FiPy: Partial Differential Equations with Python

Many existing partial differential equation solver packages focus on the important, but arcane, task of numerically solving the linearized set of algebraic equations that result from discretizing a set of PDEs. Many researchers, however, need something higher level than that.

Partial differential equations (PDEs) are ubiquitous to the mathematical description of physical phenomena: they describe the relationships between functions of more than one independent variable and partial derivatives with respect to those variables. Typical examples in the physical sciences describe the evolution of a field in time as a function of its value in space, such as in wave propagation, heat flow, or fluid dynamics.

Although PDEs are relevant throughout the sciences, we focus our attention here on materials. Materials science studies the relationships between the properties and microstructures of materials and how both characteristics are affected by (and how they affect) the processing of those materials. As the science of materials measurement advances, it has become increasingly difficult to determine what a measurement "means" or, alternatively, what measurable quantities are appropriate to shed light on the phenomena of interest. These phenomena are often at the nanoscale, with highly nonclassical behaviors that involve complex interactions of multiple systems. Teasing out the quantity to be determined from an experiment's "signal" often requires the solution of a mathematical model in the form of a set of PDEs. In most cases, those PDEs don't admit an analytical solution, so researchers use computers to obtain numerical solutions to practical problems. The goal of computational materials science is to apply computational methods to explain and predict properties such as experimental microstructures and phenomena (see Figure 1).

Lots of numerical PDE solvers are available today, and FiPy (www.ctcms.nist.gov/fipy) won't be the last; indeed, it isn't even the first discussion of a Python-based PDE solver in these pages.[1,2] Many existing PDE solver packages focus on the important, but relatively arcane, task of numerically solving the linearized set of algebraic equations that result from discretizing a set of PDEs. However, many, if not most, researchers would prefer to avoid reckoning with such details and work at a higher level of abstraction. Specifically, they have the physical knowledge to describe their model and can apply the calculus needed to obtain the appropriate governing conditions, but when faced with rendering those governing equations on a computer, their skills (or time) are limited to a straightforward implementation

1521-9615/09/$25.00 © 2009 IEEE
Copublished by the IEEE CS and the AIP

Jonathan E. Guyer, Daniel Wheeler, and James A. Warren
US National Institute of Standards and Technology

of explicit finite differences on uniform square grids. Of the PDE solver packages that focus at an appropriately high level, many are proprietary, expensive, and difficult to customize. For scientists trying to do it themselves, a search of the Internet turns up a multitude of codes with promising names and abstracts, but they're generally unapproachable by those who don't already know the answer to the question they're asking. Consequently, scientists spend considerable resources repeatedly developing limited tools for specific problems. In an effort to break this cycle, while still meeting our own research goals, we developed FiPy.

FiPy plays a critical role in our materials research on electrodeposition,[3] alloy interdiffusion,[4] surface wetting, and photovoltaics. An active user community has also applied FiPy to topics as diverse as solar irradiation of soil,[5] water percolation through peat bogs, and mRNA reaction-diffusion in fruitfly embryos, as well as a host of the materials science applications we originally envisioned. FiPy has proven to be a powerful teaching tool as well. Several of our colleagues in academia have experimented with it for teaching computational materials science at both the undergraduate and graduate levels (see, for example, www.nanohub.org/tools/vkmlpsgg). In this article, we outline the design of FiPy and present a few fully functional examples of its ability to solve coupled sets of PDEs.

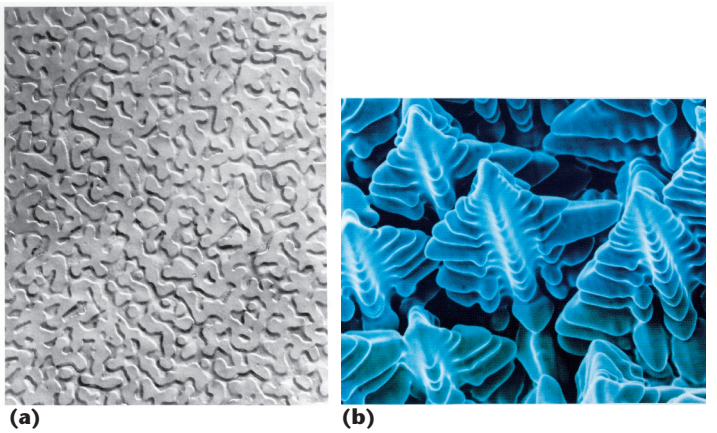

Figure 1. Experimental microstructures. (a) Spinodal decomposition in glass, which is important for polymers, hardened metal alloys, tempered glasses, and permanent magnets, and (b) dendrites formed during resolidification after welding nickel-based "superalloys" to jet-engine turbine blades. (Source: [1a] W. Haller, US Nat'l Inst. Standards and Technology, and [1b] S.A. David at Oak Ridge Nat'l Laboratory; used with permission.)

