *FiPy* with other users and with the developers, we encourage you to use one of the following modes of communication. We monitor all of these, so there is no need to post to more than one of them.

You may want to read the following resource about asking effective questions: http://www.catb.org/~esr/faqs/smart-questions.html

If you are having trouble, we are able to offer much more effective help if you provide a minimal reproducible example.

GitHub Discussions

https://github.com/usnistgov/fipy/discussions

Suitable for open-ended conversations, troubleshooting, showing off...

If a discussion highlights a bug or feature request, it's easy for us to migrate *GitHub Discussions* to *GitHub Issues*.

GitHub Issues

https://github.com/usnistgov/fipy/issues

Suitable for bug reports, feature requests, and patch submissions.

StackOverflow

https://stackoverflow.com/questions/tagged/fipy

Suitable for questions that (probably) have definitive answers ("How do I...?"). It doesn't work so well for back-and-forth conversations, which are better suited to *GitHub Discussions*. Further, it's bad at math and they tend to delete answers that link to our existing documentation, meaning that we'd need to expend considerable effort, using an inferior tool, to duplicate things we've already written.

Seriously, use *GitHub Discussions*.

Mailing List

Attention: The mailing list is deprecated. Please use *GitHub Discussions*, instead.

You can sign up for the mailing list by sending a subscription email to <mailto:fipy+subscribe@list.nist.gov>.

Once you are subscribed, you can post messages to the list simply by addressing email to <mailto:fipy@list.nist.gov>.

To get off the list, send a message to <mailto:fipy+unsubscribe@list.nist.gov>.

Send a message to <mailto:fipy+help@list.nist.gov> to learn other mailing list configurations you can change.

The list is hosted as a Google group. If you are subscribed with a Google account, you can interact with the list, configure your subscription, and see the archives at https://list.nist.gov/fipy.

List Archive

https://www.mail-archive.com/fipy@list.nist.gov/

Copies of messages sent to fipy@list.nist.gov are stored at The Mail Archive.

Older messages are archived at https://www.mail-archive.com/fipy@nist.gov/.

(note: we have also historically sent copies to http://dir.gmane.org/gmane.comp.python.fipy, but the GMANE site now appears to be defunct.)

We welcome collaborative efforts on this project.

1.4 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, referred to as the "base directory", such that:

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

will correspond to, *e.g.*:

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

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

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

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

We may at times use a

Note: to indicate something that may be of interest

or a

Warning: to indicate something that could cause serious problems.

Chapter 2

Installation

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

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

2.1 Pre-Installed on Binder

A full *FiPy* installation is available for basic exploration on Binder. The default notebook gives a rudimentary introduction to *FiPy* syntax and, like any Jupyter Notebook interface, tab completion will help you explore the package interactively.

2.2 Recommended Method

Attention: There are many ways to obtain the software packages necessary to run *FiPy*, but the most expedient way is with the conda package manager. In addition to the scientific *Python* stack, conda also provides virtual environment management. Keeping separate installations is useful *e.g.* for comparing *Python* 2 and *Python* 3 software stacks, or when the user does not have sufficient privileges to install software system-wide.

In addition to the default packages, many other developers provide "channels" to distribute their own builds of a variety of software. These days, the most useful channel is *conda-forge*, which provides everything necessary to install *FiPy*.

- install Miniconda on your computer
- run:

```
$ conda create --name <MYFIPYENV> --channel conda-forge python=<PYTHONVERSION> fipy␣
↪gmsh
```

Note: This command creates a self-contained conda environment and then downloads and populates the environment with the prerequisites for *FiPy* from the conda-forge channel at https://anaconda.org.

Gmsh is an optional package because some versions are incompatible with *FiPy*, so it must be requested explicitly.

Note: The fipy conda-forge package is a convenience. You may choose to install packages explicitly, e.g.,:

```
$ conda create --name <MYFIPYENV> --channel conda-forge python=3 numpy scipy␣
↪matplotlib-base future packaging mpich mpi4py petsc4py mayavi "gmsh <4.0|>=4.5.2"
```

or

$ conda create –name <MYFIPYENV> –channel conda-forge python=2.7 numpy scipy matplotlib-base future packaging pysparse mayavi “traitsui<7.0.0” “gmsh<4.0”

Attention: Windows x86_64 is fully supported, but this does not work on Windows x86_32, as conda-forge no longer supports that platform. For Python 2.7.x, you should be able to do:

```
conda create --name <MYFIPYENV> --channel conda-forge python=2.7 numpy scipy␣
↪matplotlib pysparse mayavi weave
```

and for Python 3.x, you should be able to do:

```
conda create --name <MYFIPYENV> --channel conda-forge python=3 numpy scipy␣
↪matplotlib pysparse gmsh
```

followed, for either, by:

```
activate <MYFIPYENV>
python -m pip install fipy
```

Attention: Bit rot has started to set in for Python 2.7. One consequence is that `VTKViewers` can raise errors (probably other uses of *Mayavi*, too). You may be able to remedy this by creating your environment with:

```
$ conda create --name <MYFIPYENV> --channel conda-forge python=2.7 fipy "traitsui
↪<7.0.0"
```

- enable this new environment with:

```
$ conda activate <MYFIPYENV>
```

or

$ source activate <MYFIPYENV>

Note: `$ activate <MYFIPYENV>` on Windows

- move on to *Using FiPy*.

Note: conda can be quite slow to resolve all dependencies when performing an installation. You may wish to consider using the alternative mamba installation manager to speed things up.

Note: On Linux and Mac OS X, you should have a pretty complete system to run and visualize *FiPy* simulations. On Windows, there are fewer packages available via conda, particularly amongst the sparse matrix *Solvers*, but the system still should be functional. Significantly, you will need to download and install *Gmsh* manually when using Python 2.7.

Attention: When installed via conda or *pip*, *FiPy* will not include its *examples*. These can be obtained by *cloning the repository* or downloading a compressed archive.

2.3 Obtaining FiPy

FiPy is freely available for download via Git or as a compressed archive. Please see *Git usage* for instructions on obtaining *FiPy* with Git.

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 Git will handle this issue automatically).

2.4 Installing FiPy

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

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

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

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

```
$ python -m pip install fipy
```

or by unpacking the archive and typing:

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

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

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

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

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

2.5 Required Packages

2.5.1 Python

http://www.python.org/

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

```
$ python
Python 2.7.15 | ...
...
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

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

Note: *FiPy* requires at least version 2.7.x of *Python*.

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

2.5.2 NumPy

http://numpy.scipy.org

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

2.6 Optional Packages

2.6.1 Gmsh

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

Gmsh is an application that allows the creation of irregular meshes. When running in parallel, *FiPy* requires a version of *Gmsh* >= 2.5 and < 4.0 or >= 4.5.2.

2.6.2 SciPy

http://www.scipy.org/

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

2.7 Level Set Packages

To use the level set ([8]) components of *FiPy* one of the following is required.

2.7.1 Scikit-fmm

http://packages.python.org/scikit-fmm/

Scikit-fmm is a python extension module which implements the fast marching method.

2.7.2 LSMLIB

http://ktchu.serendipityresearch.org/software/lsmlib/index.html

The Level Set Method Library (LSMLIB) provides support for the serial and parallel simulation of implicit surface and curve dynamics in two- and three-dimensions.

Install LSMLIB as per the instructions on the website. Additionally PyLSMLIB is required. To install, follow the instructions on the website, https://github.com/ktchu/LSMLIB/tree/master/pylsmlib#pylsmlib.

2.8 Development Environment

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

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

To avoid tampering with the system *Python* installation, you can employ one of the utilities that manage packages and their dependencies independently of the system package manager and the system directories. These utilities include conda, Nix, Stow, Virtualenv and Buildout, amongst others. Conda and Nix are only ones of these we have the resources to support.

Our preferred development environment is set up with:

```
$ conda create --name <MYFIPYENV> --channel conda-forge python=<PYTHONVERSION> fipy
$ source activate <MYFIPYENV>
$ python -m pip install scikit-fmm
$ conda remove --channel conda-forge --force fipy
$ git clone https://github.com/usnistgov/fipy.git
$ cd fipy
$ python setup.py develop
```

2.9 Git usage

All stages of *FiPy* development are archived in a Git repository at GitHub. You can browse through the code at https://github.com/usnistgov/fipy and, using a *Git client*, you can download various tagged revisions of *FiPy* depending on your needs.

Attention: Be sure to follow *Installation* to obtain all the prerequisites for *FiPy*.

2.9.1 Git client

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

2.9.2 Git branches

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 branches are used to indicate different stages of *FiPy* development. For the most part, we follow a successful Git branching model. You will need to decide on your own risk tolerance when deciding which stage of development to track.

A fresh copy of the *FiPy* source code can be obtained with:

```
$ git clone https://github.com/usnistgov/fipy.git
```

An existing Git checkout of FiPy can be shifted to a different <*branch*> of development by issuing the command:

```
$ git checkout <branch>
```

in the base directory of the working copy. The main branches for FiPy are:

`master`
: designates the (ready to) release state of FiPy. This code is stable and should pass all of the tests (or should be documented that it does not).

Past releases of FiPy are tagged as

`x.y.z`
: Any released version of FiPy will be designated with a fixed tag: The current version of FiPy is 0+unknown. (Legacy `version-x_y_z` tags are retained for historical purposes, but won't be added to.)

Tagged releases can be found with:

```
$ git tag --list
```

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

Note: For some time now, we have done all significant development work on branches, only merged back to `master` when the tests pass successfully. Although we cannot guarantee that `master` will never be broken, you can always check our *Continuous Integration* status to find the most recent revision that it is running acceptably.

Historically, we merged to `develop` before merging to `master`. We no longer do this, although for time being, `develop` is kept synchronized with `master`. In a future release, we will remove the `develop` branch altogether.

For those who are interested in learning more about Git, a wide variety of online sources are available, starting with the official Git website. The Pro Git book [9] is particularly instructive.

2.10 Nix

2.10.1 Nix Installation

FiPy now has a Nix expression for installing *FiPy* using Nix. Nix is a powerful package manager for Linux and other Unix systems that makes package management reliable and reproducible. The recipe works on both Linux and Mac OS X.

Getting Started with Nix

There are a number of tutorials on getting started with Nix. The page that I used when getting started is on the Blog of the HPC team of GRICAD,

https://gricad.github.io/calcul/nix/tuto/2017/07/04/nix-tutorial.html

I also made my own notes,

https://github.com/wd15/nixes/blob/master/NIX-NOTES.md

which are a succinct steps that I use when setting up a new system with Nix.

Installing

Once you have a working Nix installation use:

```
$ nix-shell --pure
```

in the base *FiPy* directory to install *FiPy* with Python 3 by default. Modify the *shell.nix* file to use another version of Python. `nix-shell` drops the user into a shell with a working version of *FiPy*. To test your installation use:

```
$ nix-shell --pure --command "python setup.py test"
```

Note: *Trilinos* is currently not available as part of the Nix *FiPy* installation.

Additional Packages

To install additional packages available from Nixpkgs include them in the *nativeBuildInputs* list in *shell.nix*.

Using Pip

Packages unavailable from Nix can be installed using *Pip*. In this case, the installation has been set up so that the Nix shell knows about a `.local` directory in the base *FiPy* directory used by *Pip* for installation. So, for example, to install the `toolz` package from within the Nix shell use:

```
$ python -m pip install --user toolz
```

The `.local` directory will persist after the Nix shell has been closed.

Chapter 3

Git practices

Refer to *Git usage* for the current branching conventions.

3.1 Branches

Whether fixing a bug or adding a feature, all work on FiPy should be conducted on a branch and submitted as a pull request. If there is already a reported GitHub issue, name the branch accordingly:

```
$ BRANCH=issue12345-Summary_of_what_branch_addresses
$ git checkout -b $BRANCH master
```

Edit and add to branch:

```
$ emacs ...
$ git commit -m "refactoring_stage_A"
$ emacs ...
$ git commit -m "refactoring_stage_B"
```

3.1.1 Merging changes from master to the branch

Make sure `master` is up to date:

```
$ git fetch origin
```

Merge updated state of `master` to the branch:

```
$ git diff origin/master
$ git merge origin/master
```

Resolve any conflicts and test:

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

3.1.2 Submit branch for code review

If necessary, fork the fipy repository.

Add a "remote" link to your fork:

```
$ git remote add <MYFORK> <MYFORKURL>
```

Push the code to your fork on GitHub:

```
$ git push <MYFORK> $BRANCH
```

Now create a pull request from your `$BRANCH` against the `master` branch of `usnistgov/fipy`. The pull request should initiate automated testing. Check the *Continuous Integration* status. Fix (or, if absolutely necessary, document) any failures.

Note: If your branch is still in an experimental state, but you would like to check its impact on the tests, you may prepend "`WIP:`" to your pull request title. This will prevent your branch from being merged before it's complete, but will allow the automated tests to run.

Please be respectful of the *Continuous Integration* resources and do the bulk of your testing on your local machine or against your own *Continuous Integration* accounts (if you have a lot of testing to do, before you create a pull request, push your branch to your own fork and enable the *Continuous Integration* services there.

You can avoid testing individual commits by adding "`[skip ci]`" to the commit message title.

When your pull request is ready and successfully passes the tests, you can request a pull request review or send a message to the mailing list about it if you like, but the FiPy developers should automatically see the pull request and respond to it without further action on your part.

3.1.3 Refactoring complete: merge branch to master

Attention: Administrators Only!

Use the GitHub interface to merge the pull request.

Note: Particularly for branches with a long development history, consider doing a Squash and merge.

Chapter 4

Continuous Integration

We use the *Azure* cloud service for *Continuous Integration* (CI). This CI is configured in `FiPySource/.azure/pipelines.yml`.

Chapter 5

Making a Release

Attention: Administrators Only!

5.1 Source

Make sure `master` is ready for release:

```
$ git checkout master
```

Check the issue list and update the *Change Log*:

```
$ git commit CHANGELOG.txt -m "REL: update new features for release"
```

Note: You can use:

```
$ python setup.py changelog --after=<x.y>
```

or:

```
$ python setup.py changelog --milestone=<x.z>
```

to obtain a ReST-formatted list of every GitHub pull request and issue closed since the last release.

Particularly for major and feature releases, be sure to curate the output so that it's clear what's a big deal about this release. Sometimes a pull request will be redundant to an issue, e.g., "`Issue123 blah blah`". If the pull request fixes a bug, preference is given to the corresponding issue under **Fixes**. Alternatively, if the pull request adds a new feature, preference is given to the item under **Pulls** and corresponding issue should be removed from **Fixes**. If appropriate, be sure to move the "Thanks to @mention" to the appropriate issue to recognize outside contributors.

Attention: Requires PyGithub and Pandas.

Attention: If *Continuous Integration* doesn't show all green boxes for this release, make sure to add appropriate notes in `README.txt` or `INSTALLATION.txt`!

5.2 Release from master

```
$ git checkout master
```

Resolve any conflicts and tag the release as appropriate (see *Git practices* above):

```
$ git tag --annotate x.y master
```

Push the tag to GitHub:

```
$ git push --tags origin master
```

Upon successful completion of the *Continuous Integration* systems, fetch the tagged build products from Azure Artifacts and place in *FiPySource*`/dist/`:

- `dist-Linux/FiPy-`*x.y*`-none-any.whl`
- `dist-Linux/FiPy-`*x.y*`.tar.gz`
- `dist-Windows_NT/FiPy-`*x.y*`.zip`
- `dist-docs/FiPy-`*x.y*`.pdf`
- `dist-docs/html-`*x.y*`.tar.gz`

From the *FiPySource* directory, unpack `dist/html-`*x.y*`.tar.gz` into `docs/build` with:

