

FiPy

A Finite Volume PDE Solver Using Python

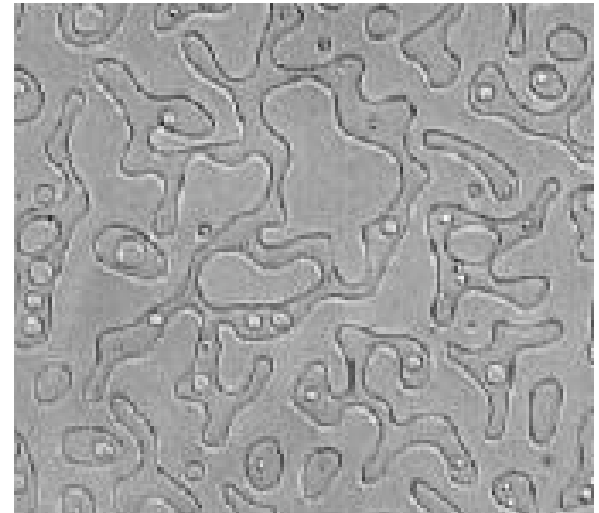
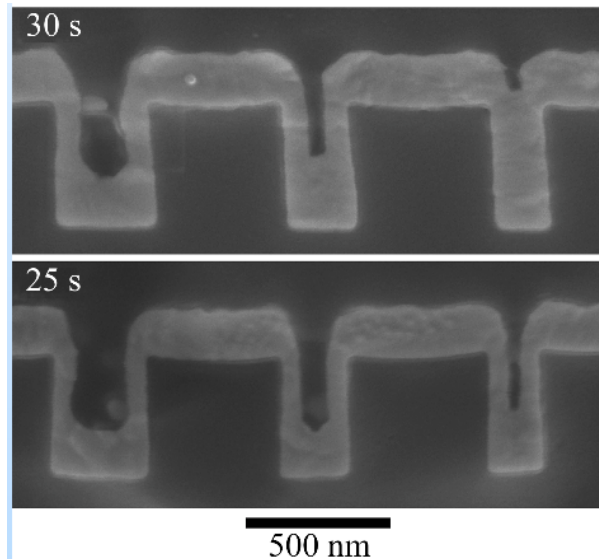
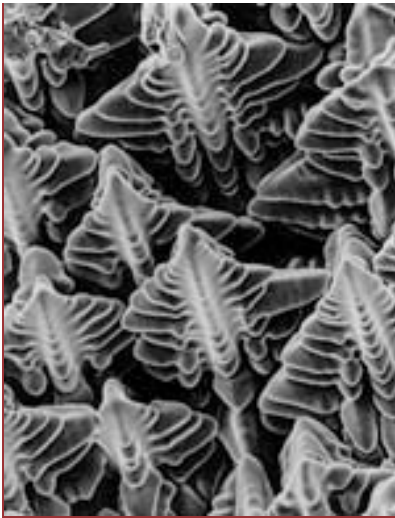
D. Wheeler, J. E. Guyer & J. A. Warren

www.ctcms.nist.gov/fipy/

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

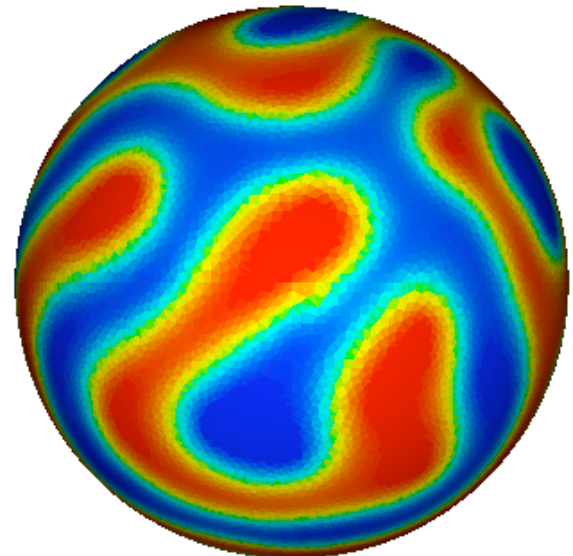
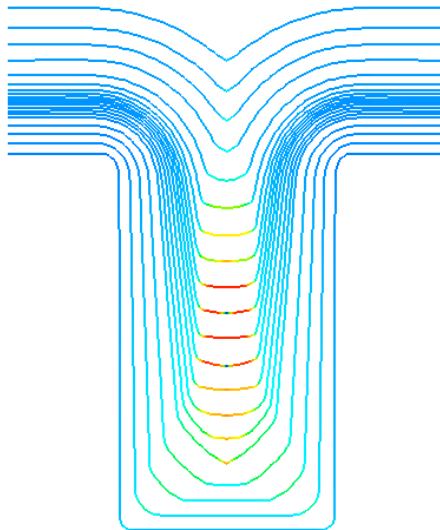
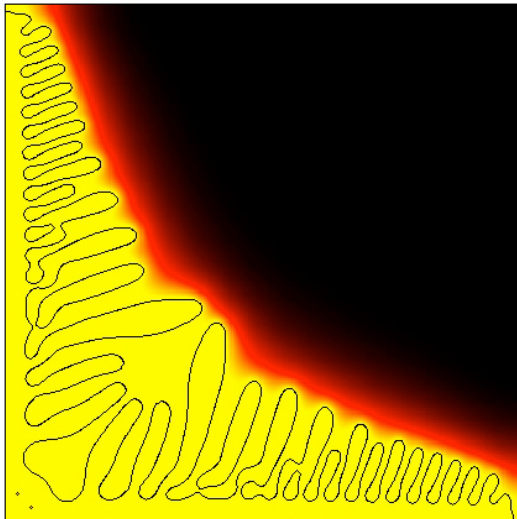
Motivation

- Why a new PDE solver?
 - Codes written in C or FORTRAN have some inherent inflexibility
 - Constant writing of new codes in CTCMS
 - Institutional memory loss
 - No distribution mechanism
- Materials scientists: Unique system, ability to pose problems, customize, without numerical background



What is FiPy?

- FiPy is a computer program written in Python to solve PDEs using the Finite Volume method
- Python is a powerful object oriented scripting language with tools for numerics
- The Finite Volume method is a way to solve a set of PDEs, similar to the Finite Element or Finite Difference methods



Simple Example

$$\underbrace{\frac{\partial \phi}{\partial t}}_{\text{transient}} = \underbrace{\nabla \cdot (\nabla \phi)}_{\text{diffusion}} + \underbrace{1}_{\text{source}}$$
$$\phi|_{x=0} = 0 \quad \phi|_{x=L} = 0$$

```
>>> ## create the mesh
.....
>>> ## create the field variable
.....
>>> ## create the viewer
.....
>>> ## create the equation
.....
>>> ## create the boundary conditions
.....
>>> ## solve the equation and plot the results
.....
```

Simple Example

$$\underbrace{\frac{\partial \phi}{\partial t}}_{\text{transient}} = \underbrace{\nabla \cdot (\nabla \phi)}_{\text{diffusion}} + \underbrace{1}_{\text{source}}$$
$$\phi|_{x=0} = 0 \quad \phi|_{x=L} = 0$$

```
>>> ## create the mesh
>>> from fipy.meshes.grid1D import Grid1D
>>> nx = 1000
>>> L = 1.
>>> mesh = Grid1D(nx = nx, dx = L / nx)

>>> ## create the field variable
.....
>>> ## create the viewer
.....
>>> ## create the equation
.....
>>> ## create the boundary conditions
.....
>>> ## solve the equation and plot the results
.....
```

Simple Example

$$\underbrace{\frac{\partial \phi}{\partial t}}_{\text{transient}} = \underbrace{\nabla \cdot (\nabla \phi)}_{\text{diffusion}} + \underbrace{1}_{\text{source}}$$
$$\phi|_{x=0} = 0 \quad \phi|_{x=L} = 0$$

```
>>> ## create the mesh
```

```
.....
```

```
>>> ## create the field variable
```

```
>>> from fipy.variables.cellVariable import CellVariable
```

```
>>> var = CellVariable(mesh = mesh)
```

```
>>> var.setValue(0.3, mesh.getCells(lambda cell: 0.4 * L < cell.getCenter() < 0.6 * L))
```

```
>>> ## create the viewer
```

```
.....
```

```
>>> ## create the equation
```

```
.....
```

```
>>> ## create the boundary conditions
```

```
.....
```

```
>>> ## solve the equation and plot the results
```