Scripting

A rather common scenario in the development of scientific codes is that the first draft hard-codes all the problem parameters. After a few (hundred) iterations of recompiling and relinking the application to explore parameter changes, the researcher adds code to read an input file containing a list of numbers. Eventually, we reach a point where it's impossible to remember which parameter comes in which order or what physical units are required, so the researcher adds code 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. However, very few scientists have actually studied computer science or 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 doing research.

In contrast, several powerful scripting languages, such as Tcl, Java, Python, Ruby, and even the venerable Basic, have open source interpreters that we can embed 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 chose 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. Although runtime performance is unquestionably important, many scientific codes are run relatively little in proportion to the time spent developing them. If we could develop a code in a day instead of a month, it might not matter if it takes a week to run instead of a day before being abandoned in favor of a new physical problem after the publication of one or two papers. Furthermore, a variety of mechanisms for diagnosing and optimizing those portions of a code that are actually time-critical exist, which is more attractive than attempting to optimize all of it by using a language that's 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 a high-level scripting language and only translates to low-level compiled code those portions of the code that prove inefficient.

Although several scripting languages might have worked, we selected Python for FiPy's implementation because it's

- 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;
- actively used and extended by other scientists and mathematicians (such as NumPy, http://numpy.scipy.org; SciPy, www.scipy.org; ScientificPython, http://dirac.cnrs-orleans.fr/plone/software/scientificpython; and PySparse, http://pysparse.sourceforge.net); and
- easily integrated with low-level languages such as C or Fortran (such as Weave and Blitz, www.scipy.org/Weave; PyRex, www.cosc.canterbury.ac.nz/~greg/python/Pyrex/; Cython, www.cython.org; SWIG, www.swig.org; and f2py, http://cens.ioc.ee/projects/f2py2e/).

Starting very early in FiPy's development, we made a commitment to use several professional coding practices that remain rare for most scientific applications—for example, FiPy's code repository and bug tracker are publicly accessible at www.matforge.org/fipy. Based on feedback on our mailing list (www.ctcms.nist.gov/fipy/mail), we've been able to refine both the documentation and the interface in subsequent releases, and we've incorporated patches and feature enhancements submitted by our users. An automated test suite (www.matforge.org/fipy/build) exercises both the low-level functionality as well as all of the examples included in our user manual. This test suite has prevented us from inadvertently breaking one part of the code when working on another and has allowed us to undertake major code refactoring on more than one occasion. Although none of these practices are unique to Python programming, they are prevalent in that community, and a number of Python tools make it easy to develop "good" software.

Numerical Approach

We designed FiPy to solve an arbitrary number of PDEs of the form

$$\underbrace{\frac{\partial(\rho\phi)}{\partial t}}_{\text{transient}} - \underbrace{\left[\nabla\cdot(\Gamma_i\nabla)\right]^n\phi}_{\text{diffusion}} - \underbrace{\nabla\cdot(\mathbf{u}\phi)}_{\text{convection}} - \underbrace{S_\phi}_{\text{source}} = 0, \quad (1)$$

where one equation is identified with each solution variable ϕ. The transient term represents the time rate of change of ϕ, with a rate factor ρ. The diffusion term represents the tendency of nonuniformities in ϕ to smooth out (material flows "down" gradients) with a diffusivity Γ_i. The shorthand $[\nabla\cdot(\Gamma_i\nabla)]^n\phi$ in the diffusion term represents any of the family of terms $\nabla\cdot(\Gamma_1\nabla\phi)$, $\nabla\cdot(\Gamma_1\nabla[\nabla\cdot(\Gamma_2\nabla\phi)])$, and so on. The convection term describes the "blowing" of ϕ by a velocity field $\mathbf{u}$. Finally, a source term represents any source or sink that injects or removes ϕ from a point in space. Equation 1 doesn't address all applications of PDEs, but it so far covers everything we've tried to do in materials science.

To solve Equation 1, we cast it in integral (weak) form as

$$\underbrace{\int_V \tau\frac{\partial(\rho\phi)}{\partial t}dV}_{\text{transient}} - \underbrace{\int_S \tau\Gamma_n(\mathbf{n}\cdot\nabla\cdots)dS}_{\text{diffusion}} - \underbrace{\int_S \tau(\mathbf{n}\cdot\mathbf{u})\phi\,dS}_{\text{convection}} - \underbrace{\int_V \tau S_\phi\,dV}_{\text{source}} = 0, \quad (2)$$

where the integrals are performed over an arbitrary subvolume V with boundary S. The test function τ depends on the characteristics of the volume element and the specific numerical discretization scheme. FiPy uses the finite volume method (FVM), which is widely used in the field of computational fluid dynamics (CFD), to reduce the model equations to a form tractable to linear solvers. One way to understand the FVM is in comparison with two better known discretization techniques: the finite element method (FEM) casts the PDE in a weak form and discretizes it with a test function that smoothly weights each solution's volume. If the FEM is constructed with a test function of 1, it reduces to the FVM; if the FVM is solved on a uniform Cartesian grid, it becomes the finite difference method (although the FDM is usually performed by discretizing Equation 1 instead of Equation 2). The FVM allows the use of unstructured grids but at the cost of greater computational complexity than the FDM. On the other hand, the FVM is easier to implement, with less computational complexity, than the FEM. Its solutions, however, are prone to greater errors, particularly with very nonorthogonal meshes. For many problems (such as CFD), these make up a reasonable set of compromises.