```
$ tar -xzf dist/html-{x.y}.tar.gz -C docs/build
```

5.3 Upload

Attach

- `dist/FiPy-`*x.y*`-none-any.whl`
- `dist/FiPy-`*x.y*`.tar.gz`
- `dist/FiPy-`*x.y*`.zip`
- `dist/FiPy-`*x.y*`.pdf`

to a GitHub release associated with tag x.y.

Upload the build products to PyPI with twine:

```
$ twine upload dist/FiPy-${FIPY_VERSION}.*
```

Upload the web site to CTCMS

```
$ export FIPY_WWWHOST=bunter:/u/WWW/wd15/fipy
$ export FIPY_WWWACTIVATE=updatewww
$ python setup.py upload_products --html
```

Warning: Some versions of `rsync` on Mac OS X have caused problems when they try to upload erroneous `\rsrc` directories. Version 2.6.2 does not have this problem.

5.4 Update conda-forge feedstock

Once you push the tag to GitHub, the fipy-feedstock should automatically receive a pull request. Review and amend this pull request as necessary and ask the feedstock maintainers to merge it.

This automated process only runs once an hour, so if you don't wish to wait (or it doesn't trigger for some reason), you can manually generate a pull request to update the fipy-feedstock with:

- revised version number
- revised sha256 (use `openssl dgst -sha256 /path/to/fipy-x.y.tar.gz`)
- reset build number to `0`

5.5 Announce

Make an announcement to fipy@list.nist.gov

Chapter 6

Solvers

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

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

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

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

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

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

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

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

6.1 PETSc

https://www.mcs.anl.gov/petsc

PETSc (the Portable, Extensible Toolkit for Scientific Computation) is a suite of data structures and routines for the scalable (parallel) solution of scientific applications modeled by partial differential equations. It employs the *MPI* standard for all message-passing communication (see *Solving in Parallel* for more details).

Attention: *PETSc* requires the *petsc4py* and *mpi4py* interfaces.

Note: *FiPy* does not implement any precoditioner objects for *PETSc*. Simply pass one of the PCType strings in the *precon*= argument when declaring the solver.

6.2 Pysparse

http://pysparse.sourceforge.net

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

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

6.3 SciPy

http://www.scipy.org/

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

6.4 PyAMG

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

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

6.5 pyamgx

https://pyamgx.readthedocs.io/

The *pyamgx* package is a *Python* interface to the NVIDIA AMGX library. *pyamgx* can be used to construct complex solvers and preconditioners to solve sparse sparse linear systems on the GPU.

6.6 Trilinos

http://trilinos.sandia.gov

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

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

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

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

Attention: *Trilinos must* be compiled with *MPI* support for *Solving in Parallel*.

Tip: *Trilinos* parallel efficiency is greatly improved by also installing *Pysparse*. If *Pysparse* is not installed, be sure to use the `--no-pysparse` flag.

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.

Chapter 7

Viewers

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

7.1 Matplotlib

http://matplotlib.sourceforge.net

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

7.2 Mayavi

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

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

Mayavi requires VTK, which can be difficult to build from source.

Note: MayaVi 1 is no longer supported.

Chapter 8

Using FiPy

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

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

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

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

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

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

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

In order to customize the examples, or to develop your own scripts, some knowledge of Python syntax is required. We recommend you familiarize yourself with the excellent Python tutorial [10] or with Dive Into Python [11]. Deeper insight into Python can be obtained from the [12].

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

8.1 Logging

Diagnostic information about a *FiPy* run can be obtained using the `logging` module. For example, at the beginning of your script, you can add:

```
>>> import logging
>>> log = logging.getLogger("fipy")
>>> console = logging.StreamHandler()
>>> console.setLevel(logging.INFO)
>>> log.addHandler(console)
```

in order to see informational messages in the terminal. To have more verbose debugging information save to a file:

```
>>> logfile = logging.FileHandler(filename="fipy.log")
>>> logfile.setLevel(logging.DEBUG)
>>> log.addHandler(logfile)

>>> log.setLevel(logging.DEBUG)
```

To restrict logging to, e.g., information about the *PETSc* solvers:

```
>>> petsc = logging.Filter('fipy.solvers.petsc')
>>> logfile.addFilter(petsc)
```

More complex configurations can be specified by setting the *FIPY_LOG_CONFIG* environment variable. In this case, it is not necessary to add any logging instructions to your own script. Example configuration files can be found in *FiPySource*`/fipy/tools/logging/`.

If *Solving in Parallel*, the mpilogging package enables reporting which MPI rank each log entry comes from. For example:

```
>>> from mpilogging import MPIScatteredFileHandler
>>> mpilog = MPIScatteredFileHandler(filepattern="fipy.%(mpirank)d_of_%(mpisize)d.log"
>>> mpilog.setLevel(logging.DEBUG)
>>> log.addHandler(mpilog)
```

will generate a unique log file for each MPI rank.

8.2 Testing FiPy

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

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

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

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

Note: You may need to first run:

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

for this to work properly.

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

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

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

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

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

8.2.1 Parallel Tests

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

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

or:

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

8.3 Command-line Flags and Environment Variables

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

8.3.1 Command-line Flags

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

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

`--inline`

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

`--cache`

Causes lazily evaluated *FiPy* `Variable` objects to retain their value.

--no-cache

Causes lazily evaluated *FiPy* `Variable` objects to always recalculate their value.

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

--pysparse

Forces the use of the *Pysparse* solvers.

--trilinos

Forces the use of the *Trilinos* solvers, but uses *Pysparse* to construct the matrices.

--no-pysparse

Forces the use of the *Trilinos* solvers without any use of *Pysparse*.

--scipy

Forces the use of the *SciPy* solvers.

--pyamg

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

--pyamgx

Forces the use of the *pyamgx* solvers.

--lsmlib

Forces the use of the *LSMLIB* level set solver.

--skfmm

Forces the use of the *Scikit-fmm* level set solver.

8.3.2 Environment Variables

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

FIPY_DISPLAY_MATRIX

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

FIPY_INLINE

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

FIPY_INLINE_COMMENT

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

FIPY_LOG_CONFIG

Specifies a *JSON*-formatted logging configuration file, suitable for passing to `logging.config.dictConfig()`. Example configuration files can be found in *FiPySource*`/fipy/tools/logging/`.

FIPY_SOLVERS

Forces the use of the specified suite of linear solvers. Valid (case-insensitive) choices are "`petsc`", "`scipy`", "`pysparse`", "`trilinos`", "`no-pysparse`", and "`pyamg`".

FIPY_VERBOSE_SOLVER

If present, causes the `LinearGeneralSolver` to print a variety of diagnostic information. All other solvers should use *Logging* and *FIPY_LOG_CONFIG*.

FIPY_VIEWER

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

FIPY_INCLUDE_NUMERIX_ALL

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

FIPY_CACHE

If present, causes lazily evaluated *FiPy* `Variable` objects to retain their value.

PETSC_OPTIONS

PETSc configuration options. Set to "*-help*" and run a script with *PETSc* solvers in order to see what options are possible. Ignored if solver is not *PETSc*.

8.4 Solving in Parallel

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

Attention: *FiPy* requires *mpi4py* to work in parallel.

Tip: You are strongly advised to force the use of only one *OpenMP* thread with *Trilinos*:

```
$ export OMP_NUM_THREADS=1
```

See *OpenMP Threads vs. MPI Ranks* for more information.

Note: Trilinos 12.12 has support for Python 3, but PyTrilinos on conda-forge presently only provides 12.10, which is limited to Python 2.x. *PETSc* is available for both *Python 3* and *Python* 2.7.

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

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

or:

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

instead of:

```
$ python myScript.py
```

The following plot shows the scaling behavior for multiple processors. We compare solution time vs number of Slurm tasks (available cores) for a Method of Manufactured Solutions Allen-Cahn problem.

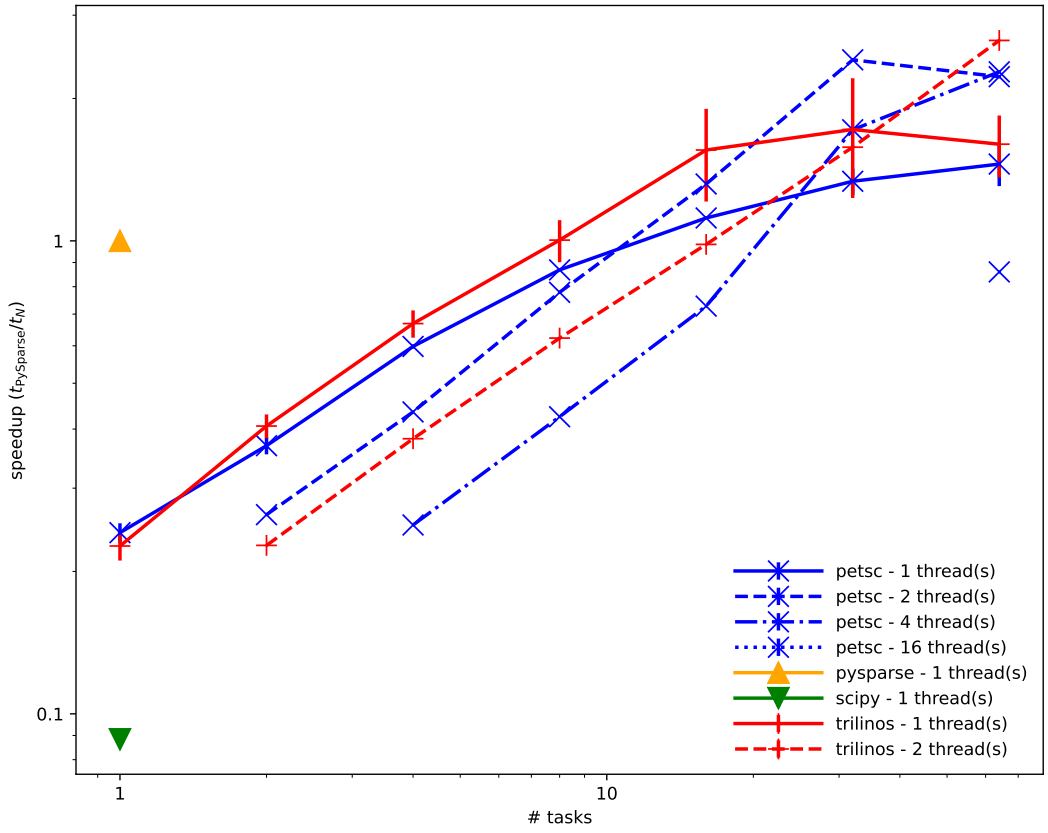


Fig. 1: Scaling behavior of different solver packages

"Speedup" relative to *Pysparse* (bigger numbers are better) versus number of tasks (processes) on a log-log plot. The number of threads per *MPI* rank is indicated by the line style (see legend). *OpenMP* threads × *MPI* ranks = Slurm tasks.

A few things can be observed in this plot:

- Both *PETSc* and *Trilinos* exhibit power law scaling, but the power is only about 0.7. At least one source of this poor scaling is that our "`...Grid...`" meshes parallelize by dividing the mesh into slabs, which leads to more communication overhead than more compact partitions. The "`...Gmsh...`" meshes partition more efficiently, but carry more overhead in other ways. We'll be making efforts to improve the partitioning of the "`...Grid...`" meshes in a future release.
- *PETSc* and *Trilinos* have fairly comparable performance, but lag *Pysparse* by a considerable margin. The *SciPy* solvers are even worse. Some of this discrepancy may be because the different packages are not all doing the same thing. Different solver packages have different default solvers and preconditioners. Moreover, the meaning of the solution tolerance depends on the normalization the solver uses and it is not always obvious which of several possibilities a particular package employs. We will endeavor to normalize the normalizations in a future release.
- *PETSc* with one thread is faster than with two threads until the number of tasks reaches about 10 and is faster than with four threads until the number of tasks reaches more than 20. *Trilinos* with one thread is faster than with two threads until the number of tasks is more than 30. We don't fully understand the reasons for this, but there may be a *modest* benefit, *when using a large number of cpus*, to allow two to four *OpenMP* threads per *MPI* rank. See *OpenMP Threads vs. MPI Ranks* for caveats and more information.

These results are likely both problem and architecture dependent. You should develop an understanding of the scaling behavior of your own codes before doing "production" runs.

The easiest way to confirm that *FiPy* is properly configured to solve in parallel is to run one of the examples, e.g.,:

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

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

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

which should print out:

```
     mpi4py           PyTrilinos           petsc4py                 FiPy
processor 0 of 3 :: processor 0 of 3 :: processor 0 of 3 :: 5 cells on processor 0 of 3
processor 1 of 3 :: processor 1 of 3 :: processor 1 of 3 :: 7 cells on processor 1 of 3
processor 2 of 3 :: processor 2 of 3 :: processor 2 of 3 :: 6 cells on processor 2 of 3
```

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

```
     mpi4py           PyTrilinos           petsc4py                 FiPy
processor 0 of 3 :: processor 0 of 1 :: processor 0 of 3 :: 10 cells on processor 0 of 1
[my.machine.com:69815] WARNING: There were 4 Windows created but not freed.
processor 1 of 3 :: processor 0 of 1 :: processor 1 of 3 :: 10 cells on processor 0 of 1
[my.machine.com:69814] WARNING: There were 4 Windows created but not freed.
processor 2 of 3 :: processor 0 of 1 :: processor 2 of 3 :: 10 cells on processor 0 of 1
[my.machine.com:69813] WARNING: There were 4 Windows created but not freed.
```

indicates *mpi4py* is properly communicating with *MPI* and is running in parallel, but that *Trilinos* is not, and is running three separate serial environments. As a result, *FiPy* is limited to three separate serial operations, too. In this instance,

the problem is that although *Trilinos* was compiled with *MPI* enabled, it was compiled against a different *MPI* library than is currently available (and which *mpi4py* was compiled against). The solution, in this instance, is to solve with *PETSc* or to rebuild *Trilinos* against the active *MPI* libraries.

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

Note: One option for debugging in parallel is:

```
$ mpirun -np {# of processors} xterm -hold -e "python -m ipdb myScript.py"
```

8.4.1 OpenMP Threads vs. MPI Ranks

By default, *PETSc* and *Trilinos* spawn as many *OpenMP* threads as there are cores available. This may very well be an intentional optimization, where they are designed to have one *MPI* rank per node of a cluster, so each of the child threads would help with computation but would not compete for I/O resources during ghost cell exchanges and file I/O. However, Python's Global Interpreter Lock (GIL) binds all of the child threads to the same core as their parent! So instead of improving performance, each core suffers a heavy overhead from managing those idling threads.

The solution to this is to force these solvers to use only one *OpenMP* thread:

```
$ export OMP_NUM_THREADS=1
```

Because this environment variable affects all processes launched in the current session, you may prefer to restrict its use to *FiPy* runs:

```
$ OMP_NUM_THREADS=1 mpirun -np {# of processors} python myScript.py --trilinos
```

The difference can be extreme. We have observed the *FiPy* test suite to run in just over two minutes when `OMP_NUM_THREADS=1`, compared to over an hour and 23 minutes when *OpenMP* threads are unrestricted. We don't know why, but other platforms do not suffer the same degree of degradation.

Conceivably, allowing these parallel solvers unfettered access to *OpenMP* threads with no *MPI* communication at all could perform as well or better than purely *MPI* parallelization. The plot below demonstrates this is not the case. We compare solution time vs number of *OpenMP* threads for fixed number of slots for a Method of Manufactured Solutions Allen-Cahn problem. *OpenMP* threading always slows down FiPy performance.

"Speedup" relative to one thread (bigger numbers are better) versus number of threads for 32 Slurm tasks on a log-log plot. *OpenMP* threads $\times$ *MPI* ranks = Slurm tasks.

See https://www.mail-archive.com/fipy@nist.gov/msg03393.html for further analysis.

It may be possible to configure these packages to use only one *OpenMP* thread, but this is not the configuration of the version available from conda-forge and building Trilinos, at least, is NotFun™.

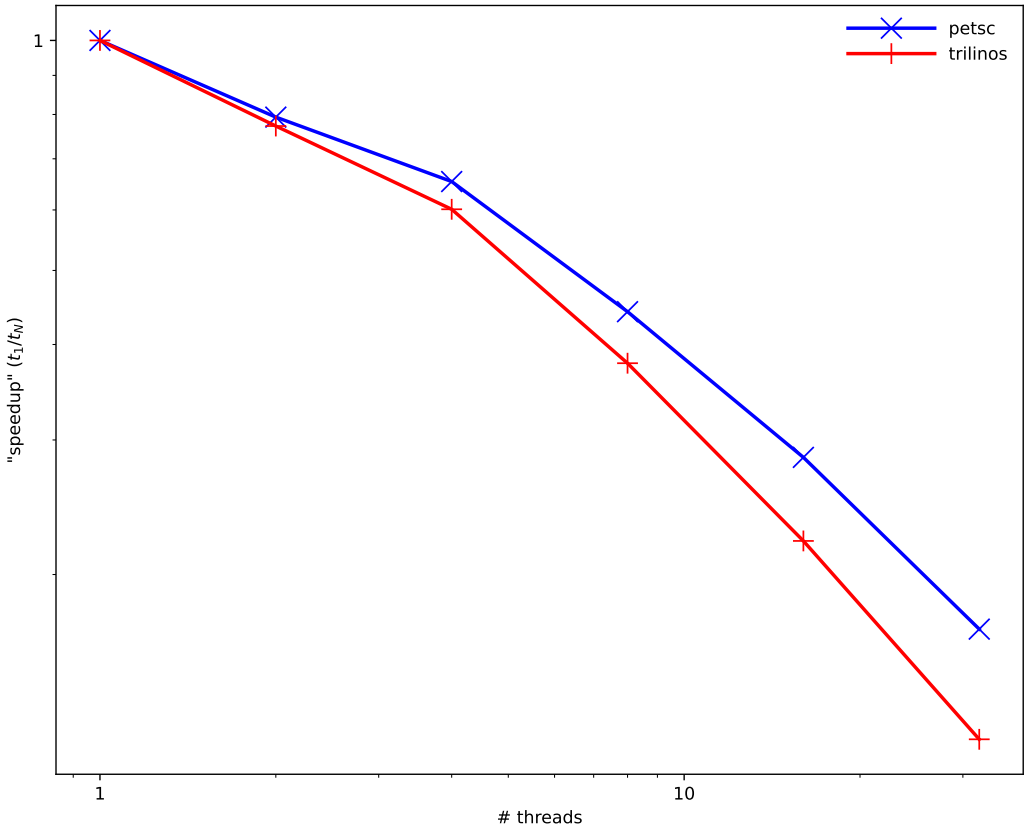


Fig. 2: Effect of having more *OpenMP* threads for each *MPI* rank

8.5 Meshing with Gmsh

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

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

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

8.6 Coupled and Vector Equations

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

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

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

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

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

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

In this case the matrix system will have the form

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

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

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

Attention: Coupled equations are not compatible with *Higher Order Diffusion* terms. This is not a practical limitation, as any higher order terms can be decomposed into multiple 2nd-order equations. For example, the pair of coupled Cahn-Hilliard & Allen-Cahn 4th- and 2nd-order equations

$$\frac{\partial C}{\partial t} = \nabla \cdot \left[M \nabla \left(\frac{\partial f(c, \phi)}{\partial C} - \kappa_C \nabla^2 C \right) \right]$$
$$\frac{\partial \phi}{\partial t} = -L \left(\frac{\partial f(c, \phi)}{\partial \phi} - \kappa_\phi \nabla^2 \phi \right)$$

can be decomposed to three 2nd-order equations

$$\frac{\partial C}{\partial t} = \nabla \cdot (M \nabla \mu)$$
$$\mu = \frac{\partial f(c, \phi)}{\partial C} - \kappa_C \nabla^2 C$$
$$\frac{\partial \phi}{\partial t} = -L \left(\frac{\partial f(c, \phi)}{\partial \phi} - \kappa_\phi \nabla^2 \phi \right)$$

8.7 Boundary Conditions

8.7.1 Default boundary conditions

If no constraints are applied, solutions are conservative, i.e., all boundaries are zero flux. For the equation

$$\frac{\partial \phi}{\partial t} = \nabla \cdot (\vec{a} \phi) + \nabla \cdot (b \nabla \phi)$$

the condition on the boundary S is

$$\hat{n} \cdot (\vec{a} \phi + b \nabla \phi) = 0 \qquad \text{on } S.$$

8.7.2 Applying fixed value (Dirichlet) boundary conditions

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

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

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

8.7.3 Applying fixed gradient boundary conditions (Neumann)

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

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

If the gradient normal to the boundary (*e.g.*, $\hat{n} \cdot \nabla\phi$) is to be set to a scalar value of *2*, use

```
>>> var.faceGrad.constrain(2 * mesh.faceNormals, where=mesh.exteriorFaces)
```

8.7.4 Applying fixed flux boundary conditions

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

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

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

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

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

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

8.7.5 Applying outlet or inlet boundary conditions

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

8.7.6 Applying spatially varying boundary conditions

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

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

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

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

then

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

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

8.7.7 Applying Robin boundary conditions

The Robin condition applied on the portion of the boundary S_R

$$\hat{n} \cdot (\vec{a}\phi + b\nabla\phi) = g \qquad \text{on } S_R$$

can often be substituted for the flux in an equation

$$\begin{aligned} \frac{\partial \phi}{\partial t} &= \nabla \cdot (\vec{a}\phi) + \nabla \cdot (b\nabla\phi) \\ \int_V \frac{\partial \phi}{\partial t}\, dV &= \int_S \hat{n} \cdot (\vec{a}\phi + b\nabla\phi)\, dS \\ \int_V \frac{\partial \phi}{\partial t}\, dV &= \int_{S \notin S_R} \hat{n} \cdot (\vec{a}\phi + b\nabla\phi)\, dS + \int_{S \in S_R} g\, dS \end{aligned}$$

At faces identified by `mask`,

```
>>> a = FaceVariable(mesh=mesh, value=..., rank=1)
>>> a.setValue(0., where=mask)
>>> b = FaceVariable(mesh=mesh, value=..., rank=0)
>>> b.setValue(0., where=mask)
>>> g = FaceVariable(mesh=mesh, value=..., rank=0)
>>> eqn = (TransientTerm() == PowerLawConvectionTerm(coeff=a)
...         + DiffusionTerm(coeff=b)
...         + (g * mask * mesh.faceNormals).divergence)
```

When the Robin condition does not exactly map onto the boundary flux, we can attempt to apply it term by term. The Robin condition relates the gradient at a boundary face to the value on that face, however *FiPy* naturally calculates variable values at cell centers and gradients at intervening faces. Using a first order upwind approximation, the boundary value of the variable at face f can be written in terms of the value at the neighboring cell P and the normal gradient at the boundary:

$$\begin{aligned} \phi_f &\approx \phi_P - \left(\vec{d}_{fP} \cdot \nabla\phi\right)_f \\ &\approx \phi_P - (\hat{n} \cdot \nabla\phi)_f \left(\vec{d}_{fP} \cdot \hat{n}\right)_f \end{aligned} \tag{8.1}$$

where $\vec{d}_{fP}$ is the distance vector from the face center to the adjoining cell center. The approximation $\left(\vec{d}_{fP} \cdot \nabla\phi\right)_f \approx (\hat{n} \cdot \nabla\phi)_f \left(\vec{d}_{fP} \cdot \hat{n}\right)_f$ is most valid when the mesh is orthogonal.

Substituting this expression into the Robin condition:

$$\begin{aligned} \hat{n} \cdot (\vec{a}\phi + b\nabla\phi)_f &= g \\ \hat{n} \cdot \left[\vec{a}\phi_P - \vec{a}\,(\hat{n} \cdot \nabla\phi)_f \left(\vec{d}_{fP} \cdot \hat{n}\right)_f + b\nabla\phi\right]_f &\approx g \\ (\hat{n} \cdot \nabla\phi)_f &\approx \frac{g_f - (\hat{n} \cdot \vec{a})_f \phi_P}{-\left(\vec{d}_{fP} \cdot \vec{a}\right)_f + b_f} \end{aligned} \tag{8.2}$$

we obtain an expression for the gradient at the boundary face in terms of its neighboring cell. We can, in turn, substitute this back into (8.1)

$$\begin{aligned}\phi_f &\approx \phi_P - \frac{g_f - (\hat{n}\cdot\vec{a})_f\,\phi_P}{-\left(\vec{d}_{fP}\cdot\vec{a}\right)_f + b_f}\left(\vec{d}_{fP}\cdot\hat{n}\right)_f \\ &\approx \frac{-g_f\left(\hat{n}\cdot\vec{d}_{fP}\right)_f + b_f\phi_P}{-\left(\vec{d}_{fP}\cdot\vec{a}\right)_f + b_f}\end{aligned} \tag{8.3}$$

to obtain the value on the boundary face in terms of the neighboring cell.

Substituting (8.2) into the discretization of the `DiffusionTerm`:

$$\begin{aligned}\int_V \nabla\cdot(\Gamma\nabla\phi)\,dV &= \int_S \Gamma\hat{n}\cdot\nabla\phi\,S \\ &\approx \sum_f \Gamma_f\left(\hat{n}\cdot\nabla\phi\right)_f A_f \\ &= \sum_{f\notin S_R}\Gamma_f\left(\hat{n}\cdot\nabla\phi\right)_f A_f + \sum_{f\in S_R}\Gamma_f\left(\hat{n}\cdot\nabla\phi\right)_f A_f \\ &\approx \sum_{f\notin S_R}\Gamma_f\left(\hat{n}\cdot\nabla\phi\right)_f A_f + \sum_{f\in S_R}\Gamma_f\frac{g_f - (\hat{n}\cdot\vec{a})_f\,\phi_P}{-\left(\vec{d}_{fP}\cdot\vec{a}\right)_f + b_f}A_f\end{aligned}$$

An equation of the form

```
>>> eqn = TransientTerm() == DiffusionTerm(coeff=Gamma0)
```

can be constrained to have a Robin condition at faces identified by `mask` by making the following modifications

```
>>> Gamma = FaceVariable(mesh=mesh, value=Gamma0)
>>> Gamma.setValue(0., where=mask)
>>> dPf = FaceVariable(mesh=mesh,
...                    value=mesh._faceToCellDistanceRatio * mesh.cellDistanceVectors)
>>> n = mesh.faceNormals
>>> a = FaceVariable(mesh=mesh, value=..., rank=1)
>>> b = FaceVariable(mesh=mesh, value=..., rank=0)
>>> g = FaceVariable(mesh=mesh, value=..., rank=0)
>>> RobinCoeff = (mask * Gamma0 * n / (-dPf.dot(a) + b)
>>> eqn = (TransientTerm() == DiffusionTerm(coeff=Gamma) + (RobinCoeff * g).divergence
...        - ImplicitSourceTerm(coeff=(RobinCoeff * n.dot(a)).divergence)
```

Similarly, for a `ConvectionTerm`, we can substitute (8.3):

$$\begin{aligned}\int_V \nabla\cdot(\vec{u}\phi)\,dV &= \int_S \hat{n}\cdot\vec{u}\phi\,dS \\ &\approx \sum_f (\hat{n}\cdot\vec{u})_f\,\phi_f A_f \\ &= \sum_{f\notin S_R}(\hat{n}\cdot\vec{u})_f\,\phi_f A_f + \sum_{f\in S_R}(\hat{n}\cdot\vec{u})_f\,\frac{-g_f\left(\hat{n}\cdot\vec{d}_{fP}\right)_f + b_f\phi_P}{-\left(\vec{d}_{fP}\cdot\vec{a}\right)_f + b_f}A_f\end{aligned}$$

Note: An expression like the heat flux convection boundary condition $-k\nabla T\cdot\hat{n} = h(T - T_\infty)$ can be put in the form of the Robin condition used above by letting $\vec{a} \equiv h\hat{n}$, $b \equiv k$, and $g \equiv hT_\infty$.

8.7.8 Applying internal “boundary” conditions

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

Internal fixed value

An equation of the form

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

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

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

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

Internal fixed gradient

An equation of the form

```
>>> eqn = TransientTerm() == DiffusionTerm(coeff=Gamma0)
```

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

```
>>> Gamma = FaceVariable(mesh=mesh, value=Gamma0)
>>> Gamma[mask.value] = 0.
>>> eqn = (TransientTerm() == DiffusionTerm(coeff=Gamma)
...        + DiffusionTerm(coeff=largeValue * mask)
...        - ImplicitSourceTerm(mask * largeValue * gradient
...                             * mesh.faceNormals).divergence)
```

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

Internal Robin condition

Nothing different needs to be done when *applying Robin boundary conditions* at internal faces.

Note: While we believe the derivations for *applying Robin boundary conditions* are “correct”, they often do not seem to produce the intuitive result. At this point, we think this has to do with the pathology of “internal” boundary conditions, but remain open to other explanations. *FiPy* was designed with diffuse interface treatments (phase field and level set) in mind and, as such, internal “boundaries” do not come up in our own work and have not received much attention.

Warning: The constraints mechanism is not designed to constrain internal values for variables that are being solved by equations. In particular, one must be careful to distinguish between constraining internal cell values during the solve step and simply applying arbitrary constraints to a `CellVariable`. Applying a constraint,

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

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

A simple example can be used to demonstrate this:

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

We wish to solve $\nabla^2\phi = 0$ subject to $\phi|_{\text{right}} = 1$ and $\phi|_{x<1} = 0.25$. We apply a constraint to the faces for the right side boundary condition (which works).

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

We create the equation with the source term constraint described above

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

and the expected value is obtained.

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

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

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

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

FiPy has simply solved $\nabla^2\phi = 0$ with $\phi|_{\text{right}} = 1$ and (by default) $\hat{n} \cdot \nabla\phi|_{\text{left}} = 0$, giving $\phi = 1$ everywhere, and then subsequently replaced the cells $x < 1$ with $\phi = 0.25$.

8.8 Running under Python 2

Thanks to the future package and to the contributions of pya and woodscn, *FiPy* runs under both *Python 3* and *Python* 2.7, without conversion or modification.

Because *Python* itself will drop support for Python 2.7 on January 1, 2020 and many of the prerequisites for *FiPy* have pledged to drop support for Python 2.7 no later than 2020, we have prioritized adding support for better *Python 3* solvers, starting with *petsc4py*.

Because the faster *PySparse* and *Trilinos* solvers are not available under *Python 3*, we will maintain *Python* 2.x support

as long as practical. Be aware that the conda-forge packages that *FiPy* depends upon are not well-maintained on *Python* 2.x and our support for that configuration is rapidly becoming impractical, despite the present performance benefits. Hopefully, we will learn how to optimize our use of *PETSc* and/or *Trilinos* 12.12 will become available on conda-forge.

8.9 Manual

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

