Editor: David Winch, winch@taosnet.net

PYTHON FOR EDUCATION

Computational Methods for Nonlinear Systems

By Christopher R. Myers and James P. Sethna

The authors' interdisciplinary computational methods course uses Python and associated numerical and visualization libraries to enable students to implement simulations for several different course modules, which highlight the breadth and flexibility of Python-powered computational environments.

The field of computational science and engineering (CSE) integrates mastery of specific domain sciences with expertise in data structures, algorithms, numerical analysis, programming methodologies, simulation, visualization, data analysis, and performance optimization. The CSE community has embraced Python as a platform for attacking a wide variety of research problems, in part because of Python's support for easily gluing together tools from different domains to solve complex problems. Many of the same advantages that Python brings to CSE research also make it useful for teaching: Python and its many batteries can help students learn a wide swath of techniques necessary to perform effective CSE research.

"Computational Methods for Nonlinear Systems" is a graduate-level computational science laboratory course that we jointly teach at Cornell. We began developing the course in summer 2004 to support the curricular needs of the Cornell IGERT program in nonlinear systems, a broad and interdisciplinary graduate fellowship program aimed at introducing theoretical and computational techniques developed in the study of nonlinear and complex systems to a range of fields.

The course's format is somewhat unusual. As a computational laboratory course, it provides relatively little in the way of lectures: we prefer to have students learn by doing rather than listening. The course is autonomous, modular, and self-paced: students choose computational modules to work on from a large (and hopefully growing) suite of those available, and then proceed to implement relevant simulations and analyses as laid out in the exercises. We provide "Hints" files to help the students along: these consist of documented skeletal code that the students are meant to flesh out. We've written several different visualization tools to provide visual feedback. We find these help engage the students in new problems and are useful in code debugging.

Python is a useful teaching language for several reasons. Its clean syntax lets students learn the language quickly, and lets us provide concise programming hints in our documented code fragments. Python's dynamic typing and high-level, built-in datatypes enable students to get programs working quickly, without struggling with type declarations and compile-link-run loops. Because Python is interpreted, students can learn the language by executing and analyzing individual commands, and we can help them debug their programs by working with them in the interpreter.

Another key advantage that Python brings to scientific computing is the availability of many packages supporting numerical algorithms and visualization. While some of our exercises require developing algorithms from scratch, others rely on established numerical routines implemented in third-party libraries. Although it's important to understand the fundamentals of algorithms, error analysis, and algorithmic complexity, it's also useful to know when and how to use existing solutions. We make heavy use of the NumPy (www.scipy.org/numpy) and SciPy (www.scipy.org) packages for efficiently manipulating arrays and for accessing routines to generate random numbers, integrate ordinary differential equations, find roots, compute eigenvalues, and so on. We use matplotlib (http://matplotlib.sourceforge.net) for *x*-*y* plotting and histograms. We've written several visualization modules that we provide to students, based on the Python Imaging Library (PIL; www.pythonware.com/products/pil), using PIL's ImageDraw module to place graphics primitives within an image, and the ImageTk module to paste an image into a Tk window for real-time animation. We recommend the use of the IPython interpreter, which facilitates exploration by students (www.ipython.scipy.org). We've also used VPython (www.vpython.org) to generate 3D animations to accompany some of our modules.

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

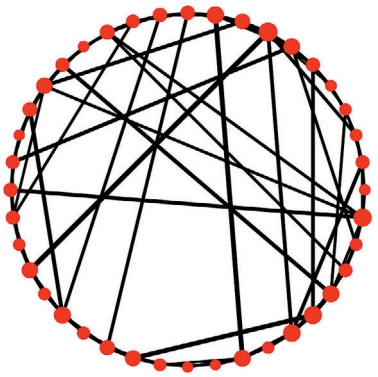

Figure 1. Node and edge betweenness in a model of small-world networks. Undirected edges (black lines) connect nodes (red dots). Betweenness measures how central each node and edge is to the shortest network paths connecting any two nodes. In this plot, node diameter and edge thickness are proportional to node and edge betweenness, respectively.

Course Modules

Our course modules are too numerous to describe in detail in this article, so we refer interested readers to our course Web site (www.physics.cornell.edu/sethna/teaching/ComputationalMethods) for more information, as well as access to problems, hints, and answers. (Many of the exercises have also been incorporated into a new textbook.[1]) Here, we highlight a few of the modules to illustrate both the breadth of science that you can teach with Python and the variety of tools and techniques that Python can bring to bear on such problems.