Objects

FiPy is based on three fundamental Python classes: `Mesh`, `Variable`, and `Term`.

A `Mesh` object represents the domain of interest. FiPy contains many different specific mesh classes to describe different geometries. FiPy can also use the open source Gmsh (www.geuz.org/gmsh) meshing tool to represent more complicated geometries. Conceptually, a `Mesh` is composed of `Cells`, each `Cell` is defined by its bounding

Faces, and each Face is defined by its bounding Vertex objects.

A Variable object represents a quantity or field that can change during the problem's evolution. A MeshVariable object is a Variable that holds a field of values, distributed over a Mesh. A CellVariable is defined at the Cell centers of the Mesh, and a FaceVariable is defined at the Faces between Cells. A MeshVariable describes the values of the field ϕ, but it isn't concerned with their geometry or topology; the Mesh takes that role.

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

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

doesn't 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 isn't calculated until we need it ("lazy evaluation"):

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

A Term object represents any of the terms in Equations 1 or 2 or any linear combination of such terms. The first three terms are represented in FiPy as TransientTerm(coeff=rho), DiffusionTerm(coeff=(Gamma1, Gamma2, ...)), and <Specific>ConvectionTerm(coeff=u). FiPy supports central difference, upwind, exponential, hybrid, power law, and van Leer weightings for the ConvectionTerm, based on the Péclet number.

Any terms that we can't write in one of the previous forms is considered a source. In FiPy, we write an explicit source essentially as it appears in mathematical form, so we might write $3\kappa^2 + \phi \sin\theta$ as `3 * kappa**2 + b * sin(theta)`. If, however, the source depends on the variable that's being solved for, it might be advantageous to linearize the source and cast part of it as an implicit source term, so we might write $3\kappa^2 + \phi \sin\theta$ as `3 * kappa**2 + ImplicitSourceTerm(coeff=sin(theta))`.

It's important to realize that, even though an expression might superficially resemble one of those just shown, if the dependent variable for that PDE doesn't appear in the appropriate place, then we should treat that term as a source, so we would write a term $\nabla \cdot (D_2 \nabla \xi)$ as `(D2 * xi.getFaceGrad()).getDivergence()` in an equation that solves for ϕ rather than ξ.

Finally, an equation is simply a collection of Terms, formed by adding or equating them. For example,

```
eq = (TransientTerm()
 == DiffusionTerm(D)
 + ConvectionTerm(u)
 + ImplicitSourceTerm(sin(theta)) + 3)
```

represents $\partial\phi/\partial t = \nabla \cdot D\nabla\phi + \nabla \cdot \phi\mathbf{u} + \phi \sin\theta + 3$.

Beyond these three fundamental classes of Mesh, Variable, and Term, FiPy is composed of several related classes. A Term encapsulates the contributions to the SparseMatrix that define an equation's solution. We use BoundaryConditions to describe the conditions on the boundaries of the Mesh, and each Term interprets the BoundaryConditions as necessary to modify the SparseMatrix. An equation constructed from Terms can apply a unique Solver to invert its SparseMatrix in the most expedient and stable fashion. FiPy provides a common wrapper interface to shield the user from the implementation details of the SparseMatrix solver libraries with Python interfaces. These range from PySparse, which tends to be easy to build and install, to PyTrilinos (http://trilinos.sandia.gov), which offers both parallel solvers and an impressive collection of preconditioners. Finally, at any point during the solution, the user can invoke a Viewer to display the values of the solved Variables.

When the user invokes a FiPy equation's solve() method, FiPy queries each Term for its contributions to the SparseMatrix and right-hand-side vector of the set of linear algebraic discretization expressions. The coefficient of each Term could either be a constant number or obtained by lazy evaluation of a Variable expression that depends on the current value of the solution variables. In addition to the coefficient's value, each Term queries the Mesh for geometric factors such as Cell volume and Face area and orientation. For a Mesh composed of different polygonal (or polyhedral) elements, FiPy calculates and stores these geometric factors, but for commonly used, regular grids, FiPy can efficiently

calculate these values on the fly, saving a considerable amount of memory. In either case, the user need not be concerned with what amounts to nothing more than tedious bookkeeping.