```
$ sphinx-apidoc --output-dir=source/generated/fipy --suffix=rst ../fipy
$ sphinx-apidoc --output-dir=source/generated/examples --suffix=rst ../examples
$ sphinx-apidoc --output-dir=source/generated/tutorial --suffix=rst source/tutorial/
↪package
```

and then:

```
$ make html
```

or:

```
$ make latexpdf
```

Note: This mechanism is intended primarily for the developers. At a minimum, you will need Sphinx, plus all of its prerequisites. We are currently building with Sphinx v7.0. Python 2.7 probably won't work.

We install via conda:

```
$ conda install --channel conda-forge sphinx
```

Bibliographic citations require the *sphinxcontrib-bibtex* package:

```
$ python -m pip install sphinxcontrib-bibtex
```

Some documentation uses *numpydoc* styling:

```
$ python -m pip install numpydoc
```

Some embeded figures require *matplotlib*, *pandas*, and *imagemagick*:

```
$ conda install --channel conda-forge matplotlib pandas imagemagick
```

The PDF file requires SIunits.sty available, e.g., from texlive-science.

Spelling is checked automatically in the course of *Continuous Integration*. If you wish to check manually, you will need *pyspelling*, *hunspell*, and the *libreoffice* dictionaries:

```
$ conda install --channel conda-forge hunspell
$ python -m pip install pyspelling
$ wget -O en_US.aff  https://cgit.freedesktop.org/libreoffice/dictionaries/plain/en/en_
↪US.aff?id=a4473e06b56bfe35187e302754f6baaa8d75e54f
$ wget -O en_US.dic https://cgit.freedesktop.org/libreoffice/dictionaries/plain/en/en_US.
↪dic?id=a4473e06b56bfe35187e302754f6baaa8d75e54f
```

Chapter 9

Frequently Asked Questions

9.1 How do I represent an equation in FiPy?

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

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

and the individual terms are discussed in *Discretization*.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

How do I represent a diffusive source?

If the governing equation for ϕ is

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

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

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

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

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

How do I represent a convective source?

The convection of an independent field ξ as in

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

can be rendered as

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

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

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

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

How do I represent a transient source?

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

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

does not have an abstract form in *FiPy* and should be discretized directly, in the manner of Equation (11.3), as

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

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

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

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

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

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

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

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

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

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

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

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

9.2 How can I see what I'm doing?

9.2.1 How do I export data?

The way to save your calculations depends on how you plan to make use of the data. If you want to save it for "restart" (so that you can continue or redirect a calculation from some intermediate stage), then you'll want to "pickle" the *Python* data with the `dump` module. This is illustrated in `examples.phase.anisotropy`, `examples.phase.impingement.mesh40x1`, `examples.phase.impingement.mesh20x20`, and `examples.levelSet.electroChem.howToWriteAScript`.

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

9.2.2 How do I save a plot image?

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

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

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

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

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

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

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

9.2.3 How do I make a movie?

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

9.2.4 Why doesn't the `Viewer` look the way I want?

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

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

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

iterations

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

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

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

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

sweeps

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

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

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

the variable is created:

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

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

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

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

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

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

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

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

timesteps

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

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

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

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

or more elaborate:

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

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

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

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

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

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

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

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

This distinction is not generally important for smoothly varying functions, but can dramatically affect the solution when sharp changes are present. Also, for many problems, such as a conserved concentration field that cannot be allowed to drop below zero, a harmonic average is more appropriate than an arithmetic average.

If you experience problems (unstable or wrong results, or excessively small timesteps), you may need to explicitly supply the desired `FaceVariable` rather than letting *FiPy* assume one.

9.5 How do I represent boundary conditions?

See the *Boundary Conditions* section for more details.

9.6 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 more locations.`
: 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 `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.

9.7 How do I change FiPy's default behavior?

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

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

See *Solving in Parallel*.

9.9 Why don't my scripts work anymore?

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

9.10 What if my question isn't answered here?

Please post your question to the mailing list <http://www.ctcms.nist.gov/fipy/mail.html> or file an issue at <https://github.com/usnistgov/fipy/issues/new>.

Chapter 10

Efficiency

This section will present results and discussion of efficiency evaluations with *FiPy*. Programming in *Python* allows greater efficiency when designing and implementing new code, but it has some intrinsic inefficiencies during execution as compared with the C or FORTRAN programming languages. These inefficiencies can be minimized by translating sections of code that are used frequently into C.

FiPy has been tested against an in-house phase field code, written at NIST, to model grain growth and subsequent impingement. This problem can be executed by running:

```
$ examples/phase/impingement/mesh20x20.py \
> --numberOfElements=10000 --numberOfSteps=1000
```

from the base *FiPy* directory. The in-house code was written by Ryo Kobayashi and is used to generate the results presented in [13].

The raw CPU execution times for 10 time steps are presented in the following table. The run times are in seconds and the memory usage is in kilobytes. The Kobayashi code is given the heading of FORTRAN while *FiPy* is run with and without inlining. The memory usage is for *FiPy* simulations with the `--inline`. The `--no-cache` flag is on in all cases for the following table.

Elements	FiPy (s)	FiPy `--inline` (s)	FORTRAN (s)	FiPy memory (KiB)	FORTRAN memory (KiB)
100	0.77	0.30	0.0009	39316	772
400	0.87	0.37	0.0031	39664	828
1600	1.4	0.65	0.017	40656	1044
6400	3.7	2.0	0.19	46124	1880
25600	19	10	1.3	60840	5188
102400	79	43	4.6	145820	18436

The plain *Python* version of *FiPy*, which uses `Numeric` for all array operations, is around 17 times slower than the FORTRAN code. Using the `--inline` flag, this penalty is reduced to about 9 times slower.

It is hoped that in future releases of *FiPy* the process of C inlining for `Variable` objects will be automated. This may result in some efficiency gains, greater than we are seeing for this particular problem since all the `Variable` objects will be inlined. Recent analysis has shown that a `Variable` with multiple operations could be up to 6 times faster at calculating its value when inlined.

As presented in the above table, memory usage was also recorded for each *FiPy* simulation. From the table, once base memory usage is subtracted, each cell requires approximately 1.4 kilobytes of memory. The measurement of the

maximum memory spike is hard with dynamic memory allocation, so these figures should only be used as a very rough guide. The FORTRAN memory usage is exact since memory is not allocated dynamically.

10.1 Efficiency comparison between `--no-cache` and `--cache` flags

This table shows results for efficiency tests when using the caching flags. Examples with more variables involved in complex expressions show the largest improvement in memory usage. The *`--no-cache`* option mainly prevents intermediate variables created due to binary operations from caching their values. This results in large memory gains while not effecting run times substantially. The table below is with *`--inline`* switched on and with 102400 elements for each case. The *`--no-cache`* flag is the default option.

Example	time per step `--no-cache` (s)	time per step `--cache` (s)	memory per cell `--no-cache` (KiB)	memory per cell `--cache` (KiB)
`examples.phase.impingement.mesh20x20`	4.3	4.1	1.4	2.3
`examples.phase.anisotropy`	3.5	3.2	1.1	1.9
`examples.cahnHilliard.mesh2D`	3.0	2.5	1.1	1.4
`examples.levelSet.electroChem.simpleTrenchSystem`	62	62	2.0	2.8

10.2 Efficiency discussion of Pysparse and Trilinos

Trilinos provides multigrid capabilities which are beneficial for some problems, but has significant overhead compared to Pysparse. The matrix-building step takes significantly longer in Trilinos, and the solvers also have more overhead costs in memory and performance than the equivalent Pysparse solvers. However, the multigrid preconditioning capabilities of Trilinos can, in some cases, provide enough of a speedup in the solution step to make up for the overhead costs. This depends greatly on the specifics of the problem, but is most likely in the cases when the problem is large and when Pysparse cannot solve the problem with an iterative solver and must use an LU solver, while Trilinos can still have success with an iterative method.

Chapter 11

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 [14], Patankar [15] or Versteeg and Malalasekera [16].

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

11.1 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

$$\frac{\partial \phi}{\partial t} = H(\phi, \lambda_i) \tag{11.1}$$

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,

$$\underbrace{\frac{\partial(\rho\phi)}{\partial t}}_{\text{transient}} + \underbrace{\nabla\cdot(\vec{u}\phi)}_{\text{convection}} = \underbrace{\left[\nabla\cdot(\Gamma_i\nabla)\right]^n\phi}_{\text{diffusion}} + \underbrace{S_\phi}_{\text{source}} \tag{11.2}$$

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

in the equation. The diffusion term can represent any higher order diffusion-like term, where the order is given by the exponent n. For example, the diffusion term can represent conventional Fickian diffusion [*i.e.*, $\nabla \cdot (\Gamma \nabla \phi)$] when the exponent $n = 1$ or a Cahn-Hilliard term [*i.e.*, $\nabla \cdot (\Gamma_1 \nabla [\nabla \cdot \Gamma_2 \nabla \phi)]$] [18] [19] [20]] when $n = 2$, or a phase field crystal term [*i.e.*, $\nabla \cdot (\Gamma_1 \nabla [\nabla \cdot \Gamma_2 \nabla \{\nabla \cdot \Gamma_3 \nabla \phi)\})]$] [21]] when $n = 3$, although spectral methods are probably a better approach. Higher order terms ($n > 3$) are also possible, but the matrix condition number becomes quite poor.

11.2 Finite Volume Method

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

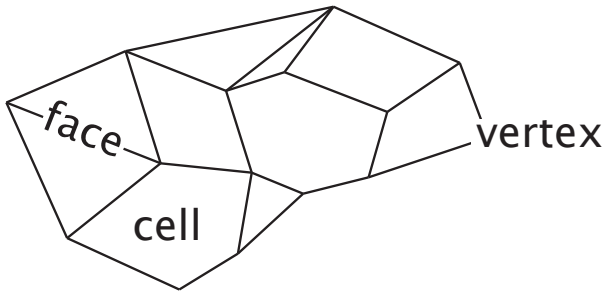


Fig. 1: Mesh
A mesh consists of cells, faces and vertices. For the purposes of *FiPy*, the divider between two cells is known as a face for all dimensions.

11.2.1 Cell Centered FVM (CC-FVM)

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

Boundary Conditions

The natural boundary condition for CC-FVM is no-flux. For (11.2), the boundary condition is

$$\hat{n} \cdot [\vec{u}\phi - (\Gamma_i \nabla)^n] = 0$$

11.2.2 Vertex Centered FVM (VC-FVM)

In the VC-FVM, the CV is centered around the vertices and the cells are divided into sub-control volumes that make up the main CVs (see Figure *CV structure for an unstructured mesh*). The vertices "store" the average variable values over the CVs. The CV faces are constructed within the cells rather than using the cell faces as in the CC-FVM. The face fluxes use all the vertex values from the cell where the face is located to calculate interpolations. For this reason, the VC-FVM is less efficient and requires more storage (a larger matrix band width) than the CC-FVM. However, the mesh topology does not have the same restrictions as the CC-FVM. *FiPy* does not have a VC-FVM capability.

11.3 Discretization

The first step in the discretization of Equation (11.2) 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 (11.2) will be examined separately.

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

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

$$\int_V \frac{\partial(\rho\phi)}{\partial t} dV \simeq \frac{(\rho_P\phi_P - \rho_P^{\text{old}}\phi_P^{\text{old}})V_P}{\Delta t} \tag{11.3}$$

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

This term is represented in *FiPy* as

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

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

The discretization for the convection term is given by

$$\begin{aligned}\int_V \nabla\cdot(\vec{u}\phi)\, dV &= \int_S (\vec{n}\cdot\vec{u})\phi\, dS \\ &\simeq \sum_f (\vec{n}\cdot\vec{u})_f \phi_f A_f\end{aligned} \tag{11.4}$$

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

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

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

This term is represented in *FiPy* as

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

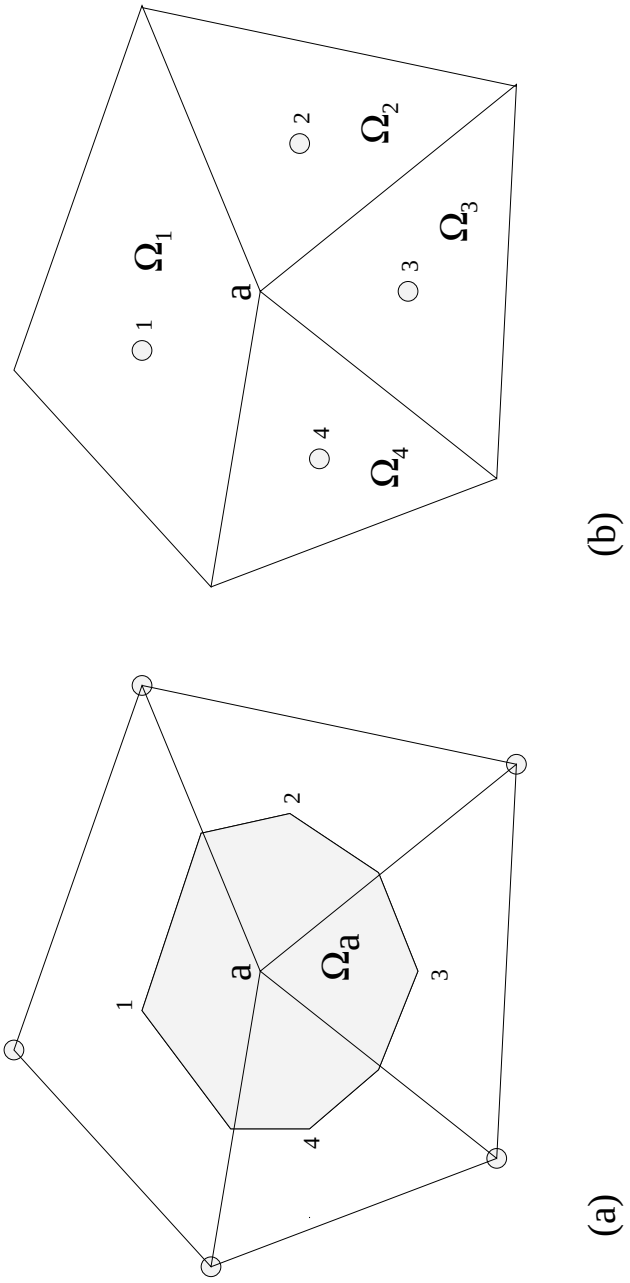


Fig. 2: CV structure for an unstructured mesh
(a) Ω_α represents a vertex-based CV and (b) Ω_1, Ω_2, Ω_3 and Ω_4 represent cell centered CVs.

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

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

The discretization for the diffusion term is given by

$$\begin{aligned} \int_V \nabla \cdot (\Gamma \nabla \{\ldots\}) dV &= \int_S \Gamma (\vec{n} \cdot \nabla \{\ldots\}) dS \\ &\simeq \sum_f \Gamma_f (\vec{n} \cdot \nabla \{\ldots\})_f A_f \end{aligned} \tag{11.5}$$

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

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

where the value of d_{AP} is the distance between neighboring cell centers. This estimate relies on the orthogonality of the mesh, and becomes increasingly inaccurate as the non-orthogonality increases. Correction terms have been derived to improve this error but are not currently included in *FiPy* [14].

This term is represented in *FiPy* as

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

or

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

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

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

Higher Order Diffusion

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

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

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

Note: While this multiple-coefficient form is still supported, *Coupled and Vector Equations* are the recommended approach for higher order expressions.

11.3.4 Source Term

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

$$\int_V S_\phi \, dV \simeq S_\phi V_P. \tag{11.6}$$

Including any negative dependence of S_ϕ on ϕ increases solution stability. The dependence can only be included in a linear manner so Equation (11.6) becomes

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

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

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

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

11.4 Linear Equations

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

Combining Equations (11.3), (11.4), (11.5) and (11.6), the complete discretization for equation (11.2) can now be written for each CV as

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

Equation (11.7) 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

$$a_P\phi_P = \sum_f a_A\phi_A + b_P, \tag{11.7}$$

where

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

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

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

11.5 Numerical Schemes

The coefficients of equation (11.7) 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

$$\frac{1}{1 - \alpha_f} > P_f > -\frac{1}{\alpha_f} \tag{11.8}$$

must be satisfied. The parameter α_f is defined by the chosen scheme, depending on Equation (11.8). The various differencing schemes are:

the central differencing scheme,
where

$$\alpha_f = \frac{1}{2} \tag{11.9}$$

so that $|P_f| < 2$ satisfies Equation (11.8). Thus, the central differencing scheme is only numerically stable for a low values of P_f.

the upwind scheme,
where

$$\alpha_f = \begin{cases} 1 & \text{if } P_f > 0, \\ 0 & \text{if } P_f < 0. \end{cases} \tag{11.10}$$

Equation (11.10) satisfies the inequality in Equation (11.8) for all values of P_f. However the solution over predicts the diffusive term leading to excessive numerical smearing (“false diffusion”).

the exponential scheme,
where

$$\alpha_f = \frac{(P_f - 1) \exp(P_f) + 1}{P_f (\exp(P_f) - 1)}. \tag{11.11}$$

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

the hybrid scheme,
where

$$\alpha_f = \begin{cases} \frac{P_f - 1}{P_f} & \text{if } P_f > 2, \\ \frac{1}{2} & \text{if } |P_f| < 2, \\ -\frac{1}{P_f} & \text{if } P_f < -2. \end{cases} \tag{11.12}$$

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

the power law scheme,
where

$$\alpha_f = \begin{cases} \frac{P_f - 1}{P_f} & \text{if } P_f > 10, \\ \frac{(P_f - 1) + (1 - P_f/10)^5}{P_f} & \text{if } 0 < P_f < 10, \\ \frac{(1 - P_f/10)^5 - 1}{P_f} & \text{if } -10 < P_f < 0, \\ -\frac{1}{P_f} & \text{if } P_f < -10. \end{cases} \tag{11.13}$$

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

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

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

Chapter 12

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.

12.1 Design

12.1.1 Numerical Approach

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

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

12.1.2 Object Oriented Structure

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

12.1.3 Test Based Development

FiPy has been developed with a large number of test cases. These test cases are in two categories. The lower level tests operate on the core modules at the individual method level. The aim is that every method within the core installation has a test case. The high level test cases operate in conjunction with example solutions and serve to test global solution algorithms and the interaction of various modules.

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

12.1.4 Open Source

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

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

12.1.5 High-Level Scripting Language

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

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

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

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

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

(continues on next page)

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

(continued from previous page)