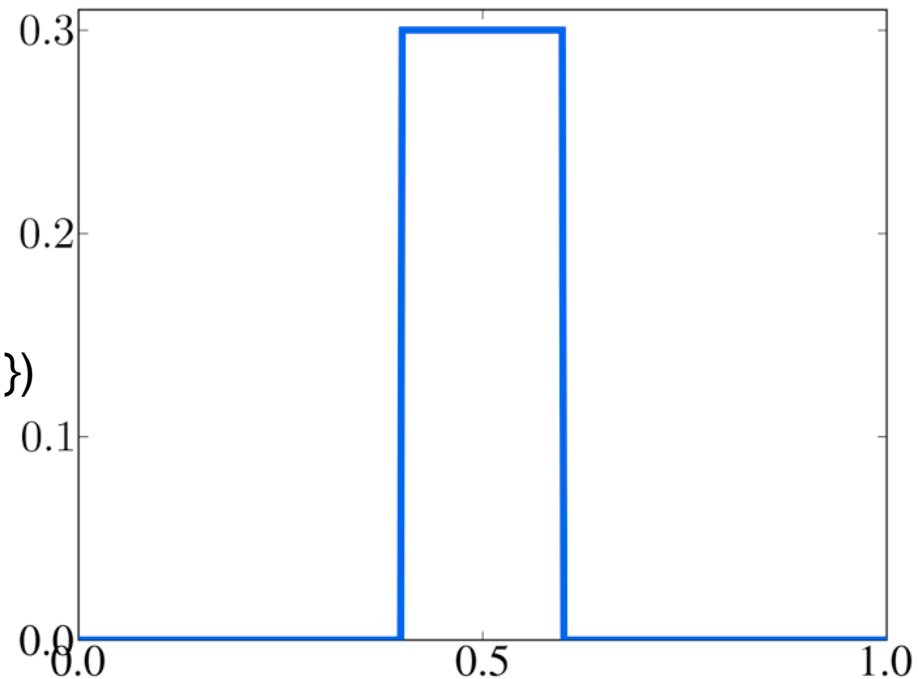
```
.....
```

Simple Example

$$\underbrace{\frac{\partial \phi}{\partial t}}_{\text{transient}} = \underbrace{\nabla \cdot (\nabla \phi)}_{\text{diffusion}} + \underbrace{1}_{\text{source}}$$
$$\phi|_{x=0} = 0 \quad \phi|_{x=L} = 0$$

```
>>> ## create the mesh
.....
>>> ## create the field variable
.....
>>> ## create the viewer
>>> from fipy.viewers import make
>>> viewer = make(var, limits = {'datamax' : 0.31})
>>> viewer.plot()

>>> ## create the equation
.....
>>> ## create the boundary conditions
.....
>>> ## solve the equation and plot the results
.....
```



Simple Example

$$\underbrace{\frac{\partial \phi}{\partial t}}_{\text{transient}} = \underbrace{\nabla \cdot (\nabla \phi)}_{\text{diffusion}} + \underbrace{1}_{\text{source}}$$
$$\phi|_{x=0} = 0 \quad \phi|_{x=L} = 0$$

```
>>> ## create the mesh
.....
>>> ## create the field variable
.....
>>> ## create the viewer
.....
>>> ## create the equation
>>> from fipy.terms.transientTerm import TransientTerm
>>> from fipy.terms.diffusionTerm import DiffusionTerm
>>> eqn = TransientTerm() == DiffusionTerm() + 1.

>>> ## create the boundary conditions
.....
>>> ## solve the equation and plot the results
.....
```


Simple Example

$$\underbrace{\frac{\partial \phi}{\partial t}}_{\text{transient}} = \underbrace{\nabla \cdot (\nabla \phi)}_{\text{diffusion}} + \underbrace{1}_{\text{source}}$$
$$\phi|_{x=0} = 0 \quad \phi|_{x=L} = 0$$

```
>>> ## create the mesh
```

```
.....
```

```
>>> ## create the field variable
```

```
.....
```

```
>>> ## create the viewer
```

```
.....
```

```
>>> ## create the equation
```

```
.....
```

```
>>> ## create the boundary conditions
```

```
>>> from fipy.boundaryConditions.fixedValue import FixedValue
```

```
>>> BCs = (FixedValue(mesh.getFacesLeft(), 0),  
...       FixedValue(mesh.getFacesRight(), 0))
```

```
>>> ## solve the equation and plot the results
```

```
.....
```

Simple Example

$$\underbrace{\frac{\partial \phi}{\partial t}}_{\text{transient}} = \underbrace{\nabla \cdot (\nabla \phi)}_{\text{diffusion}} + \underbrace{1}_{\text{source}}$$

$$\phi|_{x=0} = 0 \quad \phi|_{x=L} = 0$$

```
>>> ## create the mesh
```

```
.....
```

```
>>> ## create the field variable
```

```
.....
```

```
>>> ## create the viewer
```

```
.....
```

```
>>> ## create the equation
```

```
.....
```

```
>>> ## create the boundary conditions
```

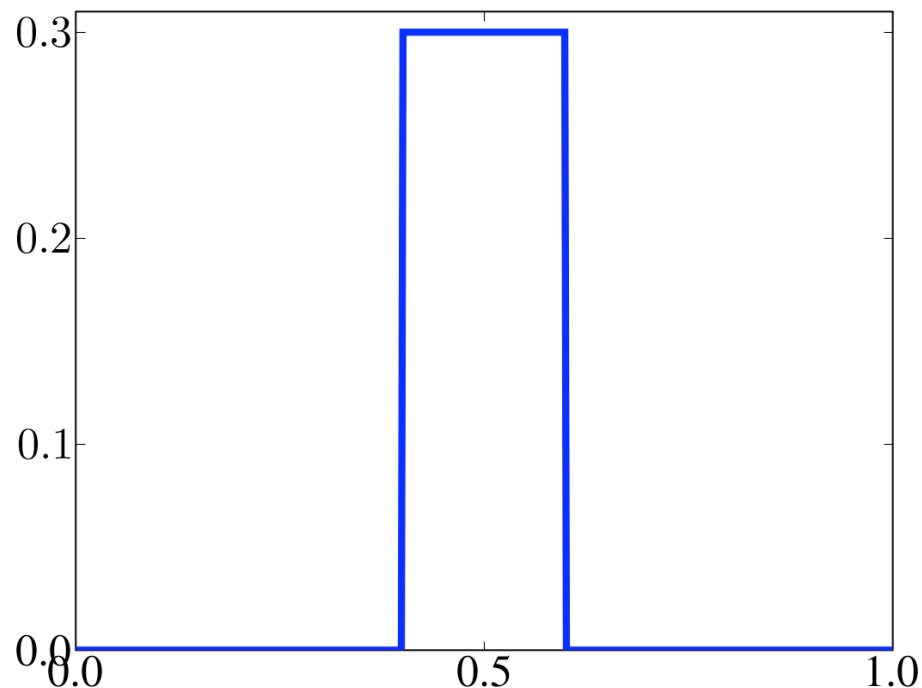
```
.....
```

```
>>> ## solve the equation and plot the results
```

```
>>> for step in range(100):
```

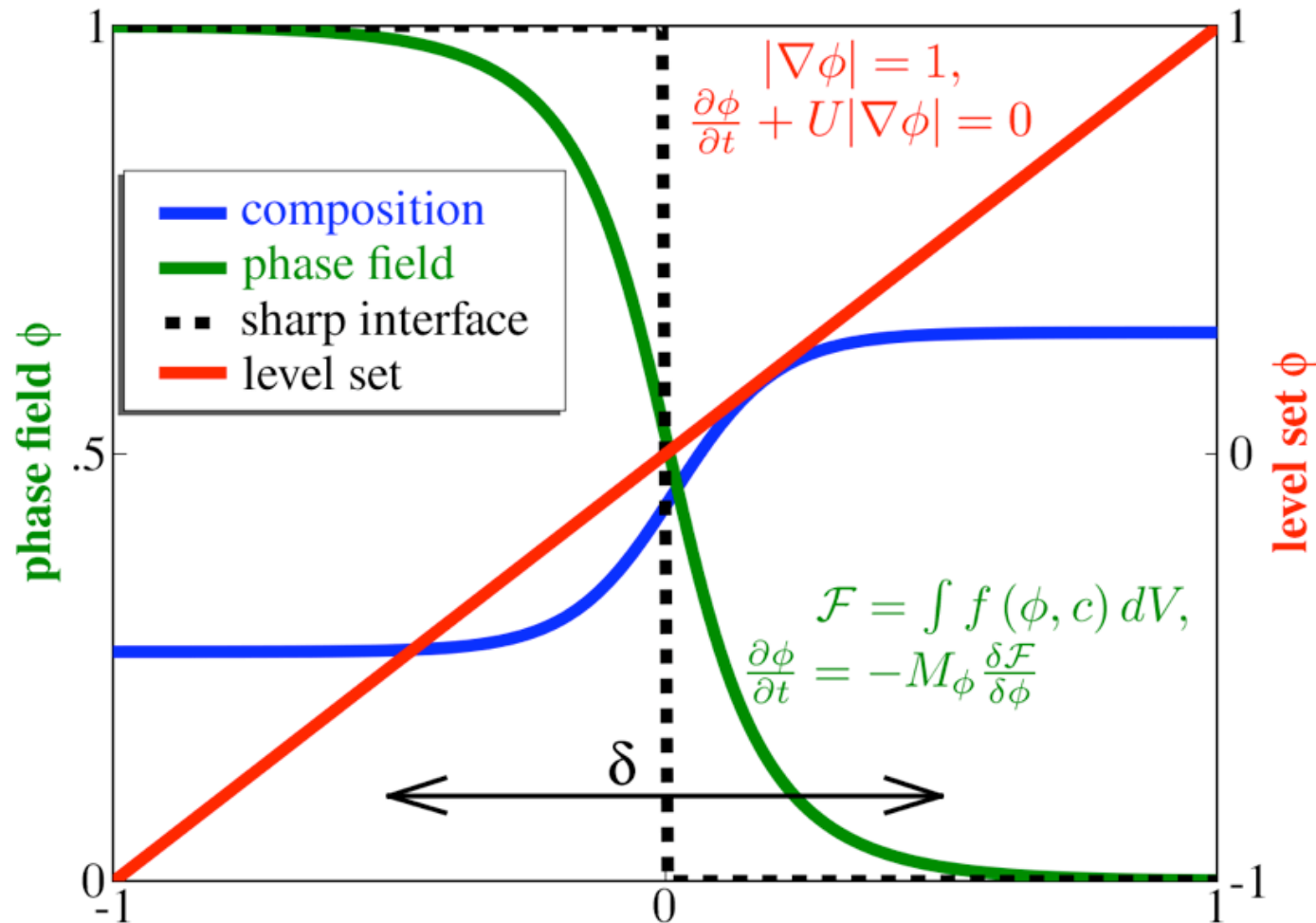
```
...     eqn.solve(var, boundaryConditions = BCs, dt = 0.001)
```

```
...     viewer.plot()
```



