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

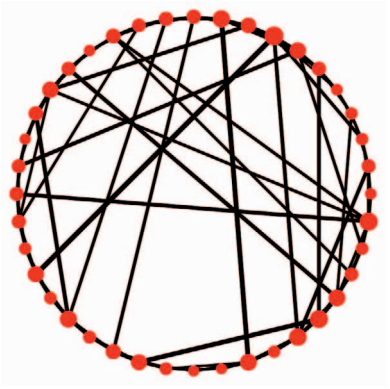


Figure 1. Node and edge betweenness in a model of small-world networks. Undirected edges (black lines) connect nodes (red dots). Betweenness measures how central each node and edge is to the shortest network paths connecting any two nodes. In this plot, node diameter and edge thickness are proportional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

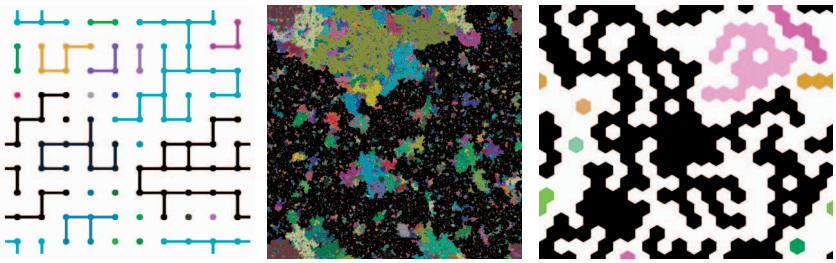


Figure 2. Two instances of bond-percolation on a 2D square lattice, and an instance of site-percolation on a triangular lattice. In bond-percolation, neighboring lattice points are connected with probability *p*, and connected clusters in the resulting network are identified via breadth-first search. 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/.
6. R. FitzHugh, "Impulses and Physiological States in Theoretical Models of Nerve Membrane," *Biophysical J.*, vol. 1, no. 6, 1961, pp. 445–466.
7. 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.
8. M. Elowitz and S. Leibler, "A Synthetic Oscillatory Network of Transcriptional Regulators," *Nature*, vol. 403, no. 6767, 2000, pp. 335–338.
9. D. Gillespie, "Exact Stochastic Simulation of Coupled Chemical Reactions," *J. Physical Chemistry*, vol. 81, no. 25, 1977, pp. 2340–2361.

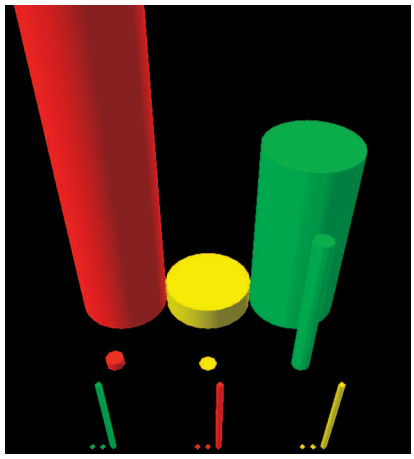


Figure 4. Snapshot in the stochastic time evolution of the Repressilator. Protein concentrations (back row), mRNA concentrations (middle) and promoter states (front) are shown. At this instant, TetR (red) concentration is high, leading to suppression of λcI (yellow). Because λcI is low, however, LacI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

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.

Editor: Dianne P. O'Leary, oleary@cs.umd.edu

Partial Solution to Last Issue's Homework Assignment

BEETLES, CANNIBALISM, AND CHAOS: ANALYZING A DYNAMICAL SYSTEM MODEL

By Dianne P. O'Leary

Last issue's installment of Your Homework Assignment featured the final problem in Dianne O'Leary's popular long-running department. In this issue, she offers a partial solution to it.

By using a *dynamical system* to model a flour beetle's life cycle, we can estimate the key parameters that describe its behavior. In the last issue, we presented the model and defined six parameters.

PROBLEM 1.

To get some experience with this model, plot the populations L, P, and A for 100 days for three sets of data: b = 11.6772, μ_L = 0.5129, c_{el} = 0.0093, c_{ea} = 0.0110, c_{pa} = 0.0178, $L(0)$ = 70, $P(0)$ = 30, $A(0)$ = 70, and μ_A = 0.1, 0.6, and 0.9. Describe the behavior of the populations in these three cases as if you were speaking to someone who isn't looking at the graphs.

Answer:

Figure 1 shows the results. When μ_A = 0.1, the solution eventually settles into a cycle, oscillating between two different values: 18.7 and 321.6 larvae, 156.7 and 9.1 pupae, and 110.1 and 121.2 adults. Thus the population at four-week intervals is constant. Note that the peak pupae population lags two weeks behind the peak larvae population, and that the adult population's oscillation is small compared to the larvae and pupae.

For μ_A = 0.6, the population eventually approaches a fixed point: 110.7 larvae, 54.0 pupae, and 42.3 adults.

In the third case, μ_A = 0.9, there is no regular pattern for the solution, so it's called *chaotic*. The number of larvae varies between 18 and 242, the number of pupae between 8 and 117, and the number of adults between 9 and 94.

PROBLEM 2.

Let μ_L = 0.5, μ_A = 0.5, c_{el} = 0.01, c_{ea} = 0.01, and c_{pa} = 0.01. Plot A_{fixed}, L_{fixed}, and P_{fixed} for b = 1.0, 1.5, 2.0, …, 20.0. To compute these values for each b, use `fsolve`, started from the solution with c_{el} = 0, to solve the equations $\hat{\mathbf{F}}(\mathbf{x}) = \mathbf{F}(\mathbf{x}) - \mathbf{x} = \mathbf{0}$. Provide `fsolve` with the Jacobian matrix for the function $\hat{\mathbf{F}}$; on your plot, mark the b values for stable equilibria with plus signs.

