

# Using Python to Solve Partial Differential Equations

*This article describes two Python modules for solving partial differential equations (PDEs): PyCC is designed as a Matlab-like environment for writing algorithms for solving PDEs, and SyFi creates matrices based on symbolic mathematics, code generation, and the finite element method.*

**O**ur work at the Simula Research Laboratory mostly focuses on computational applications in life sciences. Usually, this involves fairly typical partial differential equations such as the incompressible Navier-Stokes equations, elasticity equations, and parabolic and elliptic PDEs, but these PDEs are typically coupled either with each other or with ordinary differential equations (ODEs). Hence, even though the PDEs themselves are reasonably well understood, the couplings between them make the problems we study quite challenging.

Our design goals are therefore threefold. First, we want to easily define systems of PDEs. Second, we want it to be easy to play with different solution algorithms for systems of coupled PDEs. Finally, we want to reuse existing software to avoid reinventing the wheel.

We use many good and mature libraries from the Web, including Dolfin ([www.fenics.org/dolfin/](http://www.fenics.org/dolfin/)), GiNaC ([www.ginac.de/](http://www.ginac.de/)), MayaVi (<http://mayavi.sourceforge.net/>), NumPy (<http://numpy.scipy.org/>),

PETSc ([www.mcs.anl.gov/petsc/](http://www.mcs.anl.gov/petsc/)), SciPy ([www.scipy.org/](http://www.scipy.org/)), Trilinos (<http://software.sandia.gov/trilinos/>), and VTK ([www.vtk.org/](http://www.vtk.org/)). In fact, we're mixing these libraries with our own packages:

- Famms (verification based on the method of manufactured solutions),
- Instant ([www.fenics.org/instant](http://www.fenics.org/instant); inlining of C++ in Python),
- PyCC ([http://folk.uio.no/skavhaug/heart\\_simulations.html](http://folk.uio.no/skavhaug/heart_simulations.html); the underlying framework for gluing components together),
- PySE (<http://pyfdm.sf.net>; a finite difference toolbox),
- Swiginac (<http://swiginac.berlios.de/>; a Python interface to the symbolic mathematics engine GiNaC), and
- SyFi ([www.fenics.org/syfi/](http://www.fenics.org/syfi/); a finite element toolbox).

Some of these packages are Python modules, whereas the others—thanks to Python's popularity in scientific computing—are equipped with Python interfaces. By using Python, we don't have to mix these packages at the C level, which is a huge advantage.

## Solving Systems of PDEs

Currently, our most important application is in cardiac electrophysiology.<sup>1</sup> The central model here is the *bidomain model*,<sup>2</sup> which is a system of two PDEs

1521-9615/07/\$25.00 © 2007 IEEE  
Copublished by the IEEE CS and the AIP

KENT-ANDRE MARDAL, OLA SKAVHAUG, GLENN T. LINES,  
GUNNAR A. STAFF, AND ÅSMUND ØDEGÅRD  
*Simula Research Laboratory*

with the following form:

$$\frac{\partial v}{\partial t} = \nabla \cdot (M_i \nabla v) + \nabla \cdot (M_i \nabla u) - I_{ion}(v, s), \quad (1)$$

$$0 = \nabla \cdot (M_i \nabla v) + \nabla \cdot ((M_i + M_e) \nabla u). \quad (2)$$

(The domain here is the same for both PDEs—that is, the heart—but it has two potentials, intra- and extra-cellular, which live inside and outside heart cells.) The primary unknowns here are the transmembrane potential  $v$  and the extra cellular potential  $u$ . The function  $I_{ion}(v, s)$  describes the flow of ions across the cell membrane and can be quite complicated;  $M_i$  represents intracellular conductivity, and  $M_e$  is extracellular. The second argument  $I_{ion}(v, s)$  is generally a vector of variables, governed by a set of ODEs:

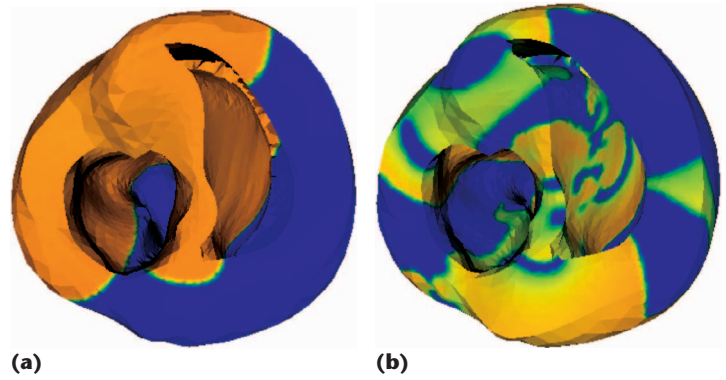
$$\frac{ds}{dt} = F(s, t). \quad (3)$$

Note that  $s = s(x)$ , so the ODE system is defined in each point (you can find examples of cell models at [www.cellml.org](http://www.cellml.org)). Hence, we must solve the system for each computational node.