Phase Transformations / Material Interfaces

- Typical system consists of:
 - one non-conservative equation for interface or order parameter
 - multiple conservative equations for species concentrations and heat
- Phase field, level set or sharp interface methods



Phase Separation

```
>>> ## create a mesh
```

```
.....
```

```
>>> ## create a field variable
```

```
.....
```

```
>>> ## create the equation
```

```
.....
```

```
>>> ## create a solver
```

```
.....
```

```
>>> ## create a viewer
```

```
.....
```

```
>>> ## solve the equation and plot the results
```

```
.....
```

$$\frac{\partial \phi}{\partial t} = \nabla \cdot D \nabla \left(\frac{\partial f}{\partial \phi} - \epsilon^2 \nabla^2 \phi \right)$$

$$f = \frac{a^2}{2} \phi^2 (1 - \phi)^2$$

$$\vec{n} \cdot \nabla \phi = 0 \quad \text{on all boundaries}$$

$$\vec{n} \cdot \nabla^3 \phi = 0 \quad \text{on all boundaries}$$

Phase Separation

```
>>> ## create a mesh
>>> from fipy.meshes.grid2D import Grid2D
>>> N = 100
>>> mesh = Grid2D(dx = 2., dy = 2., nx = N, ny = N)
>>>
>>> ## create a field variable
.....
>>> ## create the equation
.....
>>> ## create a solver
.....
>>> ## create a viewer
.....
>>> ## solve the equation and plot the results
.....
```

$$\frac{\partial \phi}{\partial t} = \nabla \cdot D \nabla \left(\frac{\partial f}{\partial \phi} - \epsilon^2 \nabla^2 \phi \right)$$

$$f = \frac{a^2}{2} \phi^2 (1 - \phi)^2$$

$$\vec{n} \cdot \nabla \phi = 0 \quad \text{on all boundaries}$$

$$\vec{n} \cdot \nabla^3 \phi = 0 \quad \text{on all boundaries}$$

Phase Separation

$$\frac{\partial \phi}{\partial t} = \nabla \cdot D \nabla \left(\frac{\partial f}{\partial \phi} - \epsilon^2 \nabla^2 \phi \right)$$

$$f = \frac{a^2}{2} \phi^2 (1 - \phi)^2$$

```
>>> ## create a mesh
.....
>>> ## create a field variable
>>> from RandomArray import random
>>> from fipy.variables.cellVariable import CellVariable
>>> var = CellVariable(mesh = mesh,
>>>                     value = 0.5 + 0.01 * (random(mesh.getNumberOfCells()) - 0.5))
>>>
>>> ## create the equation
.....
>>> ## create a solver
.....
>>> ## create a viewer
.....
>>> ## solve the equation and plot the results
.....
```

Phase Separation

```
>>> ## create a mesh
```

```
.....
```

```
>>> ## create a field variable
```

```
.....
```

```
>>> ## create the equation
```

```
>>> a = 1
```

```
>>> e = 1
```

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

```
>>> from fipy.terms.diffusionTerm import DiffusionTerm
```

```
>>> from fipy.terms.transientTerm import TransientTerm
```

```
>>> eqn = TransientTerm() == DiffusionTerm(D * a**2 * (1 - 6 * var * (1 - var))) - \
    DiffusionTerm((D, e**2))
```

```
...
```

```
>>>
```

```
>>> ## create a solver
```

```
.....
```

```
>>> ## create a viewer
```

```
.....
```

```
>>> ## solve the equation and plot the results
```

$$\underbrace{\frac{\partial \phi}{\partial t}}_{\text{transient}} = \underbrace{\nabla \cdot D a^2 [1 - 6\phi(1 - \phi)] \nabla \phi}_{2^{\text{nd}} \text{ order diffusion}} - \underbrace{\nabla \cdot D \nabla \epsilon^2 \nabla^2 \phi}_{4^{\text{th}} \text{ order diffusion}}$$

$$\vec{n} \cdot \nabla \phi = 0 \quad \text{on all boundaries}$$

$$\vec{n} \cdot \nabla^3 \phi = 0 \quad \text{on all boundaries}$$

Phase Separation

```
>>> ## create a mesh
```

```
.....
```

```
>>> ## create a field variable
```

```
.....
```

```
>>> ## create the equation
```

```
.....
```

```
>>> ## create a solver
```

```
>>> from fipy.solvers.linearLUSolver import LinearLUSolver
```

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

```
>>>
```

```
>>> ## create a viewer
```

```
.....
```

```
>>> ## solve the equation and plot the results
```

```
.....
```

$$\frac{\partial \phi}{\partial t} = \nabla \cdot D \nabla \left(\frac{\partial f}{\partial \phi} - \epsilon^2 \nabla^2 \phi \right)$$

$$f = \frac{a^2}{2} \phi^2 (1 - \phi)^2$$

Phase Separation

```
>>> ## create a mesh
```

```
.....
```

```
>>> ## create a field variable
```

```
.....
```

```
>>> ## create the equation
```

```
.....
```

```
>>> ## create a solver
```

```
.....
```

```
>>> ## create a viewer
```

```
>>> from fipy.viewers import make
```

```
>>> viewer = make(vars = var, limits = {'datamin': 0, 'datamax': 1})
```

```
>>> viewer.plot()
```

```
>>>
```

```
>>> ## solve the equation and plot the results
```

```
.....
```

$$\frac{\partial \phi}{\partial t} = \nabla \cdot D \nabla \left(\frac{\partial f}{\partial \phi} - \epsilon^2 \nabla^2 \phi \right)$$

$$f = \frac{a^2}{2} \phi^2 (1 - \phi)^2$$

$$\vec{n} \cdot \nabla \phi = 0 \quad \text{on all boundaries}$$

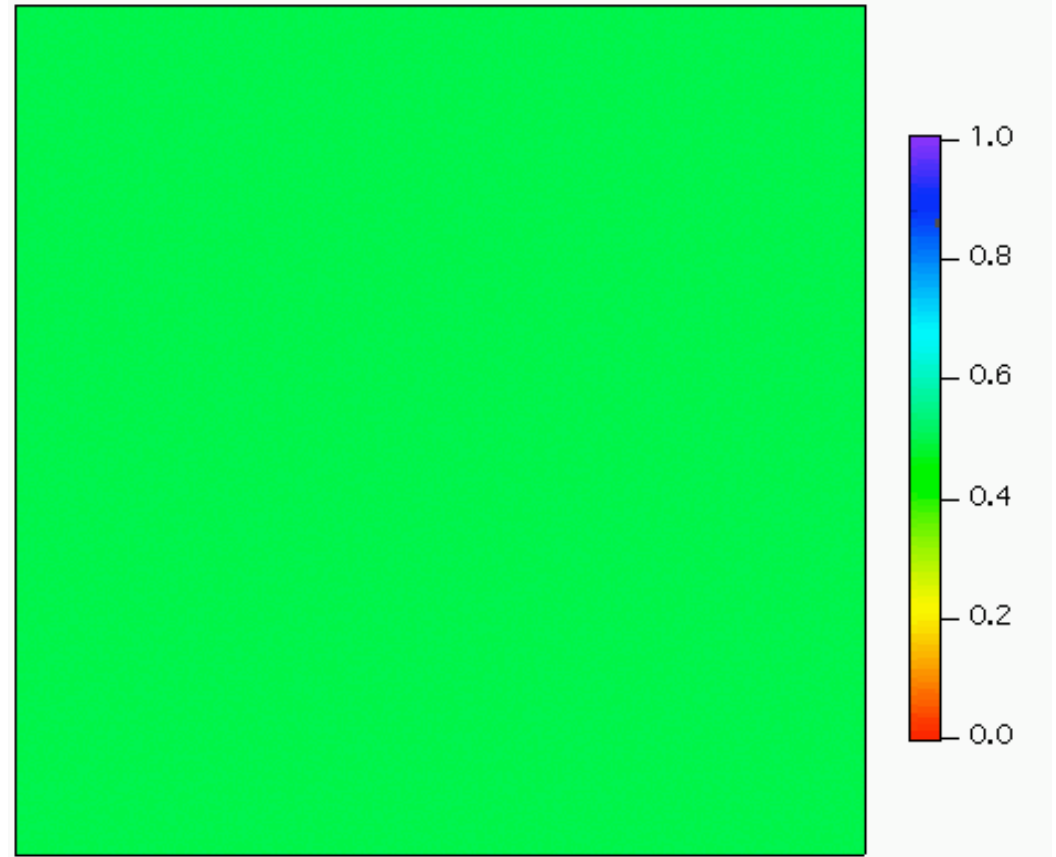
$$\vec{n} \cdot \nabla^3 \phi = 0 \quad \text{on all boundaries}$$