Answer:

Figure 2 shows the results. For the stable solutions, if we start with population values near A_{fixed}, L_{fixed}, and P_{fixed}, we'll converge to these equilibrium values.

PROBLEM 3.

(a) Let μ_L = 0.5128, c_{el} = 0.0, c_{ea} = 0.01, and c_{pa} = 0.09. For μ_A = 0.02, 0.04, …, 1.00, use the LPA relations to determine the population for 250 cycles. On a single graph, plot the last 100 values as a function of μ_A to produce the bifurcation diagram.

(b) Determine the largest of the values μ_A = 0.02, 0.04, …, 1 for which the constant solution is stable (that is, well-conditioned).

(c) Explain why the bifurcation diagram isn't just a plot of L_{fixed} versus μ_A when the system is unstable.

(d) Give an example of a value of μ_A for which nearby solutions cycle between two fixed values. Give an example of a value of μ_A for which nearby solutions are chaotic (or at least have a long cycle).

Answer:

Figure 3 shows the bifurcation diagram. The largest tested value of μ_A that gives a stable solution is 0.58. If we perform the computation in exact arithmetic, the graph would just be a plot of L_{fixed} versus μ_A. When the solution is stable, a rounding error in the computation produces a

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

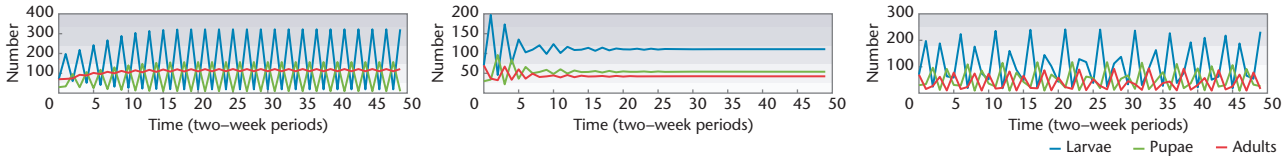


Figure 1. Results of the LPA model with three different choices of μ_A. Model predictions for $b = 11.6772$, $\mu_L = 0.5129$, $c_{el} = 0.0093$, $c_{ea} = 0.0110$, $c_{pa} = 0.0178$, $L(0) = 70$, $P(0) = 30$, $A(0) = 70$, and $\mu_A = 0.1$ (left), 0.6 (middle), and 0.9 (right). Number of larvae is in blue, pupae in green, and adults in red.

Table 1. Parameter estimates computed in Problem 4.

Colony	c_{el}	c_{ea}	c_{pa}	b	μ_L	μ_A	Residual
New: a	0.018664	0.008854	0.020690	5.58	0.144137	0.036097	5.04
Old: a	0.009800	0.017500	0.019800	23.36	0.472600	0.093400	17.19
New : b	0.004212	0.013351	0.028541	6.77	0.587314	0.000005	7.25
Old: b	0.010500	0.008700	0.017400	11.24	0.501400	0.093000	14.24
New : c	0.018904	0.006858	0.035082	6.47	0.288125	0.000062	4.37
Old: c	0.008000	0.004400	0.018000	5.34	0.508200	0.146800	4.66
New: d	0.017520	0.012798	0.023705	6.79	0.284414	0.005774	6.47
Old: d	0.008000	0.006800	0.016200	7.20	0.564600	0.109900	7.42

nearby point from which the iteration tends to return to the fixed point. When the solution is unstable, a rounding error in the computation can cause the computed solution to drift away. Sometimes it produces a solution that oscillates between two values (for example, when $\mu_A = 0.72$), and sometimes the solution becomes chaotic or at least has a long cycle (for example, when $\mu_A = 0.94$).

PROBLEM 4.

(a) Use `lsqnonlin` to solve the least-squares minimization problem, using each of the four sets of data in `beetledata.m`. In each case, determine the six parameters (μ_L, μ_A, c_{el}, c_{ea}, c_{pa}, and b). Set reasonable upper and lower bounds on the parameters and perhaps start the least-squares iteration with the guess $\mu_L = \mu_A = 0.5$, $c_{el} = c_{ea} = c_{pa} = 0.1$, and $b = 10$. Print the solution parameters and the corresponding residual norm.

(b) Compare your results with those that Brian Dennis and his colleagues computed (see `param_dl` in `beetledata.m`). Be sure to include a plot that compares the predicted values with the observed values.

Answer:

I used bounds of 0 and 1 for all parameters except b. For b, I used [0.1, 9.0]. The results are summarized in Tables 1 and 2 and contrast with the results of our model (new) with that of Dennis and his colleagues (old).

Figure 4 shows the predictions obtained from my parameters and from those of Dennis and his colleagues. Note that none of the models gives good predictions; we will see

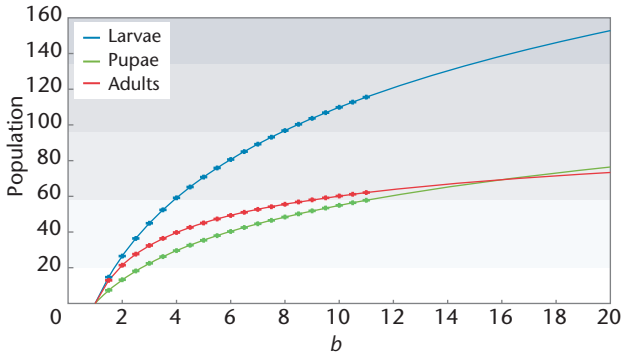


Figure 2. Equilibrium population as a function of b for $\mu_L = 0.5$, $\mu_A = 0.5$, $c_{el} = 0.01$, $c_{ea} = 0.01$, and $c_{pa} = 0.01$, $b = 1.0$, 1.5, 2.0, …, 20.0. Stable solutions are marked with pluses.

Table 2. Residual norms computed in Problem 4.