Small-World Networks

The study of complex networks has flourished over the past several years as researchers have discovered commonalities among networked structures that arise in diverse fields such as biology, ecology, sociology, and computer science.[2] An interesting property found in many complex networks is exemplified in the popular notion of "six degrees of separation," which suggests that any two people on Earth are connected through roughly five intermediate acquaintances. Duncan Watts (now at Columbia) and Steve Strogatz at Cornell[3] developed a simple model of random networks that demonstrates this "small world" property. Our course module enables students to construct small-world networks and examine how the average path length connecting two nodes decreases rapidly as random, long-range bonds are introduced into a network consisting initially of only short-ranged bonds (see Figure 1).

Computationally, this module introduces students to data structures that represent undirected graphs, object-oriented encapsulation of those data structures, and graph-traversal algorithms. Python makes the development of an undirected graph data structure exceedingly simple, a point made long ago by Python creator Guido van Rossum in one of his early essays on the language.[4] In an undirected graph, nodes are connected to other nodes by edges. A simple way to implement this is to combine the two cornerstones of container-based programming in Python: lists and dictionaries. In our `UndirectedGraph` class, a dictionary of network neighbor connections (a neighbor dictionary) maps a node identifier to a list of other nodes to which the reference node is connected. Because the graph edges are undirected, we duplicate the connection information for each node: if an edge is added connecting nodes 1 and 2, the neighbor dictionary must be updated so that node 2 is added to node 1's list of neighbors, and vice versa.

We can, of course, hide the details of adding edges inside an `AddEdge` method defined on an `UndirectedGraph` class:

```
class UndirectedGraph:
  # ...
  def AddEdge(self, n1, n2):
    """Add an edge connecting
    nodes n1 and n2"""
    self.AddNode(n1)
    self.AddNode(n2)
    nd = self.neighbor_dict
    if n2 not in nd[n1]:
      nd[n1].append(n2)
    if n1 not in nd[n2]:
      nd[n2].append(n1)
```

In the small-world networks exercise, we choose to label nodes simply by integers, but Python's dynamic typing doesn't require this. If we were playing the "six degrees of Kevin Bacon" game of searching for shortest paths in actor collaboration networks, we could use our code snippet to build a graph connecting the names of actors (encoded as strings). This dynamic typing allows for significant code reuse (as described in the next section). Although our `UndirectedGraph` class is exceedingly simple and built to support only the analyses relevant to our course module, the same basic principles are at work in a much more comprehensive, Python-based, graph construction and analysis package—NetworkX—developed at Los Alamos National Labs (http://networkx.lanl.gov).

Percolation

Percolation is the study of how objects become connected (or disconnected) as they're randomly wired together (or cut apart). It's an important and classic problem in the study of phase transitions that has practical relevance as well: the oil and gas industry, for example, has shown considerable interest over the years in percolation phenomena because fluid is extracted through a network of pores in rock.

Although percolation is traditionally studied on regular lattices, it's a problem more generally applicable to arbitrary networks, and in fact, we're able to reuse some of the code devel-

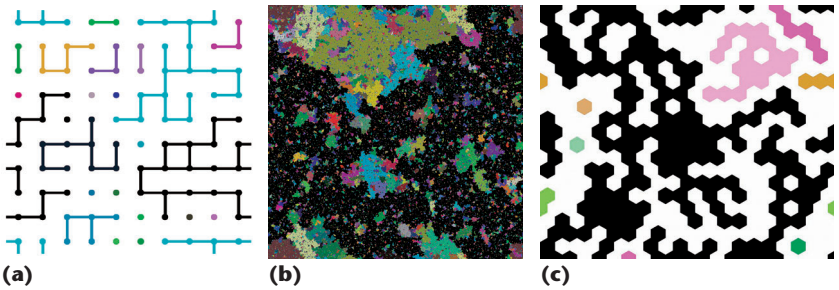

Figure 2. Two instances of bond-percolation on a 2D square lattice, and an instance of site-percolation on a triangular lattice. In bond-percolation, neighboring lattice points are connected with probability *p*, and connected clusters in the resulting network are identified via breadth-first search. Separate clusters are colored distinctly, for (a) a 10 × 10 grid and (b) a 1,024 × 1,024 grid. In (c) site-percolation, lattice sites are filled with probability *p*, and clusters connect the filled neighboring sites.