Phase Separation

$$\frac{\partial \phi}{\partial t} = \nabla \cdot D \nabla \left(\frac{\partial f}{\partial \phi} - \epsilon^2 \nabla^2 \phi \right)$$

$$f = \frac{a^2}{2} \phi^2 (1 - \phi)^2$$

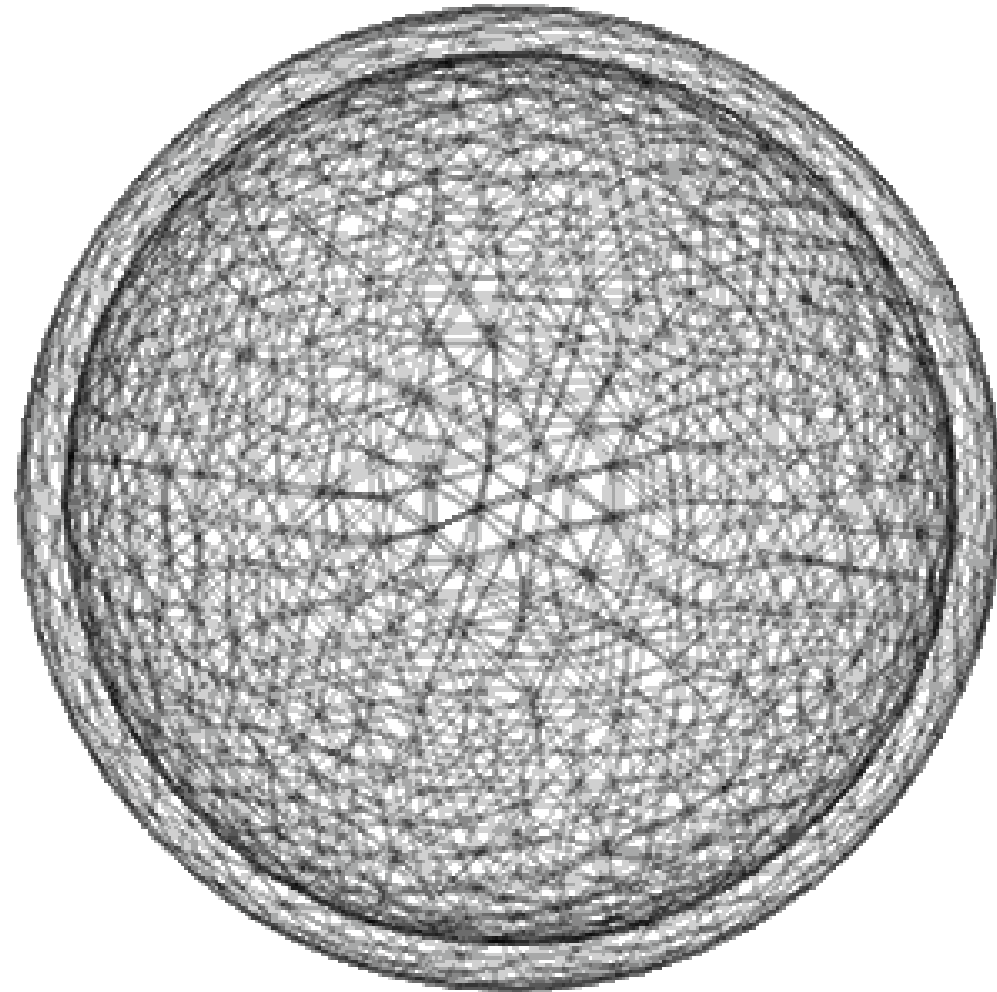
```
>>> ## create a mesh
.....
>>> ## create a field variable
.....
>>> ## create the equation
.....
>>> ## create a solver
.....
>>> ## create a viewer
.....
>>> ## solve the equation and plot the results
>>> dexp=-5
>>> from fipy.tools import numerix
>>> for step in range(1000):
...     eqn.solve(var, solver = solver, \
...               dt = min(10, numerix.exp(dexp)))
...     dexp += 0.1
...     viewer.plot()
```



Phase Separation

$$\frac{\partial \phi}{\partial t} = \nabla \cdot D \nabla \left(\frac{\partial f}{\partial \phi} - \epsilon^2 \nabla^2 \phi \right)$$

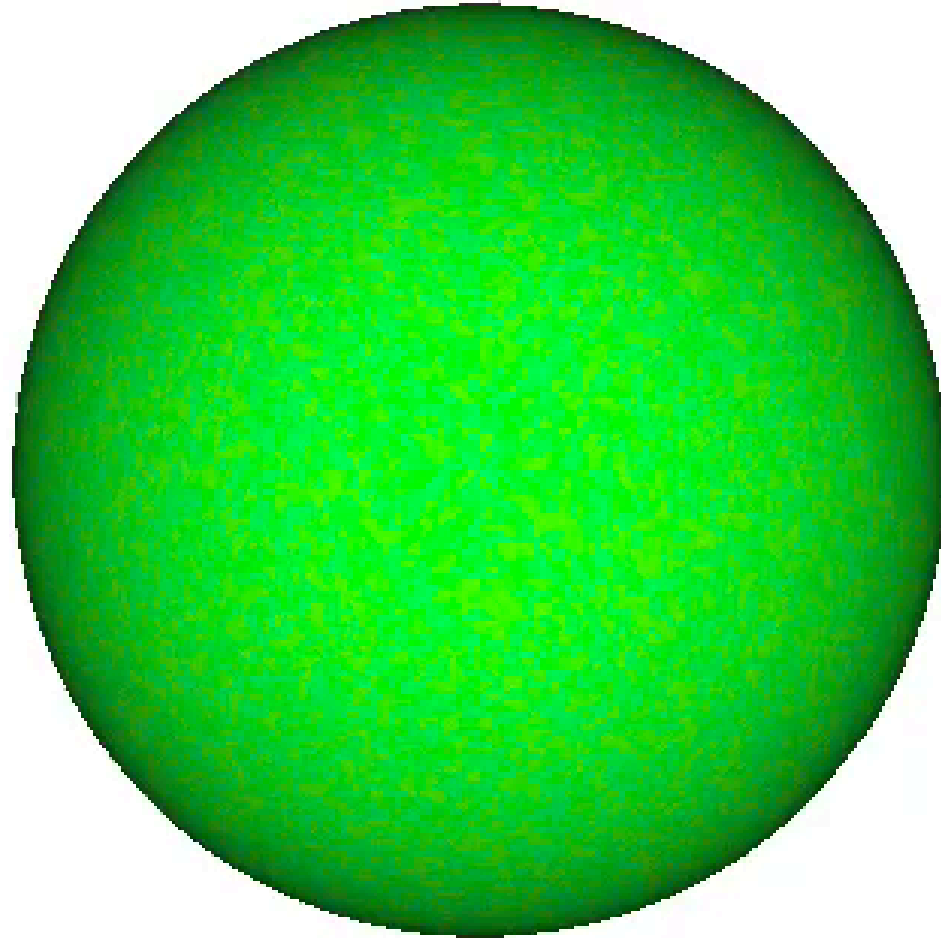
```
>>> ## create a mesh
>>> from fipy.meshes.gmshImport import GmshImporter
>>> mesh = GmshImporter('sphere.msh')
>>> ## create a field variable
.....
>>> ## create the equation
.....
>>> ## create a solver
.....
>>> ## create a viewer
.....
>>> ## solve the equation and plot the results
.....
```



Phase Separation

$$\frac{\partial \phi}{\partial t} = \nabla \cdot D \nabla \left(\frac{\partial f}{\partial \phi} - \epsilon^2 \nabla^2 \phi \right)$$

```
>>> ## create a mesh
.....
>>> ## create a field variable
.....
>>> ## create the equation
.....
>>> ## create a solver
.....
>>> ## create a viewer
.....
>>> ## solve the equation and plot the results
>>> dexp=-5
>>> from fipy.tools import numerix
>>> for step in range(1000):
...     dt = min(10, numerix.exp(dexp))
...     dexp += 0.1
...     eqn.solve(var, solver = solver, dt = dt)
...     viewer.plot()
```

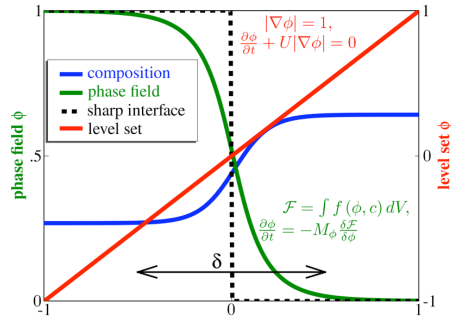


Range of FiPy examples

- Problems modeled with FiPy
 - Phase separation
 - Dendrites
 - Grain growth
 - Ternary Alloys
 - Phase field crystals
 - Kirkendall effect
 - Electrochemistry
 - Superconformal electrodeposition

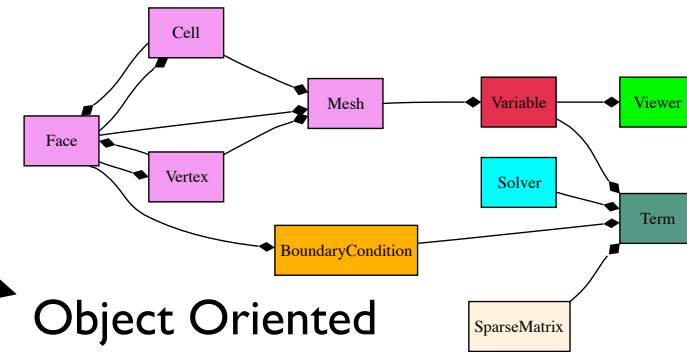
Design

$$\int_V \frac{\partial(\rho\phi)}{\partial t} dV = \underbrace{\int_S \Gamma(\vec{n} \cdot \nabla \phi) dS}_{\text{diffusion}} + \underbrace{\int_S \Gamma_n(\vec{n} \cdot \nabla \dots) dS}_{n^{\text{th}} \text{ order diffusion}} + \underbrace{\int_S (\vec{n} \cdot \vec{u}) \phi dS}_{\text{convection}} + \underbrace{\int_V S_\phi dV}_{\text{source}}$$



Finite Volume Method

Documentation



Object Oriented

Testing

```
$ python setup.py test
running test
import fiPy.solvers.test ... ok
import fiPy.models.test ... ok
import fiPy.terms.test ... ok
import fiPy.tools.test ... ok
import fiPy.meshes.numMesh.test ... ok
import fiPy.variables.test ... ok
import fiPy.viewers.test ... ok
...
```

Distribution (CVS, bugs, mailing lists)