```
            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[2].

[2] A discussion of efficiency issues can be found in *Efficiency*.

12.1.6 Python Programming Language

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

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

12.2 Implementation

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

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

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

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

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

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

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

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

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

A `Term` object
: represents any of the terms in Equation (11.2) 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 (11.2), 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 a `DiffusionTerm` or a linear combination of other `Term` objects that build up to form an expression such as Equation (11.2).

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

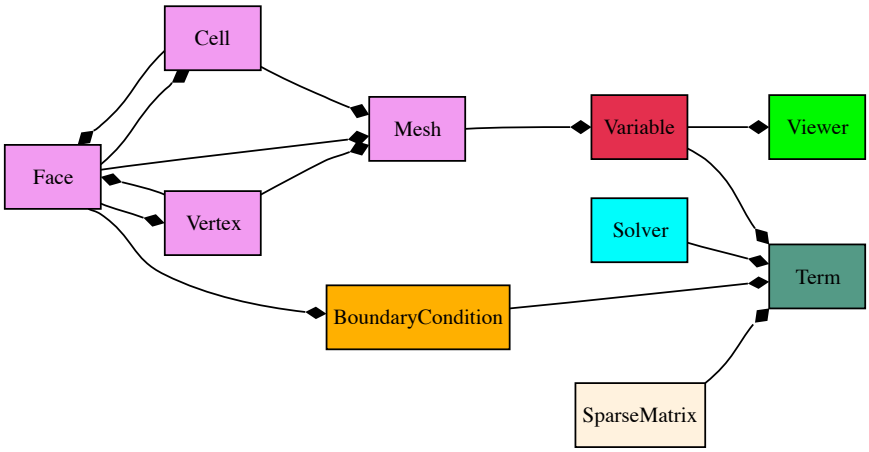


Fig. 1: Primary object relationships in *FiPy*.

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

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

Chapter 13

Virtual Kinetics of Materials Laboratory

The VKML is a set of simple *FiPy* examples that simulate basic aspects of kinetics of materials through an interactive Graphical User Interface. The seminal development by Michael Waters and Prof. R. Edwin Garcia of Purdue University includes four examples:

Polycrystalline Growth and Coarsening

simulates the growth, impingement, and coarsening of a random distribution of crystallographically oriented nuclei. The user can control every aspect of the model such as the nuclei radius, the size of the simulation cell, and whether the grains are homogeneously dispersed or only on one wall of the simulation.

Dendritic Growth

simulates the anisotropic solidification of a single solid seed with an N-fold axis of crystallographic symmetry embedded in an undercooled liquid. The user can specify many material aspects of the solidification process, such as the thermal diffusivity and the strength of the surface tension anisotropy. Default values are physical but arbitrary. This model is based on the phase field method and an example shown in the *FiPy* manual.

Two-Dimensional Spinodal Decomposition

simulates the time-dependent segregation of two chemical components and its subsequent coarsening, as presented by John Cahn. The default values are physical but arbitrary.

Three-Dimensional Spinodal Decomposition

has the same functionality as the 2D version, but has an interactive Three-Dimensional viewer.

These modules provide a Graphical User Interface to *FiPy*, and allow you to perform simulations directly through your web browser. This approach to computing removes the need to install the software on your local machine (unless you really want to), allows you to assess current and potential *FiPy* applications and instead you only need a web browser to access it and run it. In other words, you can run these simulations (and simulations like this one) from a Windows machine, a Mac, or a Linux box, and you can also run the modules from Michigan, Boston, Japan, or England: from wherever you are. Moreover, if you close your web browser and leave your calculation running, when you come back a few hours later, your calculation will persist. Additionally, if there is something you want to share with a coworker, wherever he or she might be (e.g., the other side of the planet), you can grant him temporary access to your calculation so that the third party can directly see the output (or specify inputs directly into it, without having to travel to where you are). It is a great way to privately (or publicly) collaborate with other people, especially if the users are in different parts of the world.

The only requirement to run VKML is to register (registration is 100% free) in the nanoHUB.

Chapter 14

Contributors

Jon Guyer
is a member of the research staff of the Materials Science and Engineering Division in the Material Measurement 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 guest researcher in the Materials Science and Engineering Division in the Material Measurement Laboratory at the National Institute of Standards and Technology. Daniel's interests are in numerical modeling, finite volume techniques, and level set treatments.

Jim Warren
is the leader of the Thermodynamics and Kinetics group in the Materials Science and Engineering Division and Director of the Center for Theoretical and Computational Materials Science of the Material Measurement 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 anisotropic diffusion coefficients while he was on sabbatical from the University of Maine.

Olivia Buzek
worked on adding *Trilinos* parallel computations while she was a SURF student from the University of Maryland

Daniel Stiles
worked on adding *Trilinos* parallel computations while he was a student at Montgomery Blair High School.

James O'Beirne
added full mesh partitioning using *Gmsh*. James also greatly improved the *Gmsh*-*FiPy* pipeline. Other contributions include updating *FiPy* to use properties pervasively, deployment of a *Buildbot* server to automate *FiPy* testing and a full refactor of the `Mesh` classes.

Chapter 15

Publications

Attention: If you use FiPy in your research, please cite:

J. E. Guyer, D. Wheeler & J. A. Warren, "FiPy: Partial Differential Equations with Python," *Computing in Science & Engineering* **11** (3) pp. 6-15 (2009), doi:10.1109/MCSE.2009.52. (pdf)

Other publications that have used FiPy. Please contact us to add your work to this list.

- D. Wheeler, and J. A. Warren & W. J. Boettinger, "Modeling the early stages of reactive wetting" *Physical Review E* **82** (5) pp. 051601 (2010), doi:10.1103/PhysRevE.82.051601.
- R. R. Mohanty, J. E. Guyer & Y. H. Sohn, "Diffusion under temperature gradient: A phase-field model study" *Journal of Applied Physics* **106** (3) pp. 034912 (2009), doi:10.1063/1.3190607.
- J. A. Warren, T. Pusztai, L. Környei & L. Gránásy, "Phase field approach to heterogeneous crystal nucleation in alloys," *Physical Review B* **79** 014204 (2009), doi:10.1103/PhysRevB.79.014204.
- T. P. Moffat, D. Wheeler, S.-K. Kim & D. Josell, "Curvature enhanced adsorbate coverage mechanism for bottom-up superfilling and bump control in Damascene processing," *Electrochimica Acta* **53** (1) pp. 145-154 (2007), doi:10.1016/j.electacta.2007.03.025.
- W. J. Boettinger, J. E. Guyer, C. E. Campbell, G. B. McFadden, "Computation of the Kirkendall velocity and displacement fields in a one-dimensional binary diffusion couple with a moving interface," *Proceedings of the Royal Society A: Mathematical, Physical & Engineering Sciences* **463** (2088) pp. 3347-3373 (2007), doi:10.1098/rspa.2007.1904.
- T. Cickovski, K. Aras, M. Swat, R. M. H. Merks, T. Glimm, H. G. E. Hentschel, M. S. Alber, J. A. Glazier, S. A. Newman & J. A. Izaguirre, "From Genes to Organisms Via the Cell: A Problem-Solving Environment for Multicellular Development," *Computing in Science & Engineering* **9** (4) pp. 50-60 (2007), doi:10.1109/MCSE.2007.74.
- L. Gránásy, T. Pusztai, D. Saylor & J. A. Warren, "Phase Field Theory of Heterogeneous Crystal Nucleation," *Physical Review Letters* **98** 035703 (2007) 10.1103/PhysRevLett.98.035703.
- J. Mazur, "Numerical Simulation of Temperature Field in Soil Generated by Solar Radiation," *Journal de Physique IV France* **137** pp. 317-320 (2006), doi:10.1051/jp4:2006137061.
- T. P. Moffat, D. Wheeler, S. K. Kim & D. Josell, "Curvature enhanced adsorbate coverage model for electrodeposition," *Journal of The Electrochemical Society* **153** (2) pp. C127-C132 (2006), 10.1149/1.2165580.

- D. Josell, D. Wheeler & T. P. Moffat, “Gold superfill in submicrometer trenches: Experiment and prediction,” *Journal of The Electrochemical Society* **153** (1) pp. C11-C18 (2006), 10.1149/1.2128765.

Chapter 16

Presentations

We were honored to be invited to deliver a keynote presentation on "Modeling of Materials with Python" at the 2009 Python for Scientific Computing Conference at Caltech, August 2009.

Other invited talks about FiPy:

- "FiPy: An Open Source Finite Volume PDE Solver Implemented in Python" by J. E. Guyer at the George Mason University Department of Mathematical Sciences, October 2009.
- "FiPy: An Open-Source PDE Solver for Materials Science" by J. E. Guyer at the Center for Devices and Radiological Health of the Food and Drug Administration, June 2009.
- "FiPy: An Open-Source PDE Solver for Materials Science" by J. E. Guyer at GE Global Research, June 2009.
- "FiPy: A PDE Solver for Materials Science" by J. E. Guyer at the SIAM Conference on Computational Science and Engineering, March 2009.
- "FiPy: An Open Source Finite Volume PDE Solver Implemented in Python" by J. E. Guyer in the Open Source Tools for Materials Research and Engineering session of the TMS 2009 Annual Meeting, February 2009.
- "FiPy: A Finite Volume PDE Solver Implemented in Python" by J. E. Guyer in the Computational Materials Research and Education Luncheon Roundtable of the TMS Annual Meeting, February 2009.
- "FiPy - An Object-Oriented Tool for Phase Transformation Simulations Using Python" by J. E. Guyer at Microstructology III, Birmingham, AL, May 2005.
- "FiPy - An Object-Oriented Tool for Phase Transformation Simulations Using Python" by J. E. Guyer at the 2004 MRS Fall Meeting, November 2004.

Chapter 17

Change Log

17.1 Version 0+unknown

This maintenance release adds *Logging* and resolves compatibility issues with recent builds of *PETSc* and *NumPy*.

17.1.1 Pulls

- Fix numpy 1.25 issues (#930)
- Get CI working again (#925)
- Discourage StackOverflow (#876)
- Add Logging (#875)
- Add tests for the Nix build (#791)

17.1.2 Fixes

- #896: Poor garbage collection with petsc4py 3.18.3 (was "Memory leak in *term.justErrorVector()*", but this isn't strictly a leak)

17.2 Version 3.4.3 - 2022-06-15

This maintenance release adds a new example contributed by @Jon83Carvalho, clarifies many points in the documentation, migrates all *Continuous Integration* to Azure, updates to using wheels for distribution, and substantially refactors matrices to work more consistently across solvers.

17.2.1 Pulls

- Update CI documentation to refer only to Azure (#863)
- Refine azure runs (#851)
- Debug CIs (#848)
- Collect contact information on single page (#847)
- Set up CI with Azure Pipelines (#822)
- Replace deprecated numpy types (#798)
- Move trilinos tests to Py3k (#797)
- Fix Python 2.7 conda environment (#795)
- fix: stop divide by zero warning in LU solvers (#790)
- Introduce *SharedTemporaryFile* (bis) (#769)
- Raise *ImportError* before trying to unpack solvers (#768)
- Disable TVTK tests if its prerequisites aren't met (#764)
- Tabulate versions of FiPy dependencies when tests are run (#763)
- Debug CI failures (#749)
- Stokes Cavity - non-Newtonian (#748) Thanks to @Jon83Carvalho.
- Refactor matrices (#721)

17.2.2 Fixes

- #862: Could not load the Qt platform plugin "*xcb*"
- #858: CI issues
- #856: *FaceVariable* does not accumulate properly in parallel
- #850: Switch to wheels
- #849: *linux-py27-pysparse* fails
- #841: *Matplotlib2DViewer* should accept color map as string
- #836: Document that coupled and high-order diffusion terms are incompatible
- #833: *fipy.tools.dump* undocumented that it always gzips
- #828: *colorbar=True* no longer works Stokes flow example
- #826: Gmsh load issue
- #818: Document that *GridND* meshes are always Cartesian
- #811: In python 3.9 __repr__ throws an exception with abs
- #801: CircleCI test-36-trilinos-serial extremely slow
- #800: CircleCI conda2_env is really slow and ends up installing FiPy 3.3
- #796: *examples.phase.polyxtal* freezes on CircleCI with Py3k and scipy solvers
- #792: *circleQuad* example fails with Gmsh > 4.4

- #781: *MatplolibViewer.axes* property is not documented
- #778: Binder failed build
- #762: Equations on Website don't show right
- #742: No documentation for *Variable.mag*
- #735: *pip install fipy* fails
- #734: Document the residual
- #688: try-except not needed for circle Viewer
- #676: Default no-flux condition is not explicitly stated
- #609: Parallelizing of Gmsh meshes not clearly documented
- #400: Fix *FaceVariable.globalValue* method

17.3 Version 3.4.2.1 - 2020-08-01

This release fixes assorted viewer issues, fixes a problem with convection boundary conditions, and introduces spherical meshes.