By default, `DiffusionTerm` and `ConvectionTerm` are implicit, but an `ExplicitDiffusionTerm` and `ExplicitUpwindConvectionTerm` are available, primarily for didactic purposes. In some cases, semi-explicit approaches can yield larger time steps than a naïve, fully explicit implementation, but with better accuracy than a fully implicit treatment. For example, on a uniform, one-dimensional grid with a constant coefficient D, we can discretize the expression $\partial u/\partial t = \nabla \cdot D\nabla u$ at each point j as

$$\frac{u_j^{new} - u_j^{old}}{\Delta t} = \alpha D \frac{u_{j+1}^{new} - 2u_j^{new} + u_{j-1}^{new}}{\Delta x^2} + (1-\alpha)D \frac{u_{j+1}^{old} - 2u_j^{old} + u_{j-1}^{old}}{\Delta x^2}.$$

When $\alpha = 0$, u advances by a fully explicit forward Euler time step; when $\alpha = 1$, it takes a fully implicit backward Euler time step. We obtain the semi-explicit Crank-Nicolson scheme when $\alpha = ½$. The FiPy implementation of this scheme is as simple as

```
eq = (TransientTerm()
 == ImplicitDiffusionTerm(alpha*D)
 + ExplicitDiffusionTerm((1-alpha)*D))
```

The solution step is left largely up to the user, who calls an equation's `solve()` method with the `CellVariable` to be solved for the `BoundaryConditions`, the time step, and the matrix `Solver`. FiPy uses a preconditioned conjugate-gradient solver by default, unless any `Term` of the equation indicates that it produces an asymmetric matrix (such as most `ConvectionTerms`), in which case, FiPy falls back on lower-upper (LU) matrix decomposition. Users can always override these defaults, providing a solver and preconditioner of their choosing.

Examples

To give a flavor of working with FiPy, we present some fully functional example scripts for problems of interest in materials science.

Spinodal Decomposition

The spinodal decomposition phenomenon displayed in Figure 1a is a spontaneous separation of an initially homogeneous mixture into two distinct regions of different properties (spin up/spin down, component A/component B). It's a "barrierless" phase separation process, such that under the right thermodynamic conditions, any fluctuation, no matter how small, will tend to grow. This is in contrast to nucleation, in which a fluctuation must exceed some critical magnitude before it will survive and grow. We can describe spinodal decomposition via the Cahn-Hilliard equation[6,7] (also known as "conserved Ginsberg-Landau" or "model B" of Hohenberg and Halperin):

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

where ϕ is a conserved order parameter, possibly representing alloy composition or spin. The double-well free energy function $f = (a^2/2)\phi^2(1 - \phi)^2$ penalizes states with intermediate values of ϕ between 0 and 1. The gradient energy term $\varepsilon^2\nabla^2\phi$, on the other hand, penalizes sharp changes of ϕ. These two competing effects result in the segregation of ϕ into domains of 0 and 1, separated by abrupt, but smooth, transitions. The parameters a and ε determine the relative weighting of the two effects, and D is a rate constant.

We can simulate this process in FiPy with a simple script:

```
from fipy import *
```

(Note that all of NumPy's functionality is imported along with FiPy, although much is augmented for FiPy's needs.)

```
mesh = Grid2D(nx=1000, ny=1000,
              dx=0.25, dy=0.25)
phi = CellVariable(name=r"$\phi$",
                   mesh=mesh)
```

We start the problem with random fluctuations about $\phi = ½$:

```
phi.setValue(
 GaussianNoiseVariable(mesh=mesh,
                       mean=0.5,
                       variance=0.01))
```

FiPy doesn't plot or output anything unless you tell it to:

```
viewer = Viewer(vars=(phi,),
                datamin=0., datamax=1.)
```

For FiPy, we must perform the partial derivative $\partial f/\partial\phi$ manually and then put the equation in the canonical form of Equation 1 by decompos-

ing the spatial derivatives so that each `Term` is of a single, even order:

$$\frac{\partial \phi}{\partial t} = \nabla \cdot Da^2[1-6\phi(1-\phi)]\nabla\phi - \nabla \cdot D\nabla\varepsilon^2\nabla^2\phi \ .$$

FiPy would automatically interpolate `D * a**2 * (1 - 6 * phi * (1-phi))` onto the `Faces`, where the diffusive flux is calculated, but we obtain somewhat more accurate results by performing a linear interpolation from `phi` at `Cell` centers to `PHI` at `Face` centers. Some problems benefit from nonlinear interpolations, such as harmonic or geometric means, and FiPy makes it easy to obtain these, too:

```
PHI = phi.getArithmeticFaceValue()
D = a = eps = 1.
eq = (TransientTerm()
 == DiffusionTerm(coeff=
    D * a**2 * (1 - 6 * PHI * (1-PHI)))
 - DiffusionTerm(coeff=(D, eps**2)))
```

Because the evolution of a spinodal microstructure slows with time, we use exponentially increasing time steps to keep the simulation "interesting." The FiPy user always has direct control over the problem's evolution:

```
dexp = -5
elapsed = 0.
while elapsed < 1000.:
  dt = min(100, exp(dexp))
  elapsed += dt
  dexp += 0.01
  eq.solve(phi, dt=dt)
  viewer.plot()
```