Interface Tracking

FiPy

Leverage existing tools

Python

numarray

PySparse

matplotlib



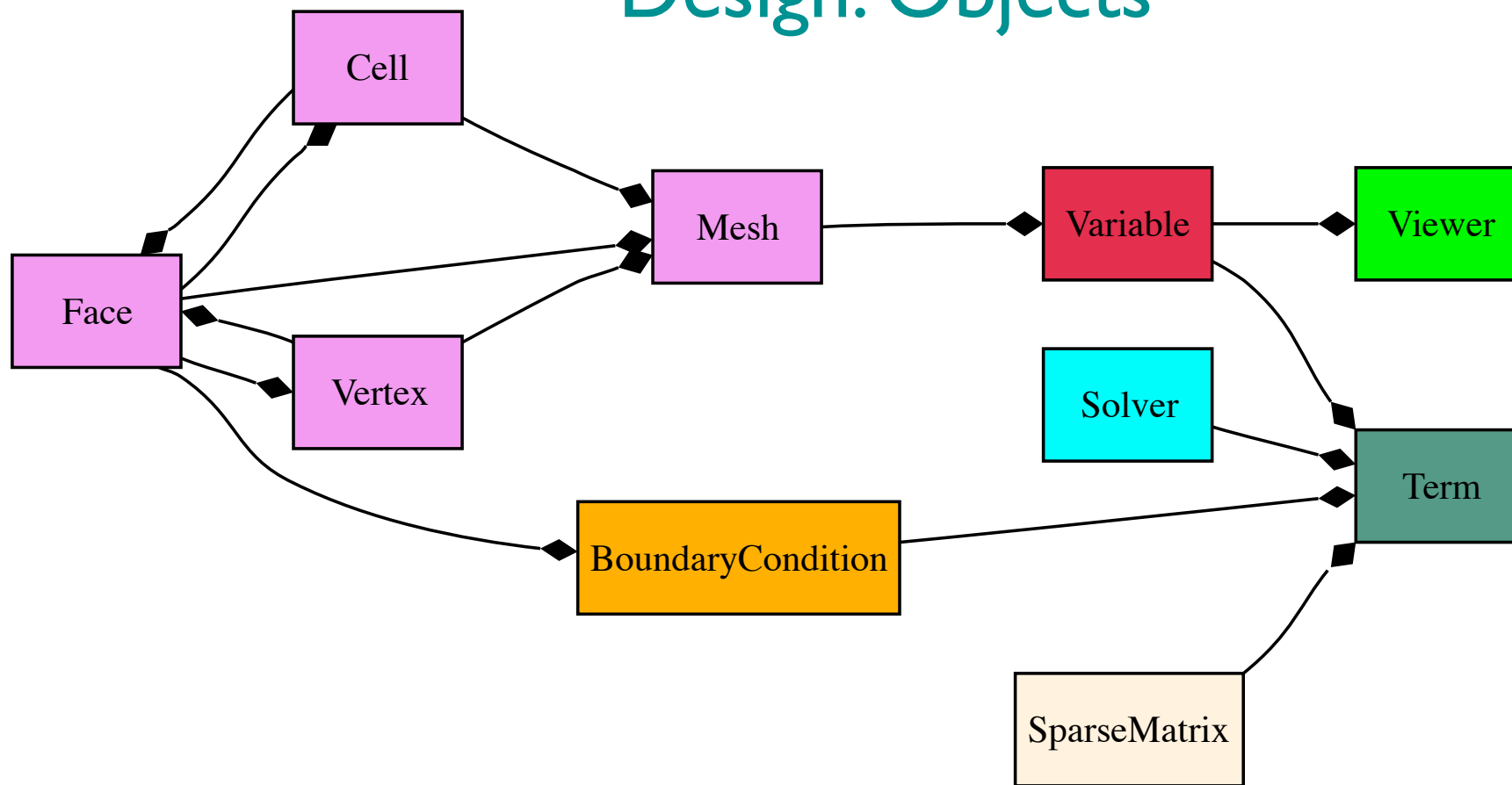
ScientificPython

NIST

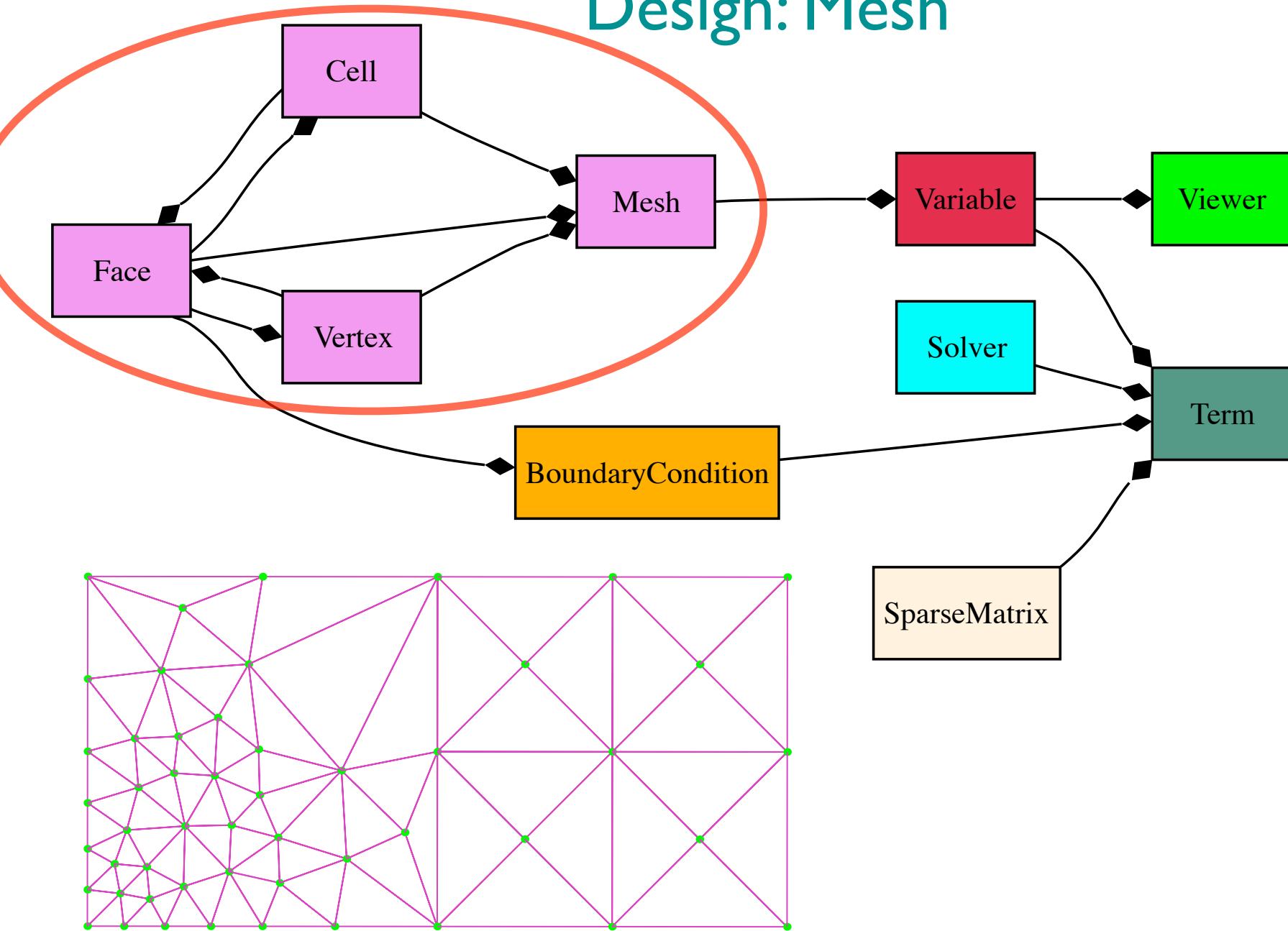
National Institute of Standards and Technology
Technology Administration, U.S. Department of Commerce