Attention: There are known failures with the VTK viewers (bitrot has started to set in since the demise of Python 2.7). There's also a new parallel failure in *NonUniformGrid1D* that we need to figure out.

17.3.1 Pulls

- Move mailing list (#747)
- *Spherical1D* (*Uniform* and *NonUniform*) meshes (#732) Thanks to @klkuhlm.
- fix Neumann BCs using constraints with convection terms (#719) Thanks to @atismer.
- Add vertex index inversions (#716)

17.3.2 Fixes

- #726: *MayaviClient* not compatible with Python 3
- #663: *datamin*/*datamax* argument ignored by viewer
- #662: Issues Scaling *Colorbar* with *Datamin* and *Datamax Args*

17.4 Version 3.4.1 - 2020-02-14

This release is primarily for compatibility with `numpy` 1.18.

17.4.1 Pulls

- Fix documentation (#711)
- build(nix): fix broken plm_rsh_agent error (#710)
- CIs error on deprecation warning (#708)

17.4.2 Fixes

- #703: FORTRAN array ordering is deprecated

17.5 Version 3.4 - 2020-02-06

This release adds support for the *PETSc* solvers for *Solving in Parallel*.

17.5.1 Pulls

- Add support for PETSc solvers (#701)
- Assorted fixes while supporting PETSc (#700) - Fix print statements for Py3k - Resolve Gmsh issues - Dump only on processor 0 - Only write *timetests* on processor 0 - Fix conda-forge link - Upload PDF - Document *print* option of *FIPY_DISPLAY_MATRIX* - Use legacy numpy formatting when testing individual modules - Switch to matplotlib's built-in symlog scaling - Clean up tests
- Assorted fixes for benchmark 8 (#699) - Stipulate *–force* option for *conda remove fipy* - Update Miniconda installation url - Replace *_CellVolumeAverageVariable* class with *Variable* expression - Fix output for bad call stack
- Make CircleCI build docs on Py3k (#698)
- Fix link to Nick Croft's thesis (#681)
- Fix NIST header footer (#680)
- Use Nixpkgs version of FiPy expression (#661)
- Update the Nix recipe (#658)

17.5.2 Fixes

- #692: Can't copy example scripts with the command line
- #669: input() deadlock on parallel runs
- #643: Automate release process

17.6 Version 3.3 - 2019-06-28

This release brings support for Python 2 and Python 3 from the same source, without any translation. Thanks to @pya and @woodscn for getting things started.

17.6.1 Pulls

- Automate spell check (#657)
- Fix gmsh on windows (#648)
- Fix sphinx documentation (#647)
- Migrate to Py3k (#645)
- *gmshMesh.py* compatibility with Gmsh > 3.0.6 (#644) Thanks to @xfong.

17.6.2 Fixes

- #655: When Python 2 and 3 are installed, Mayavi wont work. Thanks to @Hendrik410.
- #646: Deprecate develop branch
- #643: Automate release process
- #601: `contents.rst` and `manual.rst` are a recursive mess
- #597: Use GitHub link for the compressed archive in documentation
- #557: *faceGradAverage* is stupid
- #552: documentation integration
- #458: Documentation wrong for precedence of *Lx* and *dx* for *NonUniformGrids*
- #457: Special methods are not included in Sphinx documentation
- #432: Python 3 issues
- #340: Don't upload packages to PyPI, just add the master url

17.7 Version 3.2 - 2019-04-22

This is predominantly a DevOps release. The focus has been on making FiPy easier to install with *conda*. It's also possible to install a minimal set of prerequisites with *pip*. Further, *FiPy* is automatically tested on all major platforms using cloud-based *Continuous Integration* (*linux* with *CircleCI*, *macOS* with *TravisCI*, and *Windows* with *AppVeyor*).

17.7.1 Pulls

- Make badges work in GitHub and pdf (#636)
- Fix Robin errors (#615)
- Issue555 inclusive license (#613)
- Update CIs (#607)
- Add CHANGELOG and tool to generate from issues and pull requests (#600)
- Explain where to get examples (#596)
- spelling corrections using en_US dictionary (#594)
- Remove *SmoothedAggregationSolver* (#593)
- Nix recipe for FiPy (#585)
- Point PyPI to github master tarball (#582)
- Revise Navier-Stokes expression in the viscous limit (#580)
- Update *stokesCavity.py* (#579) Thanks to @Rowin.
- Add *–inline* to TravisCI tests (#578)
- Add support for binder (#577)
- Fix *epetra vector not numarray* (#574)
- add Codacy badge (#572)
- Fix output when PyTrilinos or PyTrilinos version is unavailable (#570) Thanks to @shwina.
- Fix check for PyTrilinos (#569) Thanks to @shwina.
- Adding support for GPU solvers via pyamgx (#567) Thanks to @shwina.
- revise dedication to the public domain (#556)
- Fix tests that don't work in parallel (#550)
- add badges to index and readme (#546)
- Ensure vector is *dtype* float before matrix multiply (#544)
- Revert "Issue534 physical field mishandles compound units" (#536)
- Document boundary conditions (#532)
- Deadlocks and races (#524)
- Make max/min global (#520)
- Add a Gitter chat badge to `README.rst` (#516) Thanks to @gitter-badger.
- Add TravisCI build recipe (#489)

17.7.2 Fixes

- #631: Clean up `INSTALLATION.rst`
- #628: Problems with the viewer
- #627: Document OMP_NUM_THREADS
- #625: *setup.py* should not import fipy
- #623: Start using *versioneer*
- #621: Plot *FaceVariable* with matplotlib
- #617: Pick 1st Value and last Value of 1D *CellVariable* while running in parallel
- #611: The coefficient cannot be a *FaceVariable* ??
- #610: Anisotropy example: Contour plot displaying in legend of figure !?
- #608: *var.mesh*: *Property* object not callable…?
- #603: Can't run basic test or examples
- #602: Revise build and release documentation
- #592: is `resources.rst` useful?
- #590: No module named *pyAMGSolver*
- #584: Viewers don't animate in jupyter notebook
- #566: Support for GPU solvers using pyamgx
- #565: pip install does not work on empty env
- #564: Get green boxes across the board
- #561: Cannot cast array data from *dtype('int64')* to *dtype('int32')* according to the rule *safe*
- #555: inclusive license
- #551: Sphinx spews many warnings:
- #545: Many Py3k failures
- #543: Epetra Vector can't be integer
- #539: *examples/diffusion/explicit/mixedElement.py* is a mess
- #538: badges
- #534: *PhysicalField* mishandles compound units
- #533: pip or conda installation don't make clear where to get examples
- #531: *drop_tol* argument to *scipy.sparse.linalg.splu* is gone
- #530: conda installation instructions not explicit about python version
- #528: scipy 1.0.0 incompatibilities
- #525: conda *guyer/pysparse* doesn't run on osx
- #513: Stokes example gives wrong equation
- #510: Weave, Scipy and *–inline*
- #509: Unable to use conda for installing FiPy in Windows
- #506: Error using spatially varying anisotropic diffusion coefficient

- #488: Gmsh 2.11 breaks *GmshGrids*
- #435: *pip install pysparse* fails with "*fatal error: 'spmatrix.h' file not found*"
- #434: *pip install fipy* fails with "*ImportError: No module named ez_setup*"

17.8 Version 3.1.3 - 2017-01-17

17.8.1 Fixes

- #502: gmane is defunct

17.9 Version 3.1.2 - 2016-12-24

17.9.1 Pulls

- remove *recvobj* from calls to *allgather*, require *sendobj* (#492)
- restore trailing whitespace to expected output of pysparse matrix tests (#485)
- Format version string for pep 440 (#483)
- Provide some documentation for what *_faceToCellDistanceRatio* is and why it's scalar (#481)
- Strip all trailing white spaces and empty lines at EOF for *.py* and *.r*? (#479) Thanks to @pya.
- *fipy/meshes/uniformGrid3D.py*: fix *_cellToCellIDs* and more *concatenate()* calls (#478) Thanks to @pkgw.
- Remove incorrect *axis* argument to *concatenate* (#477)
- Updated to NumPy 1.10 (#472) Thanks to @pya.
- Some spelling corrections (#471) Thanks to @pkgw.
- Sort entry points by package name before testing. (#469)
- Update import syntax in examples (#466)
- Update links to prerequisites (#465)
- Correct implementation of *examples.cahnHilliard.mesh2DCoupled*. Fixes ? (#463)
- Fix typeset analytical solution (#460)
- Clear *pdflatex* build errors by removing *Python* from heading (#459)
- purge gist from viewers and optional module lists in *setup.py* (#456)
- Remove deprecated methods that duplicate NumPy ufuncs (#454)
- Remove deprecated Gmsh importers (#452)
- Remove deprecated getters and setters (#450)
- Update links for FiPy developers (#448)
- Render appropriately if in IPython notebook (#447)
- Plot contour in proper axes (#446)
- Robust Gmsh version checking with *distutils.version.StrictVersion* (#442)

- compare gmsh versions as tuples, not floats (#441)
- Corrected two tests (#439) Thanks to @alfrenardi.
- Issue426 fix robin example typo (#431) Thanks to @raybsmith.
- Issue426 fix robin example analytical solution (#429) Thanks to @raybsmith.
- Force *MatplotlibViewer* to display (#428)
- Allow for 2 periodic axes in 3D (#424)
- Bug with Matplotlib 1.4.0 is fixed (#419)

17.9.2 Fixes

- #498: nonlinear source term
- #496: *scipy.LinearBicgstabSolver* doesn't take arguments
- #494: Gmsh call errors
- #493: *Reviewable.io* has read-only access, can't leave comments
- #491: *globalValue* raises error from mpi4py
- #484: Pysparse tests fail
- #482: FiPy development version string not compliant with PEP 440
- #476: *setuptools* 18.4 breaks test suite
- #475: *Grid3D* broken by numpy 1.10
- #470: *Mesh3D cellToCellIDs* is broken
- #467: Out-of-sequence Viewer imports
- #462: GMSH version >= 2.10 incorrectly read by *gmshMesh.py*
- #455: *setup.py* gist warning
- #445: *DendriteViewer* puts contours over color bar
- #443: *MatplotlibViewer* still has problems in IPython notebook
- #440: Use github API to get nicely formatted list of issues
- #438: Failed tests on Mac OS X
- #437: Figure misleading in *examples.cahnHilliard.mesh2DCoupled*
- #433: Links to prerequisites are broken
- #430: Make develop the default branch on Github
- #427: *MatplotlibViewer* don't display
- #425: Links for Warren and Guyer are broken on the web page
- #421: The "limits" argument for *Matplotlib2DGridViewer* does not function
- #416: Updates to reflect move to Github

17.10 Version 3.1.1 - 2015-12-17

17.10.1 Fixes

- #415: *MatplotlibGrid2DViewer* error with Matplotlib version 1.4.0
- #414: *PeriodicGrid3D* supports Only 1 axes of periodicity or all 3, not 2
- #413: Remind users of different types of conservation equations
- #412: Pickling Communicators is unnecessary for Grids
- #408: Implement *PeriodicGrid3D*
- #407: Strange deprecation loop in reshape()
- #404: package never gets uploaded to PyPI
- #401: Vector equations are broken when *sweep* is used instead of *solve*.
- #295: Gmsh version must be >= 2.0 errors on *zizou*

17.11 Version 3.1 - 2013-09-30

The significant changes since version 3.0 are:

- Level sets are now handled by *LSMLIB* or *Scikit-fmm* solver libraries. These libraries are orders of magnitude faster than the original, *Python*-only prototype.
- The *Matplotlib* `streamplot()` function can be used to display vector fields.
- Version control was switched to the Git distributed version control system. This system should make it much easier for *FiPy* users to participate in development.

17.11.1 Fixes

- #398: Home page needs out-of-NIST redirects
- #397: Switch to *sphinxcontrib-bibtex*
- #396: enable google analytics
- #395: Documentation change for Ubuntu install
- #393: *CylindricalNonUniformGrid2D* doesn't make a *FaceVariable* for *exteriorFaces*
- #392: *exit_nist.cgi* deprecated
- #391: Péclet inequalities have the wrong sign
- #388: Windows 64 and numpy's *dtype=int*
- #384: Add support for Matplotlib *streamplot*
- #382: Neumann boundary conditions not clearly documented
- #381: numpy 1.7.1 test failures with *physicalField.py*
- #377: *VanLeerConvectionTerm* MinMod slope limiter is broken
- #376: testing *CommitTicketUpdater*

- #375: NumPy 1.7.0 doesn't have *_formatInteger*
- #373: Bug with numpy 1.7.0
- #372: convection problem with cylindrical grid
- #371: *examples/phase/binary.py* has problems
- #370: FIPY_DISPLAY_MATRIX is broken
- #368: Viewers don't inline well in IPython notebook
- #367: Change documentation to promote use of stackoverflow
- #366: *unOps* can't be pickled
- #365: Rename communicator instances
- #364: Parallel bug in non-uniform grids and conflicting mesh class and factory function names
- #360: NIST CSS changed
- #356: link to mailing list is wrong
- #353: Update Ohloh to point at git repo
- #352: *getVersion()* fails on Py3k
- #350: Gmsh importer can't read mesh elements with no tags
- #347: Include mailing list activity frame on front page
- #339: Fix for test failures on *loki*
- #337: Clean up interaction between dependencies and installation process
- #336: *fipy.test()* and *fipy/test.py* clash
- #334: Make the citation links go to the DOI links
- #333: Web page links seem to be broken
- #331: Assorted errors
- #330: *faceValue* as *FaceCenters* gives inline failures
- #329: Gmsh background mesh doesn't work in parallel
- #326: *Gmsh2D* does not respect background mesh
- #323: *getFaceCenters()* should return a *FaceVariable*
- #319: Explicit convection terms should fail when the equation has no *TransientTerm (dt=None)*
- #318: FiPy will not import
- #311: LSMLIB refactor
- #305: *mpirun -np 2 python -Wd setup.py test –trilinos* hanging on sandbox under buildbot
- #297: Remove deprecated gist and gnuplot support
- #291: efficiency_test chokes on *liquidVapor2D.py*
- #289: *diffusionTerm._test()* requires Pysparse
- #287: move FiPy to distributed version control
- #275: *mpirun -np 2 python setup.py test –no-pysparse* hangs on *bunter*
- #274: Epetra *Norm2* failure in parallel

- #272: Error adding meshes
- #269: Rename *GridXD*
- #255: numpy 1.5.1 and masked arrays
- #253: Move the mail archive link to a more prominent place on web page.
- #245: Fix *fipy.terms._BinaryTerm* test failure in parallel
- #228: *–pysparse* configuration should never attempt MPI imports
- #225: Windows interactive plotting mostly broken
- #209: add Rhie-Chow correction term in stokes cavity example
- #180: broken arithmetic face to cell distance calculations
- #128: Trying to "solve" an integer *CellVariable* should raise an error
- #123: *numerix.dot* doesn't support tensors
- #103: *subscriber()._markStale() AttributeError*
- #61: Move *ImplicitDiffusionTerm().solve(var) == 0* "failure" from *examples.phase.simple* to *examples.diffusion.mesh1D*?