The spinodal structure's coarsening with time, as seen in Figure 2, is driven by reduction of the interfacial energy between areas of ϕ = 0 and ϕ = 1, which is a function of interfacial curvature. Simulations in 2D can be misleading when compared with experimental images that are sections through 3D microstructures, such as Figure 1a. For instance, there's no 2D analog for saddle points on 3D surfaces, which have zero curvature. FiPy makes it easy to solve the same problem in 3D simply by changing the mesh declaration

```
mesh = Grid3D(nx=100, ny=100, nz=100,
              dx=0.25, dy=0.25, dz=0.25)
```

illustrated in Figure 3. No other changes are required (many other PDE solvers require manually

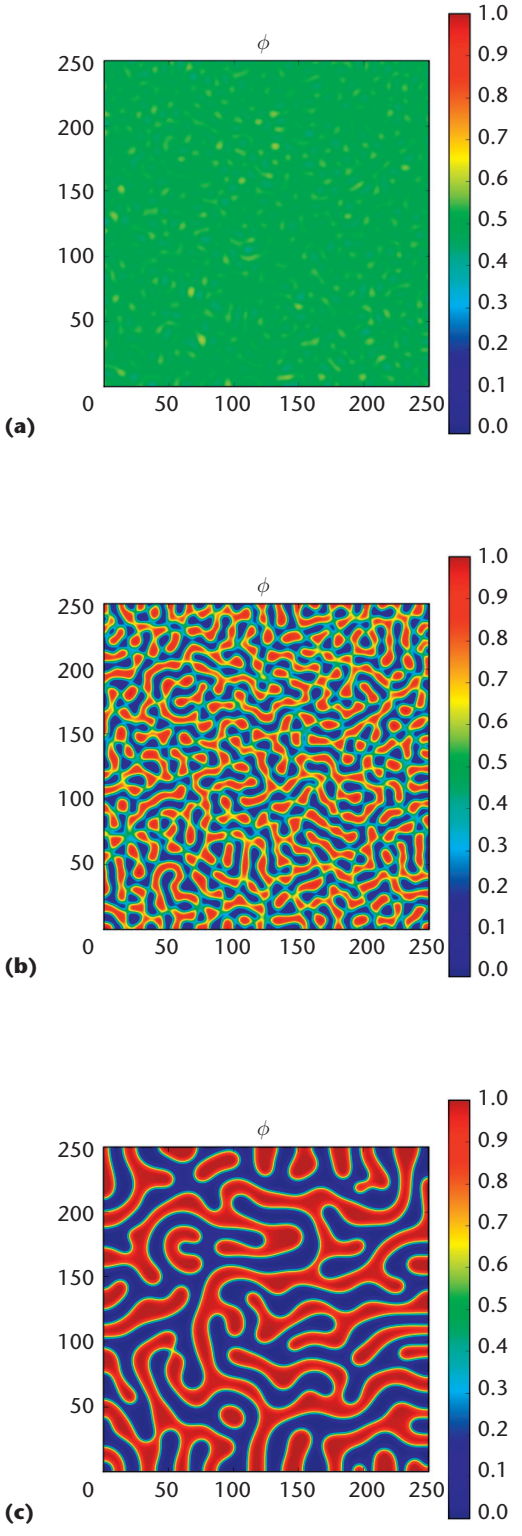

Figure 2. The spinodal decomposition's evolution in 2D. From (a) t = 30 to (b) t = 100 to (c) t = 1,000, we see that initially random fluctuations split into separate domains with a steadily increasing characteristic wavelength.

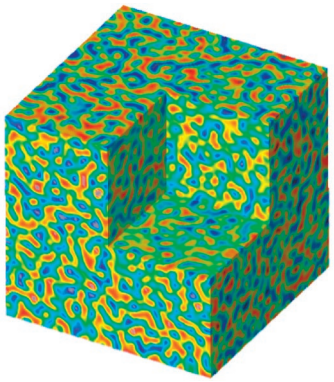

Figure 3. Spinodal decomposition in 3D. This snapshot is qualitatively similar to the 2D case, but the more complicated nature of curvature in 3D leads to changed dynamics of phase separation.

changing the ∇ operator from $(\partial/\partial x)\,\hat{i} + (\partial/\partial y)\,\hat{j}$ to $(\partial/\partial x)\,\hat{i} + (\partial/\partial y)\,\hat{j} + (\partial/\partial z)\,\hat{k}$).