The bidomain formulation gives an accurate description of the myocardial tissue's electrical conduction. Coupled with realistic ODE models of the ionic current, we can study a large range of phenomena, including conduction abnormalities due to ischmeia or channel myopathies (genetic defects), fibrillation and defibrillation, and drug intervention. Figure 1, for example, shows a geometric model of the human heart's ventricles. The color represents the transmembrane potential's magnitude; Figure 1a shows normal activation, and Figure 1b shows chaotic behavior (which corresponds to a fibrillatory heart with critically reduced pumping ability).

We solve the bidomain model in Equations 1 through 3 by using an operator-splitting approach, in which we first solve the ODE systems in each computational node at each time step before we solve the PDE system. Here's a simple Python script we use for solving this problem:

```
from dolfin import Mesh
from pycc.MatSparse import *
import numpy
from pycc import MatFac
from pycc import ConjGrad
from pycc.BlockMatrix import *
from pycc.Functions import *
from pycc.ODESystem import *
from pycc.CondGen import *
from pycc.IonicODEs import *
```



**Figure 1. Human heart ventricles. This model compares (a) normal activity and (b) chaotic behavior.**

```
mesh = Mesh("Heart.xml.gz")
matfac = MatFac.MatrixFactory(mesh)

M = matfac.computeMassMatrix()

pc = PyCond("Heart.axis")
pc.setconductances(3.0e-3, 3e-4)
ct = ConductivityTensorFunction(
    pc.conductivity)
Ai = matfac.computeStiffnessMatrix(ct)

pc.setconductances(5.0e-3, 1.6e-3)
ct = ConductivityTensorFunction(
    pc.conductivity)
Aie = matfac.computeStiffnessMatrix(ct)

# Construct compound matrices
dt = 0.1
A = M + dt*Ai
B = dt*Ai
Bt = dt*Ai
C = dt*Aie

# Create the Block system
AA = BlockMatrix((A,B),(Bt,C))
prec = DiagBlockMatrix((MLPrec(A),
    MLPrec(C)))

v = numpy.zeros(A.n, dtype='d') - 45.0
u = numpy.zeros(A.n, dtype='d')
x = BlockVector(v,u)

# Create one ODE systems for each vertex
odesys = Courtemanche_ODESystem()
ode_solver = RKF32(odesys)
ionic = IonicODEs(A.n, ode_solver,
    odesys)
```

```

ionic.setState(odesys.getDefault
               InitialCondition())

# Solve
z = numpy.zeros((A.n,), dtype='d')
for i in xrange(0, 10):
    t = i*dt
    ionic.forward(x[0], t, dt)
    ConjGrad.precondconjgrad(prec, AA,
                              x, BlockVector(M*x[0], z))

```

Although the code seems clean and simple, it's due to a powerful combination of C/C++/Fortran and Python. The script runs on desktop computers with meshes that have millions of nodes and can solve complete problems within minutes or hours. All computer-intensive calculations such as computing matrices, solving linear systems (via algebraic multigrid and the conjugate gradient method), and solving ODE systems are done efficiently in C or C++.

## Creating Matrices for Systems of PDEs

We created the tool SyFi to define finite elements and variational forms, as well as generate C++ code for finite element computations. It uses the symbolic mathematics engine GiNaC and its Python interface Swiginac for all its basic mathematical operations. SyFi enables polynomial differentiation and integration on polygonal domains. Furthermore, it uses the computed expressions, such as entries in an element matrix, to generate C++ code.

The following example demonstrates how to compute an element matrix for the Jacobian of an incompressible power-law fluid's (nonlinear) stationary Navier-Stokes equations. Let

$$\mathbf{F}_i = \int_T (\mathbf{u} \cdot \nabla \mathbf{u}) \cdot \mathbf{N}_i + \mu(\mathbf{u}) \nabla \mathbf{u} : \nabla \mathbf{N}_i dx,$$

where

$$\mathbf{u} = \sum_k u_k \mathbf{N}_k \text{ and } \mu(\mathbf{u}) = \|\nabla \mathbf{u}\|^{2n}.$$

Then,

$$\begin{aligned} J_{ij} &= \frac{\partial \mathbf{F}_i}{\partial u_j} \\ &= \frac{\partial}{\partial u_j} \int_T (\mathbf{u} \cdot \nabla \mathbf{u}) \cdot \mathbf{N}_i + \mu(\mathbf{u}) \nabla \mathbf{u} : \nabla \mathbf{N}_i dx, \quad (4) \end{aligned}$$

Here's the corresponding Python code for computing Equation 4 and generating the C code:

```

from swiginac import *
from SyFi import *