Design: Objects

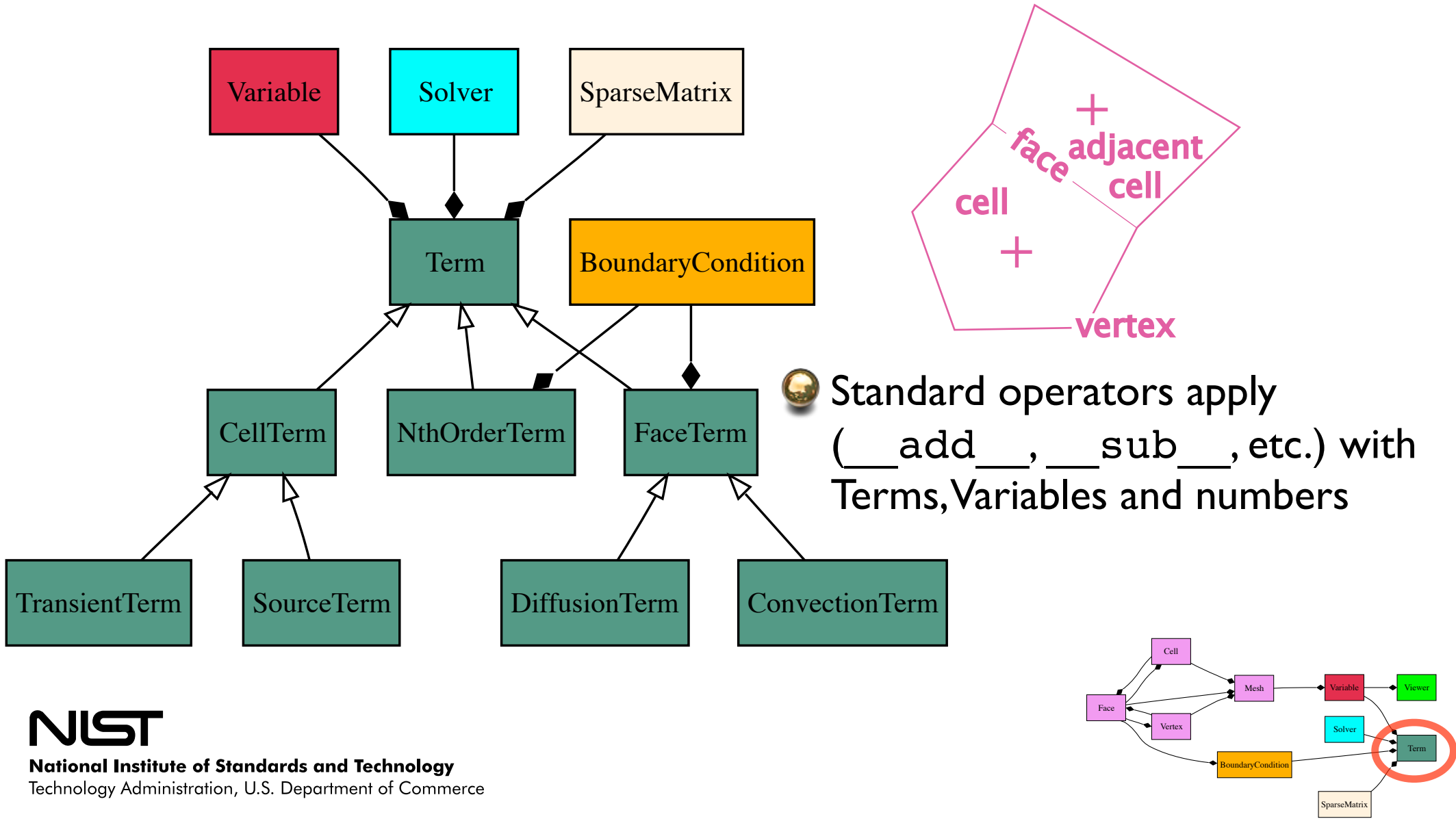


Design: Mesh

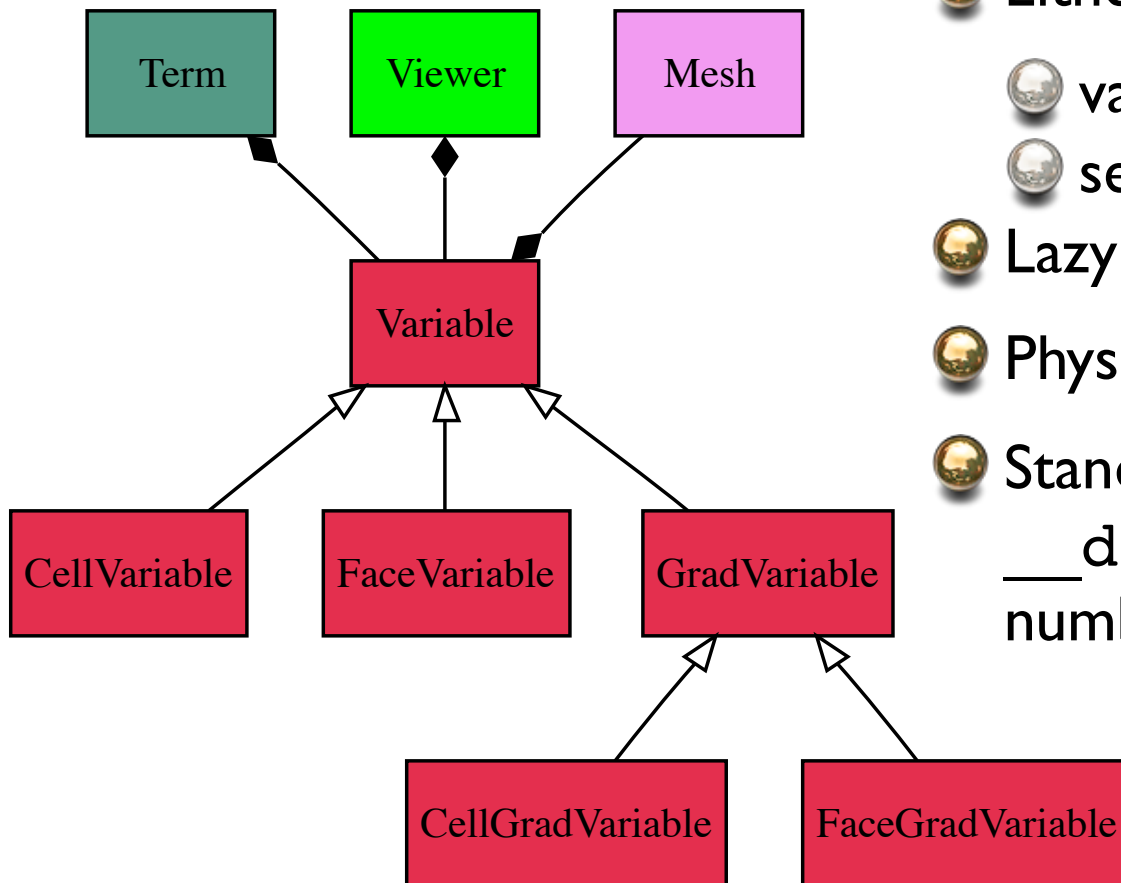


Design:Terms

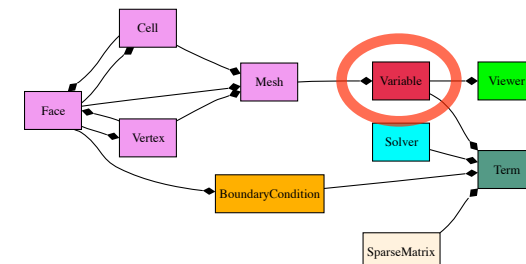
$$\underbrace{\frac{\rho\phi V - (\rho\phi)^{\text{old}} V}{\Delta t}}_{\text{transient}} = \underbrace{\sum_{\text{face}} [\Gamma A \vec{n} \cdot \nabla \phi]_{\text{face}}}_{\text{diffusion}} + \underbrace{\sum_{\text{face}} [\Gamma A \vec{n} \cdot \nabla \{\cdots\}]_{\text{face}}}_{n^{\text{th}} \text{ order diffusion}} + \underbrace{\sum_{\text{face}} [(\vec{n} \cdot \vec{u}) A \phi]_{\text{face}}}_{\text{convection}} + \underbrace{V S_{\phi}}_{\text{source}}$$



Design: Variables



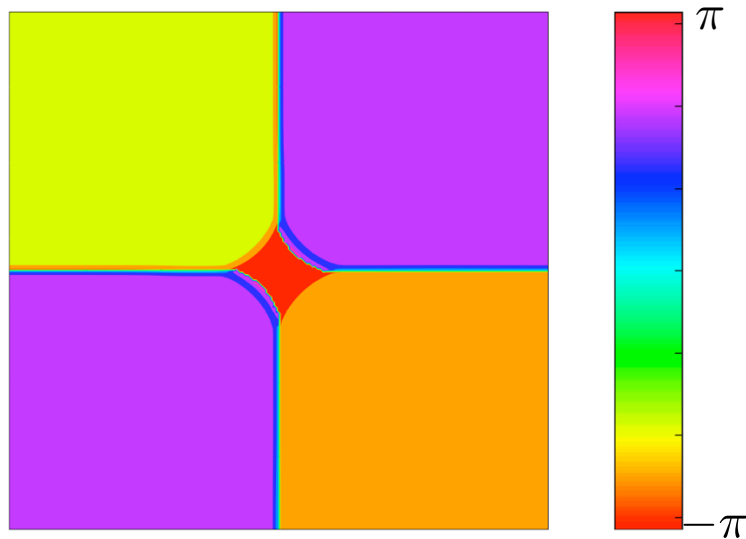
- Either:
 - variables set by assignment `[:]`
 - set by intermediate calculation
- Lazy evaluation
- Physical dimensions
- Standard operators apply (`__add__`, `__div__`, etc.) with Terms, Variables and numbers



Efficiency: Run Time Comparison

$$\underbrace{Q(\phi, \nabla\theta) \frac{\partial\phi}{\partial t}}_{\text{transient}} = \underbrace{\alpha^2 \nabla^2 \phi}_{\text{diffusion}} - \underbrace{\frac{\partial f}{\partial \phi} - \frac{\partial g}{\partial \phi} s |\nabla \phi| - \frac{\partial h}{\partial \phi} \frac{\epsilon^2}{2} |\nabla \phi|^2}_{\text{source}} \quad \underbrace{P(\phi, \nabla\theta) \frac{\partial\theta}{\partial t}}_{\text{transient}} = \underbrace{\nabla \cdot \left[h \epsilon^2 \nabla \theta + g s \frac{\nabla \theta}{|\nabla \theta|} \right]}_{\text{diffusion}}$$

Phase field equation - solved explicitly



Grain Orientation

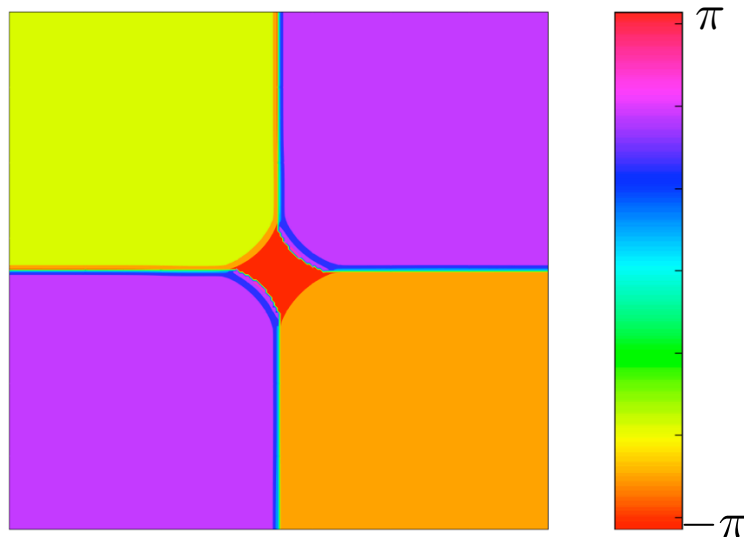
Comparison with hand tailored FORTRAN code (1800 lines) written specifically for grain impingement giving identical numerical results to FiPy (100 lines)