Colony	Norm of data vector	New residual	Old residual
Colony a	33.55	5.04	17.19
Colony b	33.70	7.25	14.24
Colony c	33.44	4.37	4.66
Colony d	33.68	6.47	7.42

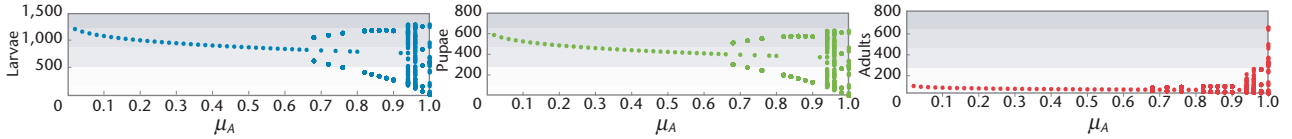


Figure 3. Bifurcation diagram for the data in Problem 3.

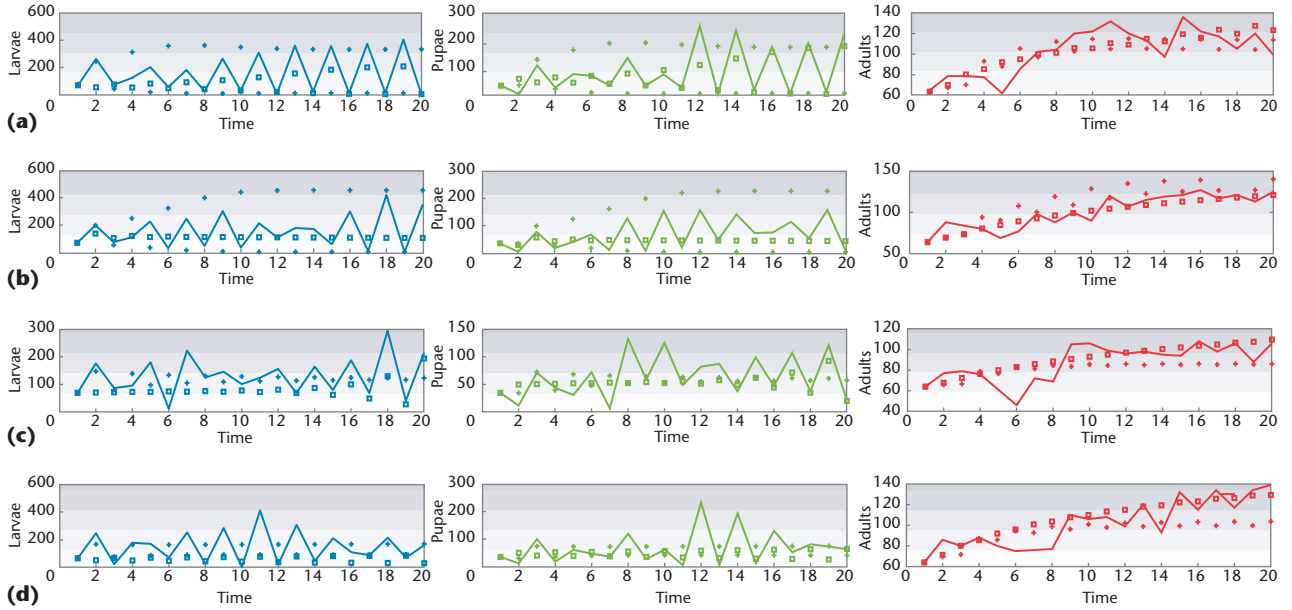


Figure 4. Model predictions for colonies (a) through (d). The solid line represents the data, the pluses are the predictions from Dennis and his colleagues, and the squares are our predictions.

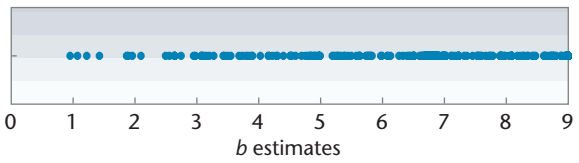


Figure 5. Values of *b* computed for colony b with 250 random perturbations of the log of the data, drawn from a normal distribution with mean 0 and standard deviation 1.

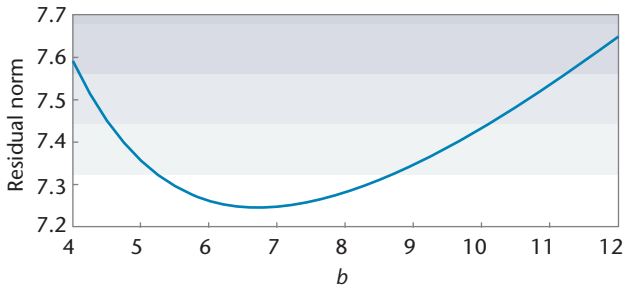


Figure 6. Changes in the residual as *b* is changed for colony b, leaving the other parameters fixed.

later that `lsqnonlin` finds only a locally optimal set of parameters—not necessarily the best choice overall. There is also the possibility of non-modeled errors in the data, and perhaps the beetles didn't respond well to the counting process.

PROBLEM 5

Consider the data for the second beetle colony. For each value $b = 0.5, 1.0, \ldots, 50.0$, minimize the least-squares function by using `lsqnonlin` to solve for the five remaining parameters. Plot the square root of the least-squares function versus b, and determine the best set of parameters. How sensitive is the function to small changes in b?

Perform further calculations to estimate the *forward error*—how sensitive the optimal parameters are to small changes in the data—and the *backward error*—how sensitive the function is to small changes in the parameters.

Answer:

When the data is randomly perturbed, the estimate of b for the second colony ranges from 4.73 to 6.83.

A larger upper bound for b tends to cause the minimizer to converge to a local solution with a much larger residual.