Compared with FEM simulations of air flow around airplanes in flight or of combustion in an engine block, computational materials science is typically simulated on simple domains (squares, cubes, and so forth). Nonetheless, using Gmsh, we can do something a little more elaborate and solve the Cahn-Hilliard problem on the surface of a sphere. Although Gmsh isn't scriptable in Python, FiPy provides the facility for generating meshes described in Gmsh's scripting language (as shown between triple quotes in the argument to GmshImporter2DIn3DSpace):

```
mesh = GmshImporter2DIn3DSpace("""
radius = 5.0;
cellSize = 0.3;

// create inner 1/8 shell
Point(1) = {0, 0, 0, cellSize};
Point(2) = {-radius, 0, 0, cellSize};
Point(3) = {0, radius, 0, cellSize};
Point(4) = {0, 0, radius, cellSize};

Circle(1) = {2, 1, 3};
Circle(2) = {4, 1, 2};
Circle(3) = {4, 1, 3};
Line Loop(1) = {1, -3, 2};
Ruled Surface(1) = {1};

// create remaining 7/8 inner shells
t1[] = Rotate {{0,0,1},{0,0,0},Pi/2}
       {Duplicata{Surface{1};}};
t3[] = Rotate {{0,0,1},{0,0,0},Pi*3/2}
       {Duplicata{Surface{1};}};
t4[] = Rotate {{0,1,0},{0,0,0},-Pi/2}
       {Duplicata{Surface{1};}};
t5[] = Rotate {{0,0,1},{0,0,0},Pi/2}
       {Duplicata{Surface{t4[0]};}};
t6[] = Rotate {{0,0,1},{0,0,0},Pi}
       {Duplicata{Surface{t4[0]};}};
t7[] = Rotate {{0,0,1},{0,0,0},Pi*3/2}
       {Duplicata{Surface{t4[0]};}};

// create entire inner and outer
// shell Surface
Loop(100)={1,t1[0],t2[0],t3[0],
           t7[0],t4[0],t5[0],t6[0]};
""").extrude(extrudeFunc=
           lambda r: 1.1 * r)
```

which is illustrated in Figure 4. Although it took the three of us roughly a week to figure out how to make this nominally isotropic spherical mesh, as it was our first experience with a complicated 3D mesh, no changes in FiPy were required in order to use it.

The `Cells` of this `Mesh` have finite thickness, and their outer `Faces` are slightly larger than their inner ones (due to dilation about the sphere's center) so, even though the mesh is defined in a Cartesian coordinate system, the ∇^2 operator automatically functions as

$$\frac{1}{r^2}\frac{\partial}{\partial r}\left(r^2\frac{\partial}{\partial r}\right)+\frac{1}{r^2\sin\theta}\frac{\partial}{\partial\theta}\left(\sin\theta\frac{\partial}{\partial\theta}\right)+\frac{1}{r^2\sin^2\theta}\frac{\partial^2}{\partial\varphi^2}$$

instead of

$$\frac{\partial^2}{\partial x^2}+\frac{\partial^2}{\partial y^2}+\frac{\partial^2}{\partial z^2}$$

although with an error that's linear instead of quadratic in the grid spacing.

Phase Field

To convert a liquid material to a solid, we must cool it to a temperature below its melting point (known as *undercooling* or *supercooling*). The rate of solidification is often assumed (and experimentally found) to be proportional to the undercooling. Under the right circumstances, the solidification front can become unstable, leading to dendritic patterns as seen in Figure 1b. Researchers[8] have described a phase-field model (Allen-Cahn, "non-conserved Ginsberg-Landau," or "model A" of Hohenberg and Halperin) of such a system, in-

cluding the effects of discrete crystalline orientations (anisotropy).

We start with a regular 2D Cartesian mesh

```
from fipy import *
dx = dy = 0.025
nx = ny = 500
mesh = Grid2D(dx=dx, dy=dy,
              nx=nx, ny=ny)
```

and we'll take fixed time steps

```
dt = 5e-4
```

We consider the simultaneous evolution of a phase-field variable ϕ (taken to be 0 in the liquid phase and 1 in the solid)

```
phase = CellVariable(name=r'$\phi$',
 mesh=mesh, hasOld=True)
```

and a dimensionless undercooling ΔT (ΔT = 0 at the melting point)

```
dT = CellVariable(name=r'$\Delta T$',
 mesh=mesh, hasOld=True)
```

The `hasOld` flag causes FiPy to store the variable value from the previous time step, which is necessary for solving equations with nonlinear coefficients or for coupling between PDEs.

The governing equation for the temperature field is the heat-flux equation, with a source due to the latent heat of solidification

$$\frac{\partial \Delta T}{\partial t} = D_T \nabla^2 \Delta T + \frac{\partial \phi}{\partial t}$$

```
DT = 2.25
heatEq = (TransientTerm()
 == DiffusionTerm(DT)
 + (phase - phase.getOld()) / dt)
```

The governing equation for the phase field is

$$\tau_\phi \frac{\partial \phi}{\partial t} = \nabla \cdot D \nabla \phi + \phi(1-\phi) m(\phi, \Delta T),$$

where

$$m(\phi, \Delta T) = \phi - \frac{1}{2} - \frac{\kappa_1}{\pi} \arctan(\kappa_2 \Delta T)$$

represents a source of anisotropy. The coefficient D is an anisotropic diffusion tensor in two dimensions:

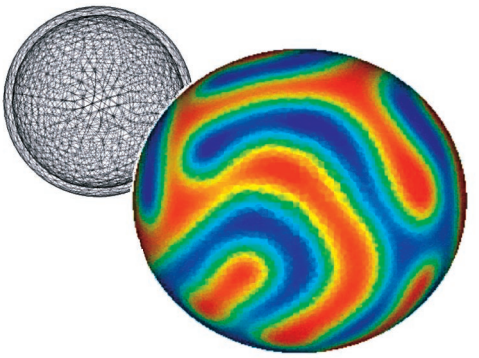

Figure 4. Spinodal decomposition on a sphere's surface. Compare this with, for example, www.youtube.com/watch?v=kDsFP67_ZSE.

$$D = \alpha^2 (1 + c\beta) \begin{bmatrix} 1 + c\beta & -c \dfrac{\partial \beta}{\partial \psi} \\ c \dfrac{\partial \beta}{\partial \psi} & 1 + c\beta \end{bmatrix},$$

where

$$\beta = \frac{1 - \Phi^2}{1 + \Phi^2},$$

$$\Phi = \tan\left(\frac{N}{2}\psi\right), \quad \psi = \theta \arctan \frac{\partial \phi / \partial y}{\partial \phi / \partial x},$$

θ is the orientation, and N is the symmetry:

```
alpha = 0.015
c = 0.02
N = 6.
theta = pi / 8.
psi = (theta
 + arctan2(phase.getFaceGrad()[1],
           phase.getFaceGrad()[0]))
Phi = tan(N * psi/2)
PhiSq = Phi**2
beta = (1. - PhiSq) / (1. + PhiSq)
DbetaDpsi = -N * 2 * Phi / (1 + PhiSq)
Ddia = (1. + c * beta)
Doff = c * DbetaDpsi
D = (alpha**2 * (1. + c * beta)
     * (Ddia * (( 1, 0),
                ( 0, 1))
        + Doff * (( 0,-1),
                  ( 1, 0)))
```

With these expressions defined, we can construct the phase-field equation as

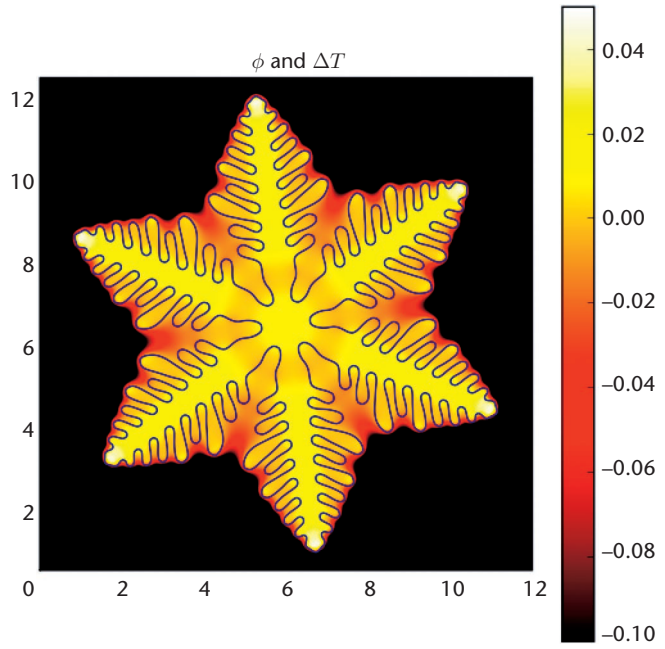

Figure 5. Dendrite formation due to temperature gradients. Faster solidification in steeper temperature gradients leads to surface instability and progressively finer branching.

```
tau = 3e-4
kappa1 = 0.9
kappa2 = 20.
phaseEq = (TransientTerm(tau)
 == DiffusionTerm(D)
 + ImplicitSourceTerm((phase - 0.5
  - kappa1 / pi * arctan(kappa2 * dT))
  * (1 - phase)))
```

We seed a circular solidified region in the center

```
R = dx * 5.
C = (nx * dx/2, ny * dy / 2)
x, y = mesh.getCellCenters()
phase.setValue(1., where=
 ((x-C[0])**2 + (y-C[1])**2) < R**2)
```

and to then quench the entire simulation domain below the melting point

```
dT.setValue(-0.5)
```

In a real solidification process, dendritic branching is induced by small thermal fluctuations along an otherwise smooth surface, but the granularity of the `Mesh` is enough "noise" in this case, so we don't need to explicitly introduce randomness, the way we did in the Cahn-Hilliard problem.