Efficiency: Run Time Comparison

$$\underbrace{Q(\phi, \nabla \theta) \frac{\partial \phi}{\partial t}}_{\text{transient}} = \underbrace{\alpha^2 \nabla^2 \phi}_{\text{diffusion}} - \underbrace{\frac{\partial f}{\partial \phi} - \frac{\partial g}{\partial \phi} s |\nabla \phi| - \frac{\partial h}{\partial \phi} \frac{\epsilon^2}{2} |\nabla \phi|^2}_{\text{source}}$$

$$\underbrace{P(\phi, \nabla \theta) \frac{\partial \theta}{\partial t}}_{\text{transient}} = \underbrace{\nabla \cdot \left[h \epsilon^2 \nabla \theta + g s \frac{\nabla \theta}{|\nabla \theta|} \right]}_{\text{diffusion}}$$

Orientation equation - solved implicitly



Grain Orientation

Comparison with hand tailored
FORTRAN code (1800 lines) written
specifically for grain impingement giving
identical numerical results to FiPy (100
lines)

Efficiency: Run Time Comparison

N	FORTTRAN (s)	FiPy (s)	Penalty
100	0.0008	0.282	$\times 353$
400	0.0037	0.402	$\times 109$
1600	0.02	0.963	$\times 48$
6400	0.19	4.04	$\times 21$
25600	1.20	20.2	$\times 17$
102400	4.43	81.0	$\times 18$

Penalty with pure Python and Numeric

How do we improve run times?

Efficiency: Run Time Comparison

N	FORTTRAN (s)	FiPy (s)	Penalty	FiPy <code>--inline</code> (s)	Penalty
100	0.0008	0.282	$\times 353$	0.230	$\times 288$
400	0.0037	0.402	$\times 109$	0.285	$\times 77$
1600	0.02	0.963	$\times 48$	0.566	$\times 28$
6400	0.19	4.04	$\times 21$	1.94	$\times 10$
25600	1.20	20.2	$\times 17$	9.05	$\times 8$
102400	4.43	81.0	$\times 18$	36.4	$\times 8$

Penalty with some C
inlining

```
>>> class SomeTerm
...
...     def _buildMatrix(self, var, boundaryConditions = (), dt = 1.):
...         from fipy.tools import inline
...         inline._optionalInline(self._buildMatrixIn, self._buildMatrixPy, L, var.getOld(), b, dt, coeffVectors)
...         return (L, b)
...
...     def _buildMatrixPy(self, L, oldArray, b, dt, coeffVectors):
...         ....
...
...     def _buildMatrixIn(self, L, oldArray, b, dt, coeffVectors):
...         inline._runInlineLoop1("""
...             b(i) += oldArray(i) * oldCoeff(i) / dt;
...             b(i) += bCoeff(i);
...             ....
```

`--inline` flag to toggle
between code variants

Efficiency: Profile, N = 102400

profile: 192 functions, 34241 calls, totaltime = 35960 ms

file	line	function	ncalls	self	percall	total	total percall
term.py	87	solve	20	70.109	3.505	35287.704	1764.385
binaryTerm.py	63	_buildMatrix	50	74.755	1.495	31587.505	631.750
variable.py	245	getValue	2610	16.754	0.006	15798.149	6.053
variable.py	342	_refresh	4830	47.114	0.010	15794.906	3.270
inline.py	6	_optionalInline	150	11.231	0.075	15463.175	103.088
diffusionTerm.py	350	_buildMatrix	40	555.884	13.897	14973.796	374.345
cellTerm.py	145	_buildMatrix	50	207.717	4.154	14883.878	297.678
cellTerm.py	125	_buildMatrixIn	50	1302.381	26.048	14587.725	291.754

Majority of time spent in Term.solve()

Callees

ncalls	%time	file	line	function
20	89.514198	binaryTerm.py	63	_buildMatrix
20	7.064664	linearPCGSolver.py	72	_solve
20	2.944877	term.py	69	_getResidual

Building is order N

Solving is higher order

90% of time building the matrix not solving

Efficiency: Profile, N = 102400

profile: 192 functions, 34241 calls, totaltime = 35960 ms								
file	line	function	ncalls	self	percall	total	▲	total percall
term.py	87	solve	20	70.109	3.505	35287.704		1764.385
binaryTerm.py	63	_buildMatrix	50	74.755	1.495	31587.505		631.750
variable.py	245	getValue	2610	16.754	0.006	15798.149		6.053
variable.py	342	_refresh	4830	47.114	0.010	15794.906		3.270
inline.py	6	_optionalInline	150	11.231	0.075	15463.175		103.088
diffusionTerm.py	350	_buildMatrix	40	555.884	13.897	14973.796		374.345
cellTerm.py	145	_buildMatrix	50	287.717	4.154	14883.878		297.678
cellTerm.py	125	_buildMatrixIn	50	1302.381	26.048	14587.725		291.754

44% of time spent calculating variables of which
58% is spent calculating non-inline variables

Efficiency: Variable Operations

```
>>> from fipy.meshes.gridID import GridID
>>> nx = 2
>>> mesh = GridID(nx = nx)
>>>
>>> from RandomArray import random
>>> from fipy.variables.cellVariable import CellVariable
>>> vars = [CellVariable(mesh = mesh, value = random(nx)) for i in range(5)]
>>> opVar = vars[0] * (vars[1] * vars[2] + vars[3] * vars[4])
>>>
>>> opVar
(CellVariable(value = ..., mesh = ...) * ((CellVariable(..) * CellVariable(..) +
(CellVariable(..) * CellVariable(..)))
>>> print opVar
[ 0.93079731, 0.33666286,]
>>> vars[0].setValue(random(nx))
>>> print opVar
[ 0.81214315, 0.24010101,]
```

Efficiency: Variable Operations

```
>>> from fipy.meshes.gridID import GridID
>>> nx = 2
>>> mesh = GridID(nx = nx)
>>>
>>> from RandomArray import random
>>> from fipy.variables.cellVariable import CellVariable
>>> vars = [CellVariable(mesh = mesh, value = random(nx)) for i in range(5)]
>>> opVar = vars[0] * (vars[1] * vars[2] + vars[3] * vars[4])
>>>
>>> opVar
(CellVariable(value = ..., mesh = ...) * ((CellVariable(..) * CellVariable(..) +
(CellVariable(..) * CellVariable(..)))
>>> print opVar
[ 0.93079731, 0.33666286,]
>>> vars[0].setValue(random(nx))
>>> print opVar
[ 0.81214315, 0.24010101,]
```

4 operations

1 operation due to lazy evaluation

Efficiency: Variable Operations

```
>>> from fipy.meshes.gridID import GridID
```

```
>>> nx = 2
```

```
>>> mesh = GridID(nx = nx)
```

```
>>>
```

```
>>> from RandomArray import random
```

```
>>> from fipy.variables.cellVariable import CellVariable
```

```
>>> vars = [CellVariable(mesh = mesh, value = random(nx)) for i in range(5)]
```

```
>>> opVar = vars[0] * (vars[1] * vars[2] + vars[3] * vars[4])
```

```
>>>
```

```
>>> opVar
```

```
(CellVariable(value = ..., mesh = ...) * ((CellVariable(..) * CellVariable(..)) +  
(CellVariable(..) * CellVariable(..))))
```

```
>>> print opVar
```

```
[ 0.93079731, 0.33666286,]
```

```
>>> vars[0].setValue(random(nx))
```

```
>>> print opVar
```

```
[ 0.81214315, 0.24010101,]
```

```
>>> inline._runInlineLoop1("""  
...     opVar(i) = v0(i) * (v1(i) * v2(i) + v3(i) * v4(i));  
...     """, v0 = vars[0],
```

```
.....
```

Proposed automated C inlining of binary and unary variable operators by forming combined strings of C code

Efficiency: Variable Operations

```
>>> from fipy.meshes.gridID import GridID
```

```
>>> nx = 2
```

```
>>> mesh = GridID(nx = nx)
```

```
>>>
```

```
>>> from RandomArray import random
```

```
>>> from fipy.variables.cellVariable import CellVariable
```

```
>>> vars = [CellVariable(mesh = mesh, value = random(nx)) for i in range(5)]
```

```
>>> opVar = vars[0] * (vars[1] * vars[2] + vars[3] * vars[4])
```

```
>>>
```

```
>>> opVar
```

```
(CellVariable(value = ..., mesh :
```

```
(CellVariable(..) * CellVariable(
```

```
>>> print opVar
```

```
[ 0.93079731, 0.33666286,]
```

```
>>> vars[0].setValue(random(
```

```
>>> print opVar
```