oped in the small-world networks module to support percolation studies. As noted earlier, Python's dynamic typing makes our definition of a node in a graph very flexible; in a percolation problem on a lattice, we can reuse our `UndirectedGraph` class described earlier by making node identifiers be lattice index tuples (i, j). We can thus easily make an instance of bond percolation on a 2D square lattice of size L (with periodic boundary conditions) and bond fraction p:

```
def MakeSquareBondPerc(L,p):
  """Constructs and returns
  a bond percolation
  instance on an LxL square
  lattice with periodic
  boundaries, where bonds are
  filled with probability p"""
  g = UndirectedGraph()
  for i in range(L):
    for j in range(L):
      g.AddNode((i,j))
      if random.random() < p:
        g.AddEdge((i,j), \
                  ((i+1)%L,j))
      if random.random() < p:
        g.AddEdge((i,j), \
                  (i,(j+1)%L))
  return g
```

Figure 2 shows instances of percolation networks generated by this procedure. Students use breadth-first search to identify all connected clusters in such a network, and our PIL-based visualization tool colors each separate cluster distinctly, taking as input a list of all nodes in each cluster.

We also introduce the concept of universality of phase transitions in the course: despite their microscopic differences, site-percolation on a 2D triangular lattice and bond-percolation on a 2D square lattice are indistinguishable from each other on long length scales, and exhibit the same critical behavior (scaling exponents). Scaling collapses are a useful construct for revealing the universality of phase transitions, and typically involve transforming the x and y axes in specified ways to get disparate data sets to "collapse" onto one universal scaling form. With Python, we can support such scaling collapses very flexibly by using the built-in `eval()` function that evaluates expressions encoded as strings. Rather than hard-coding particular functional forms for scaling collapses, we can simply encode and evaluate arbitrary mathematical expressions.

Pattern Formation in Cardiac Dynamics

Pattern formation is ubiquitous in spatially extended nonequilibrium systems. Many patterns involve regular, periodic phenomena in space and time, but equally important are localized coherent structures that break or otherwise interrupt these periodicities. Patterns lie at the root of much activity in living tissues: the regular beating of the human heart is perhaps our most familiar reminder of the spatiotemporal rhymicity of biological patterns. Cardiac tissue is an excitable medium: rhythmic voltage pulses, initiated by the heart's pacemaker cells (in the sinoatrial node), spread as a wave through the rest of the heart, inducing the heart muscle to contract and thereby pumping blood in a coherent fashion. In some situations, however, this regular beating can become interrupted by the presence of spiral waves in the heart's electrical activity (see Figure 3). These spiral waves generate voltage pulses on their own, disrupting the normal heart's coordinated rhythm, leading to cardiac arrythmia. Our course module, which we developed in conjunction with Niels Otani of Cornell's biomedical sciences department, introduces a simple model of cardiac dynamics—the two-dimensional FitzHugh-Nagumo equations.[5,6] The FitzHugh-Nagumo model describes the coupled time evolution of two fields, the transmembrane potential V and the recovery variable W (given parameters ε, γ, and β):

$$\frac{\partial V}{\partial t} = \nabla^2 V + \frac{1}{\varepsilon}(V - V^3/3 - W)$$

$$\frac{\partial W}{\partial t} = \varepsilon(V - \gamma W + \beta).$$

Fixed-point solutions to the FitzHugh-Nagumo equations come by root-finding, which we accomplish using the `brentq` function in SciPy:

```
def FindFixedPoint(c, b):
  """Given parameters
  c (gamma) and b (beta),
  returns (v*,w*) for which
  dv/dt=0 and dw/dt=0 for
```

Figure 3. Snapshot in the time evolution of the FitzHugh-Nagumo model of cardiac dynamics. The transmembrane voltage *V* is depicted via a grayscale map (higher voltages are in lighter gray). Spiral waves in the voltage field can lead to cardiac arrythmias by disrupting the normal periodic rhythm generated by the sinoatrial node.

```
FitzHugh-Nagumo model"""
f = lambda v, c, b: \
    (v-(v**3)/3.)-  \
    ((1./c)*(v+b))
vstar = brentq(f,-2.,2., \
             args=(c, b))
wstar = ((1./c)*(vstar+b))
return vstar, wstar
```