There are many ways to measure sensitivity:

- We might ask how large a change we see in b when the data is perturbed a bit. This is a *forward error* result.
- We might ask how large a change we see in the residual when the value of b is perturbed a bit. This is a *backward error* result.

To estimate the forward error, I repeated the fit after adding 50 samples of normally distributed error (mean 0, standard deviation 1) to the log of the counts. This is only an approximation to the error assumption usually made for counts—Poisson error—but by using the log function in their minimization, the authors are assuming that this is how the error behaves. Even so, the estimate shown in Figure 5 range from 1.00 to 9.00, quite a large change.

To estimate the backward error, I varied b, keeping the other parameters at their optimal values, and plotted the resulting residual versus b in Figure 6. We see that the residual isn't very sensitive to changes in b.

Then I minimized the residual as a function of the five parameters remaining after setting b to fixed values. From Figure 7, we conclude that for any value of b between 1 and 50, we can obtain a residual norm within 10 percent of the computed minimum over all choices of b. This model seems to give no insight into the true value of b.

But as a final attempt, I used a homotopy algorithm, repeating the computations from Figure 7, but starting each minimization from the optimal point found for the previous value of b. The resulting residuals, shown in Figure 8, are much smaller, and the b value is somewhat better determined—probably between 5 and 10. Even more interesting, the fitted model finally gives a reasonable approximation of most of the data (see Figure 9).

To check the reliability of these estimates, it would be a good idea to repeat the experiment for the data for the other three colonies and to repeat the least-squares calculations using a variety of initial guesses.

Dianne P. O'Leary is a professor of computer science and a faculty member in the Institute for Advanced Computer Studies and the Applied Mathematics Program at the University of Maryland. She has a BS in mathematics from Purdue University and a PhD in computer science from Stanford. O'Leary is a member of SIAM and AWM, and a fellow of the ACM. Contact her at oleary@cs.umd.edu; www.cs.umd.edu/users/oleary/.

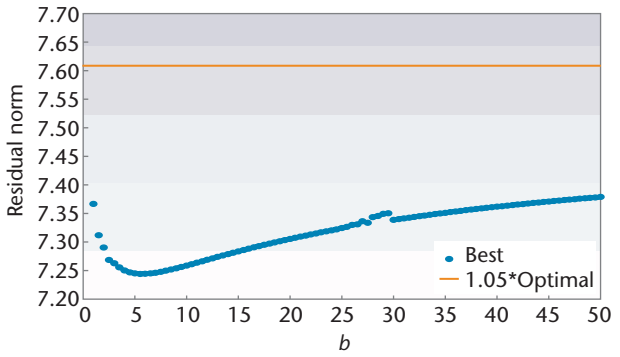


Figure 7. Best (smallest) residuals for colony b computed as a function of the parameter b (blue circles) compared with the red dotted line, indicating a 10 percent increase over the minimal computed residual.

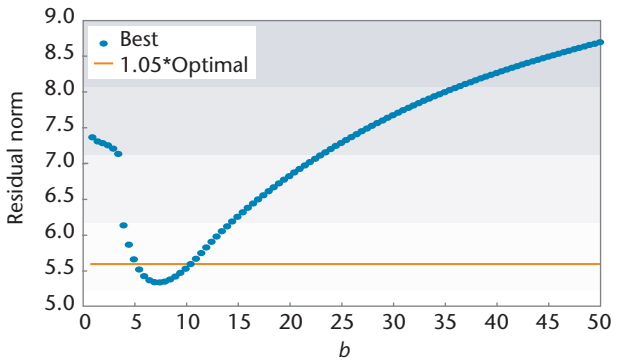


Figure 8. Best (smallest) residuals for colony b computed as a function of the parameter b (blue circles) compared with the red dotted line, indicating a 10 percent increase over the minimal computed residual, using homotopy.

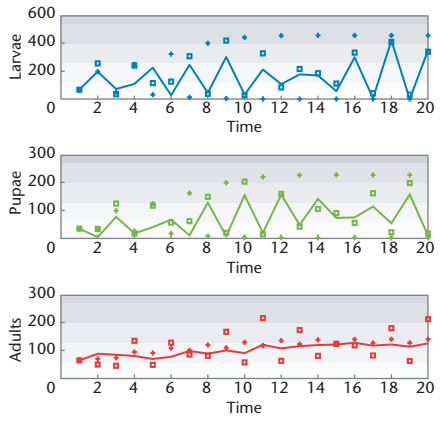


Figure 9. Revised model predictions for colony b, with parameters $c_{el} = 0.008930$, $c_{ea} = c_{pa} = 0$, $b = 7.5$, $\mu_L = 0.515596$, $\mu_A = 0.776820$. The solid line represents the data, the pluses are the predictions from Dennis and his colleagues, and the squares are our predictions.

Editor: Isabel Beichl, isabel.beichl@nist.gov

MAKING THE COMPLEX SIMPLE

By Julian V. Noble

This installment of Computing Prescriptions illustrates how complex arithmetic can simplify algorithms in two-dimensional Cartesian vector space as well as how to make difficult numerical integrals tractable. In other words, computer languages for scientific applications should support complex arithmetic.

A lot of the mathematical work I do is best expressed in complex variables, so I'm partial to programming languages that support complex arithmetic and that possess a library of standard complex functions. It's no fun to program complex arithmetic in languages lacking such extensions: the resulting code is cumbersome, opaque, and hard to maintain. Basic, C, and Pascal never pretended to support complex types, but it was a shock when the ostensibly compliant Fortran 77 I bought for my PC back in the mid-1980s didn't support complex arithmetic either.

Complex arithmetic *can*, of course, be expressed as real-number arithmetic, but the result is messy. As the *Numerical Recipes in C*[1] file `complex.c` makes clear, each complex operation—even addition and multiplication—requires a separate subroutine. Thus I was driven to work with extensible languages such as Lisp and Forth, for which I could write my own complex extensions.