17.12 Version 3.0.1 - 2012-10-03

17.12.1 Fixes

- #346: text in *trunk/examples/convection/source.py* is out of date
- #342: sign issues for equation with transient, convection and implicit terms
- #338: SvnToGit clean up

17.13 Version 3.0 - 2012-08-16

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

The significant changes since version 2.1 are:

- *Coupled and Vector Equations* are now supported.
- A more robust mechanism for specifying *Boundary Conditions* is now used.
- Most `Mesh`es can be partitioned by *Meshing with Gmsh*.
- *PyAMG* and *SciPy* have been added to the *Solvers*.
- FiPy is capable of running under *Python 3*.
- "getter" and "setter" methods have been pervasively changed to Python properties.
- The test suite now runs much faster.
- Tests can now be run on a full install using *fipy.test()*.
- The functions of the `numerix` module are no longer included in the `fipy` namespace. See `examples.updating.update2_0to3_0` for details.

- Equations containing a `TransientTerm`, must specify the timestep by passing a `dt=` argument when calling `solve()` or `sweep()`.

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

17.13.1 Fixes

- #332: Inline failure on Ubuntu x86_64
- #324: constraining values with *ImplicitSourceTerm* not documented?
- #317: *gmshImport* tests fail on Windows due to shared file
- #316: changes to *gmshImport.py* caused *–inline* problems
- #313: Gmsh I/O
- #307: Failures on sandbox under buildbot
- #306: Add in parallel buildbot testing on more than 2 processors
- #302: *CellVariable.min()* broken in parallel
- #301: *Epetra.PyComm()* broken on Debian
- #300: *examples/cahnHilliard/mesh2D.py* broken with – trilinos
- #299: Viewers not working when plotting meshes with zero cells in parallel
- #298: Memory consumption growth with repeated meshing, especially with Gmsh
- #294: *–pysparse –inline* failures
- #293: *python examples/cahnHilliard/sphere.py –inline* segfaults on OS X
- #292: two *–scipy* failures
- #290: Improve test reporting to avoid inconsequential buildbot failures
- #288: gmsh importer and gmsh tests don't clean up after themselves
- #286: get running in Py3k
- #285: remove deprecated *viewers.make()*
- #284: remove deprecated *Variable.transpose()*
- #281: remove deprecated *NthOrderDiffusionTerm*
- #280: remove deprecated *diffusionTerm=* argument to *ConvectionTerm*
- #277: remove deprecated *steps=* from Solver
- #273: Make *DiffusionTermNoCorrection* the default
- #270: tests take *too* long!!!
- #267: Reduce the run times for chemotaxis tests
- #264: HANG in parallel test of *examples/chemotaxis/input2D.py* on some configurations
- #261: *GmshImport* should read element colors
- #260: *GmshImport* should support all element types

- #259: Introduce *mesh.x* as shorthand for *mesh.cellCenters[0]* etc
- #258: *GmshExport* is not tested and does not work
- #252: Include Benny's improved interpolation patch
- #250: TeX is wrong in *examples.phase.quaternary*
- #247: *diffusionTerm(var=var1).solver(var=var0)* should fail sensibly
- #243: close out reconstrain branch
- #242: update documentation
- #240: Profile and merge reconstrain branch
- #237: *–Trilinos –no-pysparse* uses Pysparse?!?
- #236: anisotropic diffusion and constraints don't mix
- #235: changed constraints don't propagate
- #231: *factoryMeshes.py* not up to date with respect to keyword arguments
- #223: mesh in FiPy name space
- #218: Absence of *enthought.tvtk* causes test failures
- #216: Fresh FiPy gives "*ImportError: No viewers found*"
- #213: PyPI is failing
- #206: *gnuplot1d* gives error on plot of *FaceVariable*
- #205: wrong cell to cell normal in periodic meshes
- #203: Give helpful error on - or / of meshes
- #202: mesh manipulation of periodic meshes leads to errors
- #201: Use physical velocity in the manual/FAQ
- #200: FAQ gives bad guidance for anisotropic diffusion
- #195: term multiplication changes result
- #163: Default time steps should be infinite
- #162: remove ones and zeros from *numerix.py*
- #130: tests should be run with *fipy.tests()*
- #86: Grids should take *Lx*, *Ly*, *Lz* arguments
- #77: *CellVariable.hasOld()* should set *self.old*
- #44: Navier-Stokes

17.14 Version 2.1.3 - 2012-01-17

17.14.1 Fixes

- #282: remove deprecated getters and setters
- #279: remove deprecated *fipy.meshes.numMesh* submodule
- #278: remove deprecated forms of Gmsh meshes
- #268: Set up *Zizou* as a working slave
- #262: issue with solvers
- #256: *Grid1D(dx=(1,2,3))* failure
- #251: parallel is broken
- #241: Set Sandbox up as a working slave
- #238: *_BinaryTerm.var* is not predictable
- #233: coupled convection-diffusion always treated as Upwind
- #224: "matrices are not aligned" errors in example test suite
- #222: Non-uniform *Grid3D* fails to __add__
- #221: Problem with fipy and gmsh
- #219: matforge css is hammer-headed
- #208: numpy 2.0: *arrays have a dot method*
- #207: numpy 2.0: *masked arrays cast right of product to ndarray*
- #196: Pysparse won't import in Python 2.6.5 on Windows
- #152: (Re)Implement SciPy solvers
- #138: FAQ on boundary conditions
- #100: testing from the Windows dist using the ipython command line
- #80: Windows - testing - idle *-ipython*
- #46: Variable needs to consider boundary conditions
- #45: Slicing a vector Variable should produce a scalar Variable

17.15 Version 2.1.2 - 2011-04-20

The significant changes since version 2.1.1 are:

- *Trilinos* efficiency improvements
- Diagnostics of the parallel environment

17.15.1 Fixes

- #232: Mayavi broken on windows because it has no *SIGHUP*.
- #230: *factoryMeshes.py* not up to date with respect to keyword arguments
- #226: *MatplotlibViewer* fails if backend doesn't support *flush_events()*
- #225: Windows interactive plotting mostly broken
- #217: Gmsh *CellVariables* can't be unpickled
- #191: *sphereDaemon.py* missing in FiPy 2.1 and from trunk
- #187: Concatenated *Mesh* garbled by *dump.write/read*

17.16 Version 2.1.1 - 2010-10-05

The significant changes since version 2.1 are:

- `MatplotlibViewer` can display into an existing set of Matplotlib axes.
- *Pysparse* and *Trilinos* are now completely independent.

17.16.1 Fixes

- #199: dummy viewer results in "*NotImplementedError: can't instantiate abstract base class*"
- #198: bug problem with *CylindricalGrid1D*
- #197: How to tell if parallel is configured properly?
- #194: *FIPY_DISPLAY_MATRIX* on empty matrix with large b-vector throws *ValueError*
- #193: *FIPY_DISPLAY_MATRIX* raises *ImportError* in FiPy 2.1 and trunk
- #192: *FIPY_DISPLAY_MATRIX=terms* raises *TypeError* in FiPy 2.1 and trunk

17.17 Version 2.1 - 2010-04-01

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

The significant changes since version 2.0.2 are:

- *FiPy* can use *Trilinos* for *Solving in Parallel*.
- We have switched from *MayaVi* 1 to *Mayavi* 2. This `Viewer` is an independent process that allows interaction with the display while a simulation is running.
- Documentation has been switched to *Sphinx*, allowing the entire manual to be available on the web and for our documentation to link to the documentation for packages such as `numpy`, `scipy`, `matplotlib`, and for *Python* itself.

[1] Only two examples from *FiPy* 2.0 fail when run with *FiPy* 2.1:

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

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

17.17.1 Fixes

- #190: "matplotlib: list index out of range" when no title given, but only sometimes
- #182: *~binOp* doesn't work on branches/version-2_0
- #180: broken arithmetic face to cell distance calculations
- #179: *easy_install* instructions for Mac OS X are broken
- #177: broken *setuptools* url with python 2.6
- #169: The FiPy webpage seems to be broken on Internet Explorer
- #156: update the mayavi viewer to use mayavi 2
- #153: Switch documentation to use *:math:* directive

17.18 Version 2.0.3 - 2010-03-17

17.18.1 Fixes

- #188: *SMTPSenderRefused: (553, "5.1.8 <trac@matdl-osi.org>... Domain of sender address trac@matdl-osi.org does not exist", u'"FiPy" <trac@matdl-osi.org>')*
- #184: *gmshExport.exportAsMesh()* doesn't work
- #183: FiPy 2.0.2 *LinearJORSolver.__init__* calls *Solver* rather than *PysparseSolver*
- #181: Navier-Stokes again
- #151: update mayavi viewer to use mayavi2
- #13: Mesh refactor

17.19 Version 2.0.2 - 2009-06-11

17.19.1 Fixes

- #176: Win32 distribution test error
- #175: *Grid3D getFaceCenters* incorrect when mesh is offset
- #170: *Variable* doesn't implement *__invert__*

17.20 Version 2.0.1 - 2009-04-23

17.20.1 Fixes

- #154: Update manuals

17.21 Version 2.0 - 2009-02-09

Warning: *FiPy* 2 brings unavoidable syntax changes. Please see `examples.updating.update1_0to2_0` 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 `Cell`s for initial conditions and `Face`s for boundary conditions.
- Automated determination of the Péclet 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.
- #167: example showing how to go from 1.2 to 2.0
- #166: Still references to *VectorCell* and *VectorFace Variable* in manual
- #165: Edit the what's new section of the manual
- #149: Test viewers
- #143: Document syntax changes
- #141: enthought toolset?
- #140: easy_install fipy
- #136: Document anisotropic diffusion
- #135: Trilinos documentation
- #127: Examples can be very fragile with respect to floating point

17.22 Version 1.2.3 - 2009-01-0

17.22.1 Fixes

- #54: *python setup.py test* fails

17.23 Version 1.2.2 - 2008-12-30

17.23.1 Fixes

- #161: get pysparse working with python 2.4
- #160: Grid class
- #157: temp files on widows
- #155: fix some of the deprecation warnings appearing in the tests

- #150: PythonXY installation?
- #148: SciPy 0.7.0 solver failures on Macs
- #147: Disable CGS solver in pysparse
- #145: *Viewer* factory fails for rank-1 *CellVariable*
- #144: intermittent failure on *examples/diffusion/explicit/mixedelement.py –inline*
- #142: merge Viewers branch
- #139: Get a Windows Bitten build slave
- #137: Backport examples from manuscript
- #131: *MatplotlibViewer* doesn't properly report the supported file extensions
- #126: Variable, float, integer
- #125: Pickled test data embeds obsolete packages
- #124: Can't pickle a *binOp*
- #121: *simpleTrenchSystem.py*
- #120: mayavi display problems
- #118: Automatically handle casting of *Variable* from *int* to *float* when necessary.
- #117: *getFacesBottom*, *getFacesTop* etc. lack clear description in the reference
- #115: viewing 3D Cahn-Hilliard is broken
- #113: OS X (MacBook Pro; Intel) FiPy installation problems
- #112: *stokesCavity.py* doesn't display properly with matplotlib
- #111: Can't display *Grid2D* variables with matplotlib on Linux
- #110: "Numeric array value must be dimensionless" in ElPhF examples
- #109: doctest of *fipy.variables.variable.Variable.__array__*
- #108: *numerix.array * FaceVariable* is broken
- #107: Can't move matplotlib windows on Mac
- #106: Concatenation of *Grid1D* objects doesn't always work
- #105: useless broken __array__ tests should be removed
- #102: viewer limits should just be set as arguments, rather than as a dict
- #99: *Matplotlib2DGridViewer* cannot update multiple views
- #97: Windows does not seem to handle NaN correctly.
- #96: broken tests with version 2.0 of gmsh
- #95: attached code breaks with *–inline*
- #92: Pygist is dead (it's official)
- #84: Test failures on Intel Mac
- #83: *ZeroDivisionError* for *CellTerm* when calling *getOld()* on its coefficient
- #79: *viewers.make()* to *viewers.Viewer()*
- #67: Mesh viewing and unstructured data.

- #43: *TSVViewer* doesn't always get the right shape for the var
- #34: float(&infinity&) issue on windows

17.24 Version 1.2.1 - 2008-02-08

17.24.1 Fixes

- #122: check argument types for meshes
- #119: max is broken for Variables
- #116: Linux: failed test, *TypeError: No array interface...* in solve()
- #104: Syntax error in *MatplotlibVectorViewer._plot()*
- #101: matplotlib 1D viewer autoscales when a limit is set to 0
- #93: Broken examples
- #91: update the examples to use *from fipy import **
- #76: *solve()* and *sweep()* accept *dt=CellVariable*
- #75: installation of fipy should auto include README as a docstring
- #74: Some combinations of *DiffusionTerm* and *ConvectionTerm* do not work
- #51: __pos__ doesn't work for terms
- #50: Broken examples
- #39: matplotlib broken on mac with version 0.72.1
- #19: Péclet number
- #15: Boundary conditions and Terms

17.25 Version 1.2 - 2007-02-12

The significant changes since version 1.1 are:

- *–inline* automatically generates C code from *Variable* expressions.
- *FiPy* has been updated to use the *Python NumPy* module. *FiPy* no longer works with the older *Numeric* module.

17.25.1 Fixes

- #98: Windows patch for some broken test cases
- #94: *–inline* error for attached code
- #90: bug in matplotlib 0.87.7: *TypeError: only length-1 arrays can be converted to Python scalars*.
- #72: needless rebuilding of variables
- #66: PDF rendering issues for the guide on various platforms
- #62: fipy guide pdf bug: "*an unrecognized token 13c was found*"
- #55: Error for internal BCs

- #52: *FaceVariable * FaceVectorVariable* memory
- #48: Documentation is not inherited from &hidden& classes
- #42: *fipy.models.phase.phase.addOverFacesVariable* is gross
- #41: `EFFICIENCY.txt` example fails to make viewer
- #30: periodic boundary condition support
- #25: make phase field examples more explicit
- #23: sweep control, iterator object, error norms
- #21: Update FiPy to use numpy
- #16: Dimensions
- #12: Refactor viewers
- #1: Gnuplot doesn't display on windows

17.26 Version 1.1 - 2006-06-06

The significant changes since version 1.0 are:

- Memory efficiency has been improved in a number of ways, but most significantly by:
  - not caching all intermediate `Variable` values.
  - introducing `UniformGrid` classes that calculate geometric arrays on the fly.

  Details of these improvements are presented in *Efficiency*.
- Installation on Windows has been made considerably easier by constructing executable installers for *FiPy* and its dependencies.
- The arithmetic for `Variable` subclasses now works, and returns sensible answers. For example, `VectorCellVariable * CellVariable` returns a `VectorCellVariable`.
- `PeriodicGrid` meshes have been implemented. Currently, however, there and no examples of their use in the manual.
- Many of the examples have been completely rewritten
  - A basic 1D diffusion problem now serves as a general tutorial for setting up any problem in *FiPy*.
  - Several more phase field examples have been added that should make it clearer how to get from the simple 1D case to the more elaborate multicomponent, multidimensional, and anisotropic models.
  - The "Superfill" examples have been substantially improved with better functionality and documentation.
  - An example of fluid flow with the classic Stokes moving lid has been added.
- A clear distinction has been made between solving an equation via *solve()* and iterating an non-linear equation to solution via *sweep()*. An extensive explanation of the concepts involved has been added to the *Frequently Asked Questions*.
- Added a *MultiViewer* class that automatically groups several viewers together if the variables couldn't be displayed by a single viewer.
- The abbreviated syntax `from fipy import Class` or `from fipy import *` promised in version 1.0 actually works now. The examples all still use the fully qualified names.