We also introduce students to finite difference techniques for computing spatial derivatives in the solution of partial differential equations (PDEs). NumPy arrays represent the V and W fields of the FitzHugh-Nagumo model, and we can use stencil notation and array syntax to compactly compute the Laplacian of the voltage field, $\nabla^2 V(x, y)$. We ask students to implement two different approximations to the Laplacian operator (a five- and nine-point stencil), and compare their effects on the detailed form of propagating electrical waves. The computation of the five-point stencil is shown here:

```
def del2_5(a, dx):
  """del2_5(a, dx) returns
  the finite-difference
  approximation of the
  laplacian of the array a,
  with lattice spacing dx,
  using the five-point stencil:
        0   1   0
        1  -4   1
        0   1   0
  """
  del2 = scipy.zeros(a.shape,
                     float)
  del2[1:-1, 1:-1] = \
    (a[1:-1,2:]+a[1:-1,:-2]+ \
     a[2:,1:-1]+a[:-2,1:-1]- \
     4.*a[1:-1,1:-1])/(dx*dx)
  return del2
```

At this point, we provide an animation tool that we wrote, based on PIL and Tkinter, which lets students update the display of the voltage field V at every time step and use the mouse to introduce local "shocks" to the system. These shocks are both useful in initiating spiral waves and in resetting the system's global electrical state as a defibrillator might do. Optional extensions to the module, which our collaborator Otani developed, enable simulations of spontaneous pacemakers, dead regions of tissue, and more complex heart-chamber geometries, by letting the model's various parameters become spatially varying fields themselves (again implemented via NumPy arrays).

Gene Regulation and the Repressilator

Gene regulation describes a set of processes by which the expression of genes within a living cell—their transcription to messenger RNA and ultimately their translation to protein—is controlled. While modern genome sequencing has provided great insights into many organisms' constituent parts (genes, RNAs, and proteins), much less is known about how those parts are turned on and off and mixed and matched in different contexts: how is it that a brain cell and a hair cell, for example, can derive from the same genomic blueprint but have such different properties?

The Repressilator is a relatively simple synthetic gene regulatory network developed by Michael Elowitz (now at Caltech) and Stan Leibler at Rockefeller University.[7] Its name derives from its use of three repressor proteins arranged to form a biological oscillator: these three repressors act in a manner akin to the "rock-paper-scissors" game, in which TetR inhibits λcI, which in turn inhibits LacI, which in turn inhibits TetR. Figure 4 shows a snapshot of the Repressilator's time evolution.

Important scientific and computational features emphasized in this module are the differences between stochastic and deterministic representations of chemical reaction networks. (We first introduce these concepts in a warm-up exercise, called Stochastic Cells, in which students simulate a much simpler biochemical network: one representing the binding and unbinding of two monomer molecules M form a single dimer D: $M + M \leftrightarrow D$.) We introduce students to Petri nets as a graphical notation for encoding such networks, and then have them, from the underlying Petri net representation, both synthesize differential equations describing the deterministic time evolution of the system, and implement the Gillespie algorithm (a form of continuous time Monte Carlo) for stochastic simulation.[8] Gillespie's "direct method" involves choosing a particular reaction and reaction time based on instantaneous reaction rates. For the Repressilator, this can be done quite compactly using array operations within NumPy/SciPy:

```
class StochasticRepressilator:
  # ...
  def Step(self, dtmax):
    """Execute one step of
    the Gillespie direct
    method by: (1) computing
    instantaneous reaction
```

```
rates, (2) getting an
exponentially distributed
random time from rates,
(3) choosing a random
reaction with probability
proportional to reaction
rate, (4) executing the
chosen reaction based on
its stoichiometry, and
(5) returning the time
at which the reaction
takes place"""
# (1)
self.GetReactionRates()
# (2)
tot_rate = sum(self.rates)
ran_time = -scipy.log( \
  1.-random.random())  \
  /tot_rate
if ran_time > dtmax:
  return dtmax
# (3)
ran_rate = tot_rate * \
  random.random()
index = len(self.rates) \
  - sum(scipy.cumsum(\
    self.rates)> ran_rate)
reac = \
  self.reactions[index]
# (4)
for chem, dchem in \
  reac.stoichio.items():
    chem.amount += dchem
# (5)
return ran_time
```

Our course introduces students to several other problems that we can only mention in passing here. This includes modules to study chaos and bifurcations in iterated maps; biolocomotion in a simple model of a bipedal walker; properties of random walks and extremal statistics; connections between NP-complete constraint satisfaction problems and the statistical mechanics of phase transitions; universality of eigenvalue distributions in random matrix theory; the emergence of collective thermodynamic properties from molecular dynamics; and phase transitions and Monte Carlo algorithms in the Ising model of magnetic systems.