Today, Python, C++, and other languages besides Fortran (even the new C99 standard) support complex types and operators, so rolling one's own is no longer necessary. The disadvantage of this, however, is that there's no standard (to my knowledge) for the behavior of the supplied libraries in the various languages (or even within a single open source language). Hence, results might not be correct, portable, or predictable.

Because I contemplate future forays into the complex plane, it seemed reasonable to introduce the subject with applications in which complex arithmetic simplifies the programming. I chose vector analysis in two dimensions and evaluating oscillatory integrals numerically via contour deformation.

Complex Arithmetic

The complex numbers

$$z = x + iy, \quad i^2 = -1,$$

defined in terms of real number pairs (x, y), were introduced to provide solutions to such polynomial equations as $x^2 - 2x + 5 = 0$, which obviously can't be satisfied by any real number substituted for x but is satisfied by the two complex roots $z = 1 \pm 2i$.

In $z = x + iy$, we call x the *real* part of z, denoted by $\mathrm{Re}(z)$; for historical reasons, we call y the *imaginary* part, denoted by $\mathrm{Im}(z)$. The usual laws of algebra define operations such as addition, multiplication, and division. For example,

$$(x + iy)(u + iv) = xu + iyu + ixv + i^2yv \equiv (xu - yv) + i(yu + xv).$$

That is, when we multiply two complex numbers, the result is another complex number, and similarly with addition. (A mathematician would say the complex numbers are *closed* under addition and multiplication.) One important new operation we will need is *complex conjugation*, denoted by a * superscript and defined as "reverse the sign of the imaginary part":

$$z^* = (x + iy)^* \stackrel{df}{=} x - iy.$$

Geometric Interpretation

We can regard the complex number $z = x + iy$ as a vector $\mathbf{z} = x\hat{\mathbf{x}} + y\hat{\mathbf{y}}$ in two-dimensional Cartesian coordinates (here $\hat{\mathbf{x}}$ and $\hat{\mathbf{y}}$ are unit vectors in the horizontal, or x, direction and the vertical, or y, direction). This 2D vector space is often called the *Argand plane*. The magnitude of a complex number is the length of its corresponding 2-vector (I use $\mathbf{z}$ to represent a 2-vector and z to represent its corresponding complex number):

$$|z| = |\mathbf{z}| = \sqrt{x^2 + y^2}\,. \tag{1}$$

Figure 1 illustrates these definitions.

Figure 1 also exhibits the polar representation, $z = re^{i\vartheta}$, which exploits Euler's identity,

In Memoriam

Julian V. Noble passed away on 11 March in Charlottesville, Virginia. We knew him for several years through our collaboration on this department. His charm, wit, erudition, and unique view of the world of computing always came through, even in email. When we finally met him in person, we were delighted to learn that he was even better in the flesh than on the screen. Besides our mutual obsession on how computing should be taught, we shared an interest in ancient TV comedies, especially those starring Sid Caesar. Unlike one of us, he remembered more than merely the "look and feel" of those shows; he could quote large chunks of dialogue and do it with panache. Our vigorous debates over the form and content of our department were a joy to us, and we think he liked them, too. We will miss him. *—Francis Sullivan and Isabel Beichl*

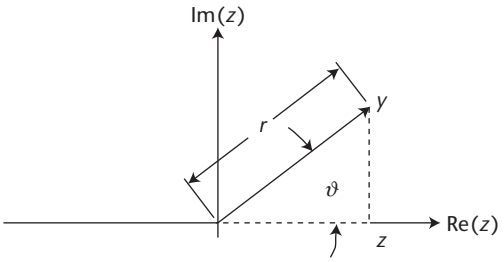


Figure 1. The Argand plane. This plane illustrates both the vector representation of a complex number and its polar representation.

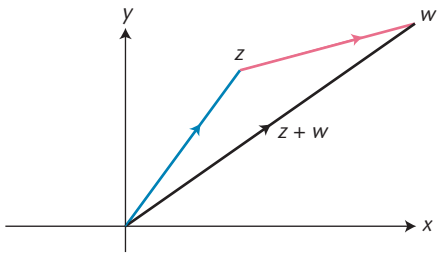


Figure 2. Addition of complex numbers. Adding complex numbers is equivalent to vector addition in the plane.

$$e^{i\vartheta} \equiv \cos\vartheta + i\sin\vartheta,$$

so that

$$x = r\cos\vartheta, \quad y = r\sin\vartheta.$$

Addition of complex numbers,

$$z + w = (x + iy) + (u + iv) = (x + u) + i(y + v),$$

is equivalent to vector addition in the plane, as Figure 2 shows.

Beware of Libraries

Some care is needed when performing complex arithmetic in various languages that now include it—libraries generally have issues the user needs to know. Let's look at two examples.

A naive definition of, say, the absolute value of a complex number,

$$|z| = \sqrt{x^2 + y^2}$$

is easy (as usual, I use a Fortran-ish *cum* C-ish pseudocode for code fragments):

```
real function cabs (z)
  return sqrt((Re(z))^2 + (Im(z))^2)
```

but it would never do in a library. If, say, Im(*z*) were larger than the square root of the maximum floating-point number, then $(\mathrm{Im}(z))^2$ would overflow.[2] Unfortunately, although the C99 Standard[3] is aware of such problems, I found the naive version in more than one implementation of the library `<complex.h>`. (The function isn't defined in Python, as far as I know.) To minimize overflow problems, the correct way to write the code is

```
real function cabs (z)
  complex z
  if z = cmplx(0,0) return 0 \ check for z = 0
  else
    x = max(abs (Re(z)), abs (Im(z)))
    y = min(abs (Re(z)), abs (Im(z)))
    return x * sqrt(1 + (y/x)2)
```

There's a speed hit relative to the naive version (because of the extra division), but that's better than unexpected overflows.