FiPy's viewers are utilitarian, striving to let users see *something*, regardless of their operating system or installed packages. As a result, users won't be able to simultaneously view two fields "out of the box," but, because all of Python is accessible and FiPy is object oriented, it isn't hard to adapt one of the existing viewers to create a specialized display:

```
import pylab
class DendriteViewer(
  Matplotlib2DGridViewer):
  def __init__(self, phase, dT,
    title=None, limits={}, **kwlimits):
    self.phase = phase
    self.contour = None
    Matplotlib2DGridViewer.__init__(
      self, vars=(dT,), title=title,
      cmap=pylab.cm.hot,
      limits=limits, **kwlimits)

  def _plot(self):
    Matplotlib2DGridViewer._plot(self)
    if self.contour is not None:
      cc = self.contour.collections
      for c in cc:
        c.remove()
    mesh = self.phase.getMesh()
    shape = mesh.getShape()
    x, y = mesh.getCellCenters()
    z = self.phase.getValue()
    x, y, z = [a.reshape(
                shape, order="FORTRAN")
               for a in (x, y, z)]
    self.contour = pylab.contour(
                       x, y, z, (0.5,))

viewer = DendriteViewer(
  phase=phase, dT=dT,
  title=r"%s & %s" % (phase.name,
                      dT.name),
  datamin=-0.1, datamax=0.05)
```

and iterate the solution in time, plotting as we go,

```
for i in range(10000):
  phase.updateOld()
  dT.updateOld()
  phaseEq.solve(phase, dt=dt)
  heatEq.solve(dT, dt=dt)
  if i % 10 == 0:
    viewer.plot()
```

as seen in Figure 5. The nonuniform temperature results from the release of latent heat at the solidifying interface. The dendrite arms grow fastest where

the temperature gradient is steepest. We note that this FiPy simulation is written in roughly 50 lines of code (excluding the custom viewer), compared with more than 800 lines of (fairly lucid) Fortran code used for the figures found in the original work.[8]

We developed FiPy to address a troublesome repetition of effort in our own research and in that of our colleagues. The result is a tool that we enjoy using and that we've been able to apply to a diverse collection of research problems. We can get an implementation of a new combination of physics far faster than we ever could when we were coding in C or Fortran. We've been gratified that many others have found our tool useful, often for applications we never dreamed of. We're also pleased that FiPy is already helping in the education of the next generation of materials scientists. Our current development efforts focus on improving performance, through parallelism and more sophisticated matrix preconditioners and solvers, by exploiting the Trilinos package from Sandia National Laboratories. We look forward to tackling even larger and more complicated problems, while keeping the ease of use that we've already established. CiSE

References

1. M. Tobis, "PyNSol: Objects as Scaffolding," *Computing in Science & Eng.*, vol. 7, no. 4, 2005, pp. 84–91.
2. K.-A. Mardal et al., "Using Python to Solve Partial Differential Equations," *Computing in Science & Eng.*, vol. 9, no. 3, 2007, pp. 48–51.
3. D. Josell, D. Wheeler, and T.P. Moffat, "Gold Superfill in Submicrometer Trenches: Experiment and Prediction," *J. Electrochemical Soc.*, vol. 153, no. 1, 2006, pp. C11–C18.
4. W.J. Boettinger et al., "Computation of the Kirkendall Velocity and Displacement Fields in a One-Dimensional Binary Diffusion Couple with a Moving Interface," *Proc. Royal Soc. A: Mathematical*, vol. 463, 2007, pp. 3347–3373.
5. J. Mazur, "Numerical Simulation of Temperature Field in Soil Generated by Solar Radiation," *J. Physique IV France*, vol. 137, Nov. 2006, pp. 317–320.
6. J.W. Cahn and J.E. Hilliard, "Free Energy of a Nonuniform System. I. Interfacial Free Energy," *J. Chemical Physics*, vol. 28, no. 2, 1958, pp. 258–267.
7. J.W. Cahn, "Free Energy of a Nonuniform System. II. Thermodynamic Basis," *J. Chemical Physics*, vol. 30, no. 5, 1959, pp. 1121–1124.
8. J.A. Warren et al., "Extending Phase Field Models of Solidification to Polycrystalline Materials," *Acta Materialia*, vol. 51, no. 20, 2003, pp. 6035–6058.

Jonathan E. Guyer *is a member of the Thermodynamics and Kinetics Group in the Metallurgy Division at the US National Institute of Standards and Technology. His interests include models of phase transformations, particularly in electrochemical and semiconductor systems. Guyer has a PhD in materials science and engineering from Northwestern University. Contact him at guyer@nist.gov.*

Daniel Wheeler *is a guest researcher in the Thermodynamics and Kinetics Group in the Metallurgy Division at the US National Institute of Standards and Technology. His interests include numerical analysis for interpreting experimental results. Wheeler has a PhD in computational mechanics from Greenwich University. Contact him at daniel.wheeler@nist.gov.*

James A. Warren *is leader of the Thermodynamics and Kinetics Group in the Metallurgy Division at the US National Institute of Standards and Technology. His interests include developing both models of materials phenomena and the tools to enable the solution of these models. Warren has a PhD in physics from the University of California, Santa Barbara. Contact him at james.warren@nist.gov.*