```

```

def sum(u_char, fe):
    ujs = symbolic_matrix(1, fe.nbf(),
                           u_char)
    u = 0
    for j in range(0, fe.nbf()):
        u += ujs.op(j)*fe.N(j)
    u = u.evalm()
    return u, ujs

nsd = cvar.nsd = 3
polygon = ReferenceTetrahedron()
fe = VectorCrouzeixRaviart(polygon, 1)
fe.set_size(nsd) # size of vector
fe.compute_basis_functions()

# create sum u_i N_i
u, ujs = sum("u", fe)
n = symbol("n")
mu = pow(inner(grad(u), grad(u)), n)

for i in range(0, fe.nbf()):
    # nonlinear power-law diffusion term
    fi_diffusion = mu*inner(grad(u),
                             grad(fe.N(i)))

    # nonlinear convection term
    uxgradu = (u.transpose()
               *grad(u)).evalm()
    fi_convection = inner(uxgradu,
                          fe.N(i), True)

    fi = fi_diffusion + fi_convection

    Fi = polygon.integrate(fi)

    for j in range(0, fe.nbf()):
        # differentiate to get the Jacobian
        uj = ujs.op(j)
        Jij = diff(Fi, uj)
        print "J[%d,%d]=%s\n"%(i,j,
                                Jij.printc())

```

Note that both the differentiation and integration is performed symbolically exactly as we would have done by hand. This naturally leads to quite efficient code compared to the traditional way of implementing such integrals—namely, as quadrature loops that involve the evaluation of basis functions, their derivatives, and so on. The `printc` function generates C++ code for the expressions; so far, we've used this system to generate roughly 60,000 lines of C++ code for computing various matrices based on various finite elements and variational forms.

To ease the integration of the generated C++ code in Python, we developed an inlining tool called

Instant, which lets us generate code, generate the corresponding wrapper code, compile and link it to an extension module, and then import the module on the fly. The following code demonstrates Instant with a simple example, in which we compute

$$y(x) = \frac{\partial}{\partial x} \sin(\cos(x))$$

symbolically, generate the corresponding C++ code, and inline the expression in Python with  $x$  as a NumPy array:

```
import swiginac as S
from Instant import inline_with_numpy
import numpy as N

x = S.symbol("x")
xi = S.symbol("x[i]")
f = S.sin(S.cos(x))

dfdx = S.diff(f,x)
print dfdx

string = ""
void func (int n, double* x, int m,
double* y) {
    if ( n != m ) {
        printf("Both arrays should be of
            the same size!");
        return;
    }

    for (int i=0; i<n; i++) {
        y[i] = %s;
    }
} "" % dfdx.subs( x == xi ).printc()

print string


func = inline_with_numpy(string, arrays
    = [['n', 'x'], ['m', 'y']])

x = N.arange(100.0 )
y = N.zeros(100, dtype='d')

func(x,y)

print x
print y
```

**W**e've shown that it's possible to solve real-life problems in a user-friendly environment by combining Python's high-level syntax with

the efficiency of compiled languages. However, this approach opens up many new possibilities for combining symbolic mathematics and code generation, which is a largely overlooked alternative to traditional approaches in finite element simulations. We're currently implementing fairly advanced finite element methods such as the mixed elasticity method,<sup>3</sup> and we also want to simulate human tissue and blood with the most realistic models available today. 

## Acknowledgments

*Mardal is supported by the Research Council of Norway under the grant ES254277.*

## References

1. J. Sundnes et al., *Computing the Electrical Activity in the Human Heart*, Monographs in Computations Science and Engineering, Springer-Verlag, 2006.
2. C.S. Henriquez, "Simulating the Electrical Behavior of Cardiac Tissue Using the Bidomain Model," *Crit. Rev. Biomedical Eng.*, vol. 21, no. 1, 1993, pp. 1-77.
3. D.N. Arnold, R.S. Falk, and R. Winther, "Finite Element Exterior Calculus, Homological Techniques, and Applications," *Acta Numerica*, 2006, pp. 1-155.

**Kent-Andre Mardal** is a postdoc at the Simula Research Laboratory. His research interests include high-level numerical programming and solution of PDEs. Mardal has a PhD in scientific computing from the University of Oslo. Contact him at [ken-and@simula.no](mailto:ken-and@simula.no).

**Ola Skavhaug** is a research scientist at the Simula Research Laboratory. His research interests include high-level numerical programming, PDEs, and code verification. Skavhaug has a PhD in scientific computing from the University of Oslo. Contact him at [skavhaug@simula.no](mailto:skavhaug@simula.no).

**Glenn T. Lines** is a research scientist at the Simula Research Laboratory. His research interests include PDEs and computer simulation of cardiac electrophysiology. Lines has a PhD in scientific computing from the University of Oslo. Contact him at [glennli@simula.no](mailto:glennli@simula.no).

**Gunnar A. Staff** is a consultant at Scandpower Petroleum Technology. His research interests include software for scientific computing and initial value problems. Staff has a PhD in scientific computing from the University of Oslo. Contact him at [gst@scandpowerpt.com](mailto:gst@scandpowerpt.com).

**Åsmund Ødegård** is an IT manager and part-time research scientist at the Simula Research Laboratory. His research interests include high-level languages in scientific computing, object-oriented numerics, PDEs, and high-level parallelism. Ødegård has a PhD in computational science from the University of Oslo. Contact him at [aasmund@simula.no](mailto:aasmund@simula.no).