We continue to look for new problems to add to this collection, and for collaborators interested in contributing their scientific and computational expertise to this endeavor. (Please contact us if you have ideas for interesting modules.) Our goal is to provide a hands-on introduction to scientific computing, and we hope that this course can help serve several educational objectives in the part of a larger curriculum in CSE.

Acknowledgments

We thank our colleagues who have helped us develop computational modules and have given us useful feedback: Steve Strogatz, Andy Ruina, Niels Otani, Bart Selman, Carla Gomes, and John Guckenheimer. We also thank all the students who have completed our course and have helped us work the bugs out of exercises and solutions. Funding from NSF awards DGE-0333366 and DMR-0218475, and from the Cornell Theory Center, helped support the development of course modules.

References

1. J.P. Sethna, *Statistical Mechanics: Entropy, Order Parameters, and Complexity*, Oxford Univ. Press, 2006.
2. A.-L. Barabasi, *Linked: How Everything Is Connected to Everything Else and What It Means*, Perseus Publishing, 2002.
3. D. Watts and S. Strogatz, "Collective Dynamics of 'Small-World' Networks," *Nature*, vol. 393, no. 6684, 1998, pp. 440–442.
4. G. van Rossum, "Python Patterns: Implementing Graphs," 1998; www.python.org/doc/essays/graphs/.
5. R. FitzHugh, "Impulses and Physiological States in Theoretical Models of Nerve Membrane," *Biophysical J.*, vol. 1, no. 6, 1961, pp. 445–466.

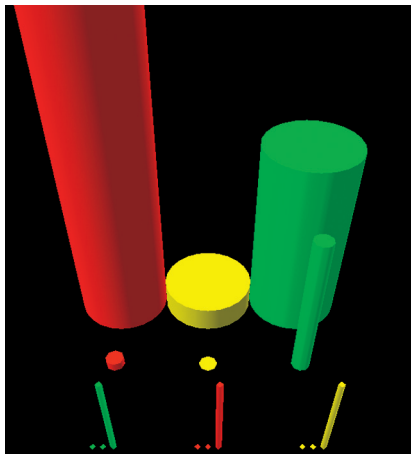

Figure 4. Snapshot in the stochastic time evolution of the Repressilator. Protein concentrations (back row), mRNA concentrations (middle) and promoter states (front) are shown. At this instant, TetR (red) concentration is high, leading to suppression of λcI (yellow). Because λcI is low, however, LacI (green) concentration can grow, leading to TetR's eventual suppression.

6. J. Nagumo, S. Arimoto, and S. Yoshizawa, "An Active Pulse Transmission Line Simulating Nerve Axon," *Proc. Inst.of Radio Engineers*, vol. 50, no. 10, 1962, pp. 2061–2070.
7. M. Elowitz and S. Leibler, "A Synthetic Oscillatory Network of Transcriptional Regulators," *Nature*, vol. 403, no. 6767, 2000, pp. 335–338.
8. D. Gillespie, "Exact Stochastic Simulation of Coupled Chemical Reactions," *J. Physical Chemistry*, vol. 81, no. 25, 1977, pp. 2340–2361.

Christopher R. Myers is a senior research associate and associate director in the Cornell Theory Center at Cornell University. For more than a decade, he has advocated for and explored the capabilities of Python-powered computational environments in physics, materials science, engineering, and biology. Myers has a PhD in physics from Cornell. Contact him at myers@tc.cornell.edu; www.tc.cornell.edu/~myers.

James P. Sethna is a professor of physics at Cornell University. He has a PhD in physics from Princeton University. Sethna is the author of *Statistical Mechanics: Entropy, Order Parameters, and Complexity*; www.physics.cornell.edu/sethna/StatMech/. Contact him via www.lassp.cornell.edu/sethna or sethna@lassp.cornell.edu.