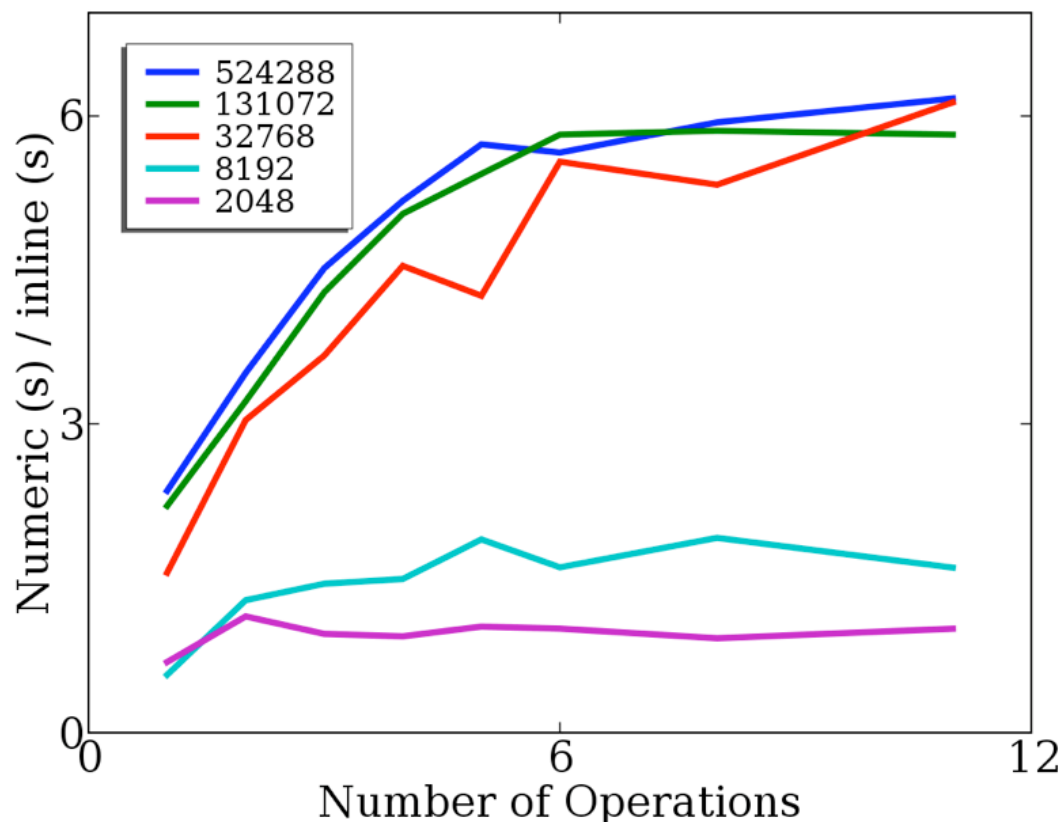
```
[ 0.81214315, 0.24010101,]
```

```
>>> inline._runInlineLoop1("""
```

```
...     opVar(i) = v0(i) * (v1(i) * v2(i) + v3(i) * v4(i));
```

```
...     """, v0 = vars[0],
```

```
.....
```

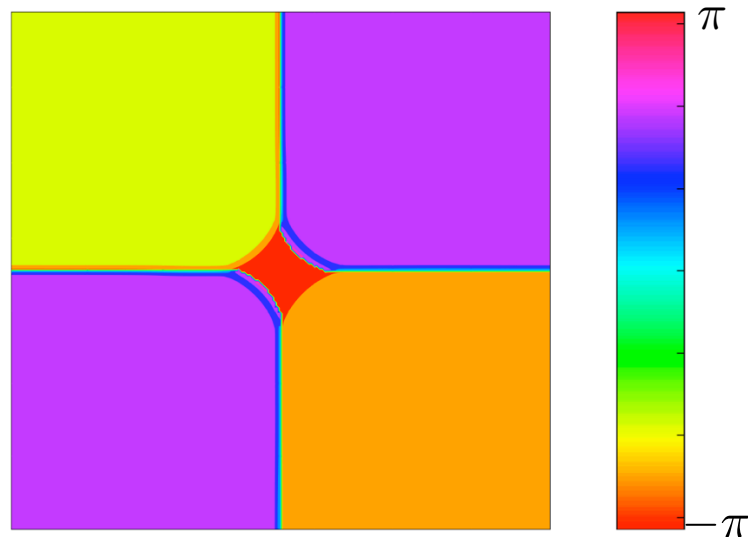


Considerable speed up when C inlining replaces Numeric for multiple operations

Efficiency: Memory Usage

N	FORTTRAN (KB)	FiPy (KB)	Penalty
100	812	30068	$\times 37$
400	884	31260	$\times 35$
1600	1080	34280	$\times 32$
6400	1920	47864	$\times 25$
25600	5240	91872	$\times 18$
102400	18480	269332	$\times 15$

Terrible
overall
memory use



Grain Orientation

Comparison with hand tailored
FORTRAN code (1800 lines) written
specifically for grain impingement giving
identical numerical results to FiPy (100
lines)

Efficiency: Memory Usage

N	FORTTRAN (KB)	FiPy (KB)	Penalty	Python (%)	Mesh (%)
100	812	30068	$\times 37$	13	40
400	884	31260	$\times 35$	12	39
1600	1080	34280	$\times 32$	11	40
6400	1920	47864	$\times 25$	8	36
25600	5240	91872	$\times 18$	4	33
102400	18480	269332	$\times 15$	1	32

How do we improve memory usage?

Memory usage breakdown

 ~30-40 % Mesh

 ~40-50 % Variable (estimate)

 sparse matrix, deleted after build.

 other Numeric arrays

Efficiency: Memory Usage

N	FORTTRAN (KB)	FiPy (KB)	Penalty	Python (%)	Mesh (%)
100	812	30068	$\times 37$	13	40
400	884	31260	$\times 35$	12	39
1600	1080	34280	$\times 32$	11	40
6400	1920	47864	$\times 25$	8	36
25600	5240	91872	$\times 18$	4	33
102400	18480	269332	$\times 15$	1	32

How do we improve memory usage?

- More efficient caching of mesh arrays
- Specialized grid meshes
- Considerable memory usage improvement

Efficiency: Memory Usage

N	FORTTRAN (KB)	FiPy (KB)	Penalty	Python (%)	Mesh (%)
100	812	30068	$\times 37$	13	40
400	884	31260	$\times 35$	12	39
1600	1080	34280	$\times 32$	11	40
6400	1920	47864	$\times 25$	8	36
25600	5240	91872	$\times 18$	4	33
102400	18480	269332	$\times 15$	1	32

How do we improve memory usage?

```
>>> opVar = vars[0] * (vars[1] * vars[2] + vars[3] * vars[4])
>>>
>>> opVar
(CellVariable(value = ..., mesh = ...) * ((CellVariable(..) * CellVariable(..)) +
(CellVariable(..) * CellVariable(..))))
```

Do not store intermediate
values for operator
variables

- Lose some benefits (but not all) of lazy evaluation
- Will be implicit for C inlined variable operations
- Many variables recalculated every time step anyway
- Considerable memory usage improvement

Sparse Matrices / Linear Solvers

🏆 What are the options?






🥈 Pysparse

🥈 scipy.linalg

🥈 Pytrilinos

Pysparse







Pysparse

-  Straightforward to install and test
-  Adequate documentation
-  Sparse matrices interact with solvers
-  Standard solvers available
-  Subsequently wrapped sparse matrix module for standard python operations in `_SparseMatrix` class

```
>>> from fipy.tools.sparseMatrix import _SparseMatrix
>>> L = _SparseMatrix(size = 3)
>>> L
---      ---      ---
---      ---      ---
---      ---      ---
>>> L.put((2., 2., 2.), (0, 1, 2), (0, 1, 2))
>>> L.put((-1., -1.), (0, 1), (1, 2))
>>> L.put((-1., -1.), (1, 2), (0, 1))
>>> L
2.000000  -1.000000   ---
-1.000000  2.000000  -1.000000
---      -1.000000  2.000000
>>> L * L
5.000000  -4.000000  1.000000
-4.000000  6.000000  -4.000000
1.000000  -4.000000  5.000000
>>> from fipy.tools import numerix
>>> x = numerix.zeros(3, 'd')
>>> from fipy.solvers.linearLUSolver import LinearLUSolver
>>> LinearLUSolver()._solve(L, x, numerix.array((0, 0, 1)))
>>> print x
[ 0.25, 0.5 , 0.75,]
```

scipy.linalg

scipy.linalg

-  Straightforward to install and test
-  No useful documentation
-  Sparse Matrices?
-  Some scipy solvers implemented in FiPy
 -  wrapped with LinearScipyLUSolver and linearScipyGMRESSolver
-  Currently requires conversion of sparse matrix to numeric arrays

Pytrilinos

- 🌟 Pytrilinos
 - 🌐 Installation?
 - 🌐 Documentation?
 - 🌐 Mailing list?

Future Work

- Efficiency improvements
- Adaptive meshes
- Algebraic multigrid
- Cell-centered finite volume
- Spectral methods
- Repair/improve support for physical dimensions

Summary

- Cross-platform, Open Source code for solving phase transformation problems
- Capable of solving multivariate, coupled, non-linear PDEs
- Extensive documentation, dozens of examples, hundreds of tests
- Python syntax both easy to learn and powerful
- Object-oriented structure easy to adapt to unique problems
- Slower to run than hand-tailored FORTRAN or C...
- ...but *much* faster to write

www.ctcms.nist.gov/fipy/

Acknowledgements

- Alex Mont – Montgomery Blair High School
- John Dukovic – Applied Materials
- Daniel Josell – NIST Metallurgy Division
- Tom Moffat – NIST Metallurgy Division
- Steve Langer – NIST Information Technology Laboratory
- Andrew Reid – NIST Materials Science and Engineering Laboratory
- Edwin García – NIST Materials Science and Engineering Laboratory
- Daniel Lewis – GE Ceramic and Metallurgy Technologies
- Yosi Shacham-Diamand - Tel Aviv University