- The repository has been converted from a CVS to a Subversion repository. Details on how to check out the new repository are given in *Installation*.
- The *FiPy* repository has also been moved from Sourceforge to the Materials Digital Library Pathway.

17.27 Version 1.0 - 2005-09-16

Numerous changes have been made since *FiPy* 0.1 was released, but the most significant ones are:

- `Equation` objects no longer exist. PDEs are constructed from `Term` objects. `Term` objects can be added, subtracted, and equated to build up an equation.
- A true 1D grid class has been added: `fipy.meshes.grid1D.Grid1D`.
- A generic "factory" method `fipy.viewers.make()` has been added that will do a reasonable job of automatically creating a `Viewer` for the supplied `Variable` objects. The `FIPY_VIEWER` environment variable allows you to specify your preferred viewer.
- A simple `TSVViewer` has been added to allow display or export to a file of your solution data.
- It is no longer necessary to `transpose()` scalar fields in order to multiply them with vector fields.
- Better default choice of solver when convection is present.
- Better examples.
- A number of *NoiseVariable* objects have been added.
- A new viewer based on *Matplotlib* has been added.
- The *PyX* viewer has been removed.
- Considerably simplified the public interface to FiPy.
- Support for Python 2.4.
- Improved layout of the manuals.
- `getLaplacian()` method has been removed from `CellVariable` objects. You can obtain the same effect with `getFaceGrad().getDivergence()`, which provides better control.
- An `import` shorthand has been added that allows for:

```
from fipy import Class
```

  instead of:

```
from fipy.some.deeply.nested.module.class import Class
```

  This system is still experimental. Please tell us if you find situations that don't work.

The syntax of *FiPy* 1.0 scripts is incompatible with earlier releases. A tutorial for updating your existing scripts can be found in `examples/updating/update0_1to1_0.py`.

17.27.1 Fixes

- #49: Documentation for many *ConvectionTerms* is wrong
- #47: Terms should throw an error on bad *coeff* type
- #40: broken levelset test case
- #38: multiple BCs on one face broken?
- #37: Better support for periodic boundary conditions
- #36: Gnuplot doesn't display the `~examples/levelSet/electroChem` problem on windows.
- #35: gmsh write problem on windows
- #33: *DiffusionTerm(coeff = CellVariable)* functionality
- #32: conflict_handler = *ignore* not valid in Python 2.4
- #31: Support simple import notation
- #29: periodic boundary conditions are broken
- #28: invoke the == for terms
- #26: doctest extraction with python 2.4
- #24: Pysparse windows binaries
- #22: automated efficiency_test problems
- #20: Test with Python version 2.4
- #18: Memory leak for the leveling problem
- #17: *distanceVariable* is broken
- #14: Testing mailing list interface
- #11: Reconcile versions of pysparse
- #10: check phase field crystal growth
- #9: implement levelling surfactant equation
- #8: merge *depositionRateVar* and *extensionVelocity*
- #7: Automate FiPy efficiency test
- #6: FiPy breaks on windows with Numeric 23.6
- #5: axisymmetric 2D mesh
- #4: Windows installation wizard
- #3: Windows installation instructions
- #2: Some tests fail on windows XP

17.28 Version 0.1.1

17.29 Version 0.1 - 2004-11-05

Original release

Chapter 18

Glossary

AppVeyor
A cloud-based *Continuous Integration* tool. See https://www.appveyor.com.

Azure
A cloud-based *Continuous Integration* tool. See https://dev.azure.com.

Buildbot
The Buildbot is a system to automate the compile/test cycle required by most software projects to validate code changes. No longer used for *FiPy*. See http://trac.buildbot.net/.

CircleCI
A cloud-based *Continuous Integration* tool. See https://circleci.com.

conda
An open source package management system and environment management system that runs on Windows, macOS and Linux. Conda quickly installs, runs and updates packages and their dependencies. Conda easily creates, saves, loads and switches between environments on your local computer. It was created for Python programs, but it can package and distribute software for any language. See https://conda.io.

Continuous Integration
The practice of frequently testing and integrating one's new or changed code with the existing code repository. See https://en.wikipedia.org/wiki/Continuous_integration.

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

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

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

JSON
JavaScript Object Notation. A text format suitable for storing structured information such as `dict` or `list`. See https://www.json.org/.

linux
An operating system. See http://www.linux.org.

macOS
An operating system. See http://www.apple.com/macos.

Matplotlib
`matplotlib` *Python* package displays publication quality results. It displays both 1D X-Y type plots and 2D contour plots for structured data. It does not display unstructured 2D data or 3D data. It works on all common platforms and produces publication quality hard copies. See http://matplotlib.sourceforge.net and *Matplotlib*.

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

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

MPI
The Message Passing Interface is a standard that allows the use of multiple processors. See http://www.mpi-forum.org

mpi4py
MPI for Python provides bindings of the Message Passing Interface (*MPI*) standard for the Python programming language, allowing any Python program to exploit multiple processors. For *Solving in Parallel*, *FiPy* requires this package, in addition to *PETSc* or *Trilinos*. See https://mpi4py.readthedocs.io.

numarray
An archaic predecessor to *NumPy*.

Numeric
An archaic predecessor to *NumPy*.

NumPy
The `numpy` *Python* package provides array arithmetic facilities. See http://www.scipy.org/NumPy.

OpenMP
The Open Multi-Processing architecture is a specification for a set of compiler directives, library routines, and environment variables that can be used to specify high-level parallelism in Fortran and C/C++ programs. See https://www.openmp.org.

pandas
"Python Data Analysis Library" provides high-performance data structures for flexible, extensible analysis. See http://pandas.pydata.org.

PETSc
The Portable, Extensible Toolkit for Scientific Computation is a suite of data structures and routines for the scalable (parallel) solution of scientific applications modeled by partial differential equations. See https://www.mcs.anl.gov/petsc and *PETSc*.

petsc4py
Python wrapper for *PETSc*. See https://petsc4py.readthedocs.io/.

pip
"pip installs python" is a tool for installing and managing Python packages, such as those found in *PyPI*. See http://www.pip-installer.org.

PyAMG
A suite of python-based preconditioners. See http://code.google.com/p/pyamg/ and *PyAMG*.

pyamgx
a *Python* interface to the NVIDIA AMGX library, which can be used to construct complex solvers and preconditioners to solve sparse sparse linear systems on the GPU. See https://pyamgx.readthedocs.io/ and *pyamgx*.

PyPI
The Python Package Index is a repository of software for the *Python* programming language. See http://pypi.python.org/pypi.

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

Pysparse
The `pysparse` *Python* package provides sparse matrix storage, solvers, and linear algebra routines. See http://pysparse.sourceforge.net and *Pysparse*.

Python
The programming language that *FiPy* (and your scripts) are written in. See http://www.python.org/.

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

PyTrilinos
Python wrapper for *Trilinos*. See http://trilinos.sandia.gov/packages/pytrilinos/.

PyxViewer
A now defunct python viewer.

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

SciPy
The `scipy` package provides a wide range of scientific and mathematical operations. *FiPy* can use *Scipy*'s solver suite for linear solutions. See http://www.scipy.org/. and *SciPy*.

Sphinx
The tools used to generate the *FiPy* documentation. See http://sphinx.pocoo.org/.

TravisCI
A cloud-based *Continuous Integration* tool. See https://travis-ci.org.

Trilinos
This package provides sparse matrix storage, solvers, and preconditioners, and can be used instead of *Pysparse*. *Trilinos* preconditioning allows for iterative solutions to some difficult problems that *Pysparse* cannot solve. See http://trilinos.sandia.gov and *Trilinos*.

Weave
The `weave` package can enhance performance with C language inlining. See https://github.com/scipy/weave.

Windows
An operating system. See http://www.microsoft.com/windows.

Part II

Examples

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

These examples can be used in at least four ways:

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

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

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

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

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

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

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

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

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

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

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

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

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

Chapter 19

Diffusion Examples

- examples.diffusion.mesh1D
- examples.diffusion.coupled
- examples.diffusion.mesh20x20
- examples.diffusion.circle
- examples.diffusion.electrostatics
- examples.diffusion.nthOrder.input4thOrder1D
- examples.diffusion.anisotropy

Chapter 20

Convection Examples

- examples.convection.exponential1D.mesh1D
- examples.convection.exponential1DSource.mesh1D
- examples.convection.robin
- examples.convection.source

Chapter 21

Phase Field Examples

- examples.phase.simple
- examples.phase.binary
- examples.phase.binaryCoupled
- examples.phase.quaternary
- examples.phase.anisotropy
- examples.phase.impingement.mesh40x1
- examples.phase.impingement.mesh20x20
- examples.phase.polyxtal
- examples.phase.polyxtalCoupled

Chapter 22

Level Set Examples

- examples.levelSet.distanceFunction.mesh1D
- examples.levelSet.distanceFunction.circle
- examples.levelSet.advection.mesh1D
- examples.levelSet.advection.circle

Chapter 23

Cahn-Hilliard Examples

- examples.cahnHilliard.mesh2DCoupled
- examples.cahnHilliard.sphere

Chapter 24

Fluid Flow Examples

- examples.flow.stokesCavity

Chapter 25

Reactive Wetting Examples

- examples.reactiveWetting.liquidVapor1D

Chapter 26

Updating FiPy

- examples.updating.update2_0to3_0
- examples.updating.update1_0to2_0
- examples.updating.update0_1to1_0

Part III

`fipy` Package Documentation

Chapter 27

How to Read the Modules Documentation

Each chapter describes one of the main sub-packages of the `fipy` package. The sub-package `fipy.package` can be found in the directory `fipy/package/`. In a few cases, there will be packages within packages, *e.g.* `fipy.package.subpackage` located in `fipy/package/subpackage/`. These sub-sub-packages will not be given their own chapters; rather, their contents will be described in the chapter for their containing package.

Bibliography

[1] W. J. Boettinger, J. A. Warren, C. Beckermann, and A. Karma. Phase-field simulation of solidification. *Annual Review of Materials Research*, 32:163–194, 2002. doi:10.1146/annurev.matsci.32.101901.155803.

[2] L. Q. Chen. Phase-field mdoels for microstructure evolution. *Annual Review of Materials Research*, 32:113–140, 2002. doi:10.1146/annurev.matsci.32.112001.132041.

[3] G. B. McFadden. Phase-field models of solidification. *Contemporary Mathematics*, 306:107–145, 2002.

[4] David M Saylor, Jonathan E Guyer, Daniel Wheeler, and James A Warren. Predicting microstructure development during casting of drug-eluting coatings. *Acta Biomaterialia*, 7(2):604–613, Jan 2011. doi:10.1016/j.actbio.2010.09.019.

[5] Daniel Wheeler, James A. Warren, and William J. Boettinger. Modeling the early stages of reactive wetting. *Physical Review E*, 82(5):051601, Nov 2010. doi:10.1103/PhysRevE.82.051601.

[6] C. M Hangarter, B. H Hamadani, J. E Guyer, H Xu, R Need, and D Josell. Three dimensionally structured interdigitated back contact thin film heterojunction solar cells. *Journal of Applied Physics*, 109(7):073514, Jan 2011. doi:10.1063/1.3561487.

[7] D. Josell, D. Wheeler, W. H. Huber, and T. P. Moffat. Superconformal electrodeposition in submicron features. *Physical Review Letters*, 87(1):016102, 2001. doi:10.1103/PhysRevLett.87.016102.

[8] J. A. Sethian. *Level Set Methods and Fast Marching Methods*. Cambridge University Press, 1996.

[9] Scott Chacon. *Pro Git*. Apress, 2009. URL: http://git-scm.com/book.

[10] Guido van Rossum. *Python Tutorial*. URL: http://docs.python.org/tut/.

[11] Mark Pilgrim. *Dive Into Python*. Apress, 2004. ISBN 1590593561. URL: http://diveintopython.org.

[12] Guido van Rossum. *Python Reference Manual*. URL: http://docs.python.org/ref/.

[13] 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):6035–6058, 2003. doi:10.1016/S1359-6454(03)00388-4.

[14] T. N. Croft. *Unstructured Mesh - Finite Volume Algorithms for Swirling, Turbulent Reacting Flows*. PhD thesis, University of Greenwich, 1998. URL: http://gala.gre.ac.uk/id/eprint/6371/.

[15] S. V. Patankar. *Numerical Heat Transfer and Fluid Flow*. Taylor and Francis, 1980.

[16] H. K. Versteeg and W. Malalasekera. *An Introduction to Computational Fluid Dynamics*. Longman Scientific and Technical, 1995.

[17] C. Mattiussi. An analysis of finite volume, finite element, and finite difference methods using some concepts from algebraic topology. *Journal of Computational Physics*, 133:289–309, 1997. URL: http://lis.epfl.ch/publications/JCP1997.pdf.

[18] John W. Cahn and John E. Hilliard. Free energy of a nonuniform system. I. Interfacial free energy. *Journal of Chemical Physics*, 28(2):258–267, 1958.

[19] John W. Cahn. Free energy of a nonuniform system. II. Thermodynamic basis. *Journal of Chemical Physics*, 30(5):1121–1124, 1959.

[20] John W. Cahn and John E. Hilliard. Free energy of a nonuniform system. III. Nucleation in a two-component incompressible fluid. *Journal of Chemical Physics*, 31(3):688–699, 1959.

[21] K. R Elder, K Thornton, and J. J Hoyt. The kirkendall effect in the phase field crystal model. *Philosophical Magagazine*, 91(1):151–164, Jan 2011. doi:10.1080/14786435.2010.506427.

[22] Greg Ward. *Installing Python Modules*. URL: http://docs.python.org/inst/.

[23] 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.

[24] D. Wheeler, D. Josell, and T. P. Moffat. Modeling superconformal electrodeposition using the level set method. *Journal of The Electrochemical Society*, 150(5):C302–C310, 2003. doi:10.1149/1.1562598.

[25] D. Josell, D. Wheeler, and T. P. Moffat. Gold superfill in submicrometer trenches: experiment and prediction. *Journal of The Electrochemical Society*, 153(1):C11–C18, 2006. doi:10.1149/1.2128765.

[26] 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):C127–C132, 2006. doi:10.1149/1.2165580.

[27] 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):7424–7439, 1992.

[28] 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):689–703, 1995.

[29] J. E. Guyer, W. J. Boettinger, J. A. Warren, and G. B. McFadden. Phase field modeling of electrochemistry I: Equilibrium. *Physical Review E*, 69:021603, 2004. arXiv:cond-mat/0308173, doi:10.1103/PhysRevE.69.021603.

[30] J. E. Guyer, W. J. Boettinger, J. A. Warren, and G. B. McFadden. Phase field modeling of electrochemistry II: Kinetics. *Physical Review E*, 69:021604, 2004. arXiv:cond-mat/0308179, doi:10.1103/PhysRevE.69.021604.

[31] T. P. Moffat, D. Wheeler, and D. Josell. Superfilling and the curvature enhanced accelerator coverage mechanism. *The Electrochemical Society, Interface*, 13(4):46–52, 2004. URL: http://www.electrochem.org/publications/interface/winter2004/IF12-04-Pg46.pdf.

[32] J. H. Ferziger and M. Perić. *Computational Methods for Fluid Dynamics*. Springer, 1996.

[33] C.-C. Rossow. A blended pressure/density based method for the computation of incompressible and compressible flows. *Journal of Computational Physics*, 185(2):375–398, 2003. doi:10.1016/S0021-9991(02)00059-1.

Index

Symbols

A

B

C

E

F

G

I

J

L

M

N

O

P

S

T

W