A second problem in writing a complex function library has to do with the singularities known as "branch cuts," in complex-speak. (A *singularity* is a place where the function behaves badly—for example, it might be infinite or discontinuous.) Consider the complex logarithm function. It's naturally defined as

$$\log(x + iy) = \ln(r) + i\vartheta \equiv \ln\left(\sqrt{x^2 + y^2}\right) + i\mathrm{Arctan}(y / x).$$

Of course, we should use the overflow-insensitive `cabs` function to compute

$$\mathrm{Re}[\log(x + iy)] \overset{df}{=} \ln\left(\sqrt{x^2 + y^2}\right)$$

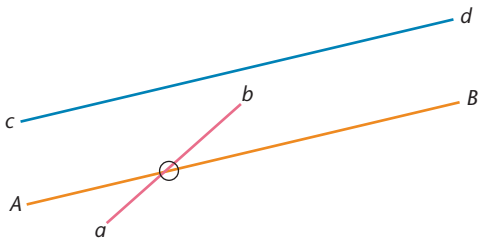


Figure 3. Line segments. Some of the segments cross and some don't.

(although this wasn't done in the libraries I examined), but the real issue is the imaginary part, involving the Arctan function. Many implementations use the library function atan2(x, y) analogous to the venerable Fortran version, whose range is $(-\pi, \pi)$. This automatically places the branch cut along the negative real axis. I hasten to add that there's nothing wrong with this, but a user expecting the branch cut to run along, say, the *positive* real axis could obtain nonsense.

Geometric Applications

The one-to-one correspondence between complex numbers and Cartesian 2-vectors lets us replace vector operations—usually expressed as matrix multiplication—with simple complex arithmetic. For example, we rotate a 2-vector counterclockwise by angle φ via

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} \cos\varphi & -\sin\varphi \\ \sin\varphi & \cos\varphi \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}.$$

Complex arithmetic simplifies this to $z' = e^{i\varphi}z \equiv (\cos\varphi + i\sin\varphi)z$.

The dot product of two vectors is also simple:

$$\mathbf{z} \cdot \mathbf{w} = xu + yv,$$

where

$$\mathbf{z} = \begin{pmatrix} x \\ y \end{pmatrix}, \quad \mathbf{w} = \begin{pmatrix} u \\ v \end{pmatrix}.$$

In most languages, we have to write the dot product as a function or subroutine, but with complex arithmetic, we can write it as a simple multiplication:

$$\mathbf{z} \cdot \mathbf{w} = xu + yv = \mathrm{Re}[(x - iy)(u + iv)] \equiv \mathrm{Re}(z^* w).$$

Similarly, we can compute the vector product of two 2-vectors (the usual vector product points perpendicular to the x–y plane; I label that direction $\hat{\zeta}$ to distinguish it from the complex number z):

$$(\mathbf{z} \times \mathbf{w}) \cdot \hat{\zeta} = xv - yu \equiv \mathrm{Im}(z^* w).$$

That is, one complex multiplication, $z^* w$, evaluates simultaneously both the dot and vector products of the 2-vectors represented by z and w.

Intersecting Lines

To remove hidden lines in an image, we need to determine whether two line segments in a plane intersect, and if so, where. Two line segments can be represented parametrically in vector notation by $\mathbf{z}(t) = \mathbf{a} + (\mathbf{b} - \mathbf{a})t$, $\mathbf{w}(u) = \mathbf{A} + (\mathbf{B} - \mathbf{A})u$, where **a**, **b** and **A**, **B** are 2-vectors representing the segments' endpoints. In complex notation,

$$z(t) = a + (b - a)t, \quad w(u) = A + (B - A)u,$$

where a, b and A, B are complex numbers representing the segments' endpoints. In either representation, the parameters t and u are real numbers in the interval [0, 1].

We must first decide whether the lines are parallel because parallel lines don't intersect. The cross-product of parallel vectors vanishes, so we compute

$$[\mathbf{z}(t) - \mathbf{z}(0)] \times [\mathbf{w}(u) - \mathbf{w}(0)] \Leftrightarrow \mathrm{Im}[(z(t) - z(0))^* (w(u) - w(0))]. \quad (2)$$

If the cross-product doesn't equal zero, the (infinitely extended) lines will intersect somewhere. We can therefore locate the point of intersection and determine whether it corresponds to values of t and u, both of which lie between 0 and 1 (that is, whether the intersection point is common to both segments). The segments cross if and only if this condition is met. Line segments $\widehat{ab}$ and $\widehat{AB}$ (crossing segments) illustrate this case; Figure 3 also shows the segments $\widehat{ab}$ and $\widehat{cd}$ (non-parallel, non-crossing) and $\widehat{AB}$ and $\widehat{cd}$ (parallel, non-crossing).

In vector notation, the point of intersection for two lines is given by $\mathbf{a} + (\mathbf{b} - \mathbf{a})t = \mathbf{A} + (\mathbf{B} - \mathbf{A})u$.

This represents the two simultaneous algebraic equations in the two unknowns, t and u:

$$\begin{pmatrix} b_x - a_x & A_x - B_x \\ b_y - a_y & A_y - B_y \end{pmatrix} \begin{pmatrix} t \\ u \end{pmatrix} = \begin{pmatrix} A_x - a_x \\ A_y - a_y \end{pmatrix}. \quad (3)$$

We solve this in the usual fashion, leading to algebraically complicated formulas for t and u.

Suppose we tackle the same problem using complex arithmetic. The intersection point is then $a + (b - a)t = A + (B - A)u$, so multiplying through by $(b - a)^*$, we have

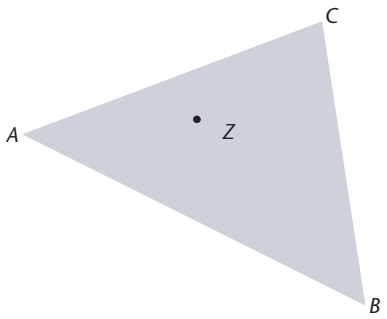


Figure 4. A point within a triangle. We want to determine whether point *Z* lies inside Δ_{ABC} (Figure 5 shows how to do so).

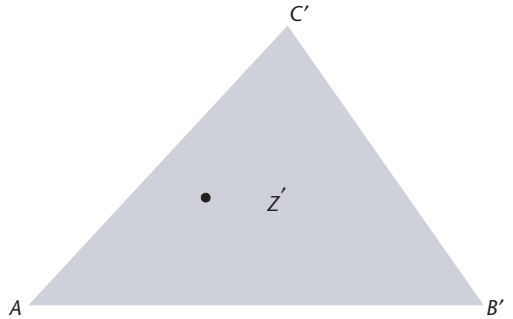


Figure 5. The point and the triangle after a rotation. Rotate the triangle and point *Z* about point *A* until side $\widehat{A'B'}$ is horizontal. Does *Z′* lie above the line $\widehat{A'B'}$?

$$(b-a)^*(b-a)t = |b-a|^2 t = (b-a)^*(A-a) + (b-a)^*(B-A)u.$$

Taking the imaginary parts of both sides, and recalling that t, u, and $|b-a|^2$ are all real, we find

$$u = \frac{\mathrm{Im}[(b-a)^*(A-a)]}{\mathrm{Im}[(B-A)^*(b-a)]} \qquad (4a)$$

and, *mutatis mutandis*,

$$t = \frac{\mathrm{Im}[(B-A)^*(A-a)]}{\mathrm{Im}[(B-A)^*(b-a)]}\ . \qquad (4b)$$

The denominators are the same for t and u. In fact, they're the determinant of the 2×2 matrix in Equation 3 as well as our criterion for parallelism. Because $\mathrm{Im}[(B-A)^*(b-a)] \neq 0$—that is, the lines aren't parallel—we aren't dividing by zero.

The expressions for t and u in complex arithmetic are rather simple compared to what we get by solving vectorial equations component-wise. The advantage of using complex arithmetic for such tasks is that it hides complexity, leading to a simpler, more maintainable program.

Point in a Triangle

Next, we'll determine whether a given point lies within a given triangle. Label the corners of the triangle with complex numbers A, B, and C as in Figure 4.

If we rotate the line $\widehat{AB}$ about the point A until it becomes horizontal (taking the sense of rotation to leave the rotated point C' above $\widehat{A'B'}$), the rotated point Z' would lie above the line $\widehat{A'B'}$ if it lies within the triangle. Figure 5 shows how it should look.

Although a point Z' outside the triangle might lie above one (rotated) side—say, $\widehat{A'B'}$—an outside point can't simultaneously lie "above" all three (rotated) sides. Thus if we repeat the rotation for the sides $\widehat{BC}$ and $\widehat{CA}$, an exterior Z' will end up below at least one rotated side. We therefore transform Z once per side, each time inquiring whether the new Z' lies above the corresponding side. If Z is "above" all three sides, it must be an interior point of Δ_{ABC}.

To transform Z relative to a given line $\widehat{AB}$ (by translating the origin to A and then rotating about A until $\widehat{A'B'}$ is horizontal), we write $B - A = |B-A|e^{i\vartheta}$, and thus

$$Z' = e^{-i\vartheta}(Z-A) = \frac{(B-A)^*}{|B-A|}(Z-A).$$

Because we need only the algebraic sign of $\mathrm{Im}(Z')$, we can eliminate the division by the (positive) length $|B-A|$. We're left with the criterion

$$\mathrm{Im}[(B-A)^*(Z-A)] > 0.$$

However, we might have rotated the wrong way, so that the point C' now lies *below* $\widehat{A'B'}$ rather than above. We could do more geometry to get the correct sense of rotation, but it seems simpler to apply the transformation once to Z and once to C and to multiply the criteria. We also have to check that the triangle's area is greater than some minimum because we aren't interested in degenerate triangles.

Another useful application in which complex arithmetic greatly simplifies the programming is motion in two dimensions. Unfortunately space doesn't permit me to illustrate this here.

Numerical Integration in the Argand Plane

In an earlier installment of Computing Prescriptions,[4] I demonstrated how we could simplify a numerically difficult real-valued Cauchy principal value integral

$$I = \mathcal{P}\int_0^\infty dx\,\frac{1}{1-x^3}$$

by rotating to the contour $z = re^{-i\pi/3}$ and integrating in the complex plane. The ability to do this derives from Cauchy's Theorem,[5,6] which states that the contour integral of a function $f(z)$ around a simple contour Γ vanishes if the function is analytic within and continuous on the contour:

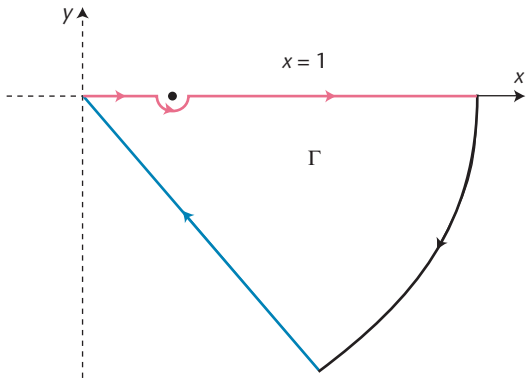


Figure 6. Deformed contour for the Cauchy principal value integral. We can replace the integral along the real line (pink) by an integral in the complex plane, along the deformed contour (blue line plus black circle).

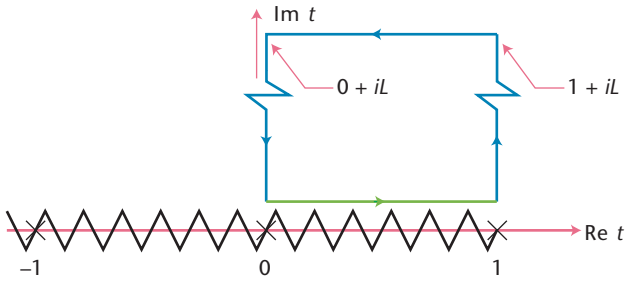


Figure 7. The original (green) and deformed (blue) contours of an oscillatory numerical integral. We replace the integral along the original line by one along the deformed contour because, for positive real *A* that changes the oscillatory behavior of the integrand to exponentially decreasing behavior, making it numerically much easier to evaluate.

$$\oint_{\Gamma} f(z)dz = 0. \tag{5}$$

Figure 6 illustrates the application of Cauchy's Theorem to this integral. The original contour (pink) avoids the pole at $x = 1$ with a small semicircle that will shrink to zero radius. The contour then continues with the black arc whose radius R will become infinite. The integral on this arc is of $\mathcal{O}(R^{-2})$ and so vanishes in that limit. Finally, the deformed (blue) contour is the line $z = re^{-i\pi/3}$. Because the integral around the entire contour is zero, the integrals on the original and deformed contours are equal. We get the desired (principal value) integral by comparing their real parts.

Consider the rather difficult integral

$$I_B(A) = \mathrm{Re}\int_0^{\pi/2} d\vartheta e^{iA\cos\vartheta}(\sin\vartheta)^B,$$

with A large and positive. Up to an uninteresting factor $I_B(A)$ is a Bessel function of argument A and order B.[7] When A is large, it's hard to evaluate $I_B(A)$ precisely because the integrand oscillates fiercely, leading to a sum consisting of small differences between large numbers. Evaluation of such oscillatory integrals seems to be a topic of some current interest.[8]

Because we're trying to speed up this calculation, relative to brute-force numerical integration, it pays to eliminate computationally expensive trig functions by the change of integration variable $t = \cos\vartheta$. The transformed integral is

$$\begin{aligned} I_B(A) &= \mathrm{Re}\int_0^1 dt\ e^{iAt}(1-t^2)^{(B-1)/2} \\ &\equiv \mathrm{Re}\int_0^1 dt\ e^{iAt}(1-t^2)^{\lambda}. \end{aligned} \tag{6}$$

The function $(1 - t^2)^{\lambda}$ has (for noninteger λ) branch points at t = ±1; we can arrange the corresponding branch lines to run along the real t-axis from $-\infty$ to +1. We're integrating along the green line $t = x + i\varepsilon$, $0 \le x \le 1$ in Figure 7. Let's add to the original line the sides and top (shown in blue) of the tall, thin rectangle of base 1 and height L to make a closed contour. According to Cauchy's Theorem (Equation 5), the integral around the closed rectangle is zero because the integrand is analytic within it. We can therefore write

$$\begin{aligned} I_B(A) = &\lim_{L\to\infty}\int_{i0+}^{iL} dt\ e^{iAt}((1-t^2)^{\lambda} - e^{iA}(1-(1+t)^2)^{\lambda}) \\ &+ \lim_{L\to\infty} e^{-AL}\int_0^1 dx\ e^{iAx}(1-(x+iL)^2)^{\lambda}. \end{aligned}$$

The second integral (the line closing the top, from 1 + iL to 0 + iL) can be neglected in the limit $L \to \infty$ because it's bounded in magnitude by

$$\begin{aligned} &\left|e^{-AL}\int_0^1 dx\ e^{iAx}(1-(x+iL)^2)^{\lambda}\right| \le \\ &e^{-AL}\int_0^1 dx\left|e^{iAx}\right|\left|(1-(x+iL)^2)^{\lambda}\right| \\ &\le e^{-AL}\int_0^1 dx(3+L^2)^{\lambda} \\ &= (3+L^2)^{\lambda}e^{-AL} \underset{L\to\infty}{\to} 0. \end{aligned}$$

Another change of variable $t \to iy$ lets us express $I_B(A)$ as

$$\begin{aligned} I_B(A) &= \mathrm{Re}\, i\int_0^{\infty} dy\ e^{-Ay}[(1+y^2)^{\lambda} - e^{iA}(y(y-2i))^{\lambda}] \\ &= \mathrm{Im}\, e^{iA}\int_0^{\infty} dy\ e^{-Ay}(y(y-2i))^{\lambda}, \end{aligned} \tag{7}$$

where the second line of Equation 7 follows from the as-

Table 1. Quadrature by real and complex arithmetic.

	Real		Complex	
A	N_{calls}	$I_{(A)}$	N_{calls}	$I_{(A)}$
10	2,445	7.31477251207E-3	1,545	7.31477251218E-3
20	3,601	–2.19385695893E-5	373	–2.19385695736E-5
30	4,917	8.87060992799E-7	177	8.87060992040E-7

Table 2. Adaptive vs. Gauss-Laguerre quadrature.

	Adaptive		10-pt Gauss-Laguerre
A	N_{calls}	$I(A)$	$I(A)$
10	1,545	7.31477251218E-3	7.31477251203E-3
20	373	–2.19385695736E-5	–2.19385695900E-5
30	177	8.87060992040E-7	8.87060992893E-7

sumption that λ is real (for complex λ, we must keep both terms from the first line). The convergence of the integral improves as A increases, quite the opposite behavior we encounter in evaluating the original oscillating integral (see Table 1).

We have thereby replaced an integrand that definitely converges (it oscillates in sign and decreases in magnitude) with one that decays exponentially, for which a Gauss-Laguerre quadrature formula would be appropriate. Gauss-Laguerre quadrature[9] approximates an integral on the interval $[0, \infty)$ in the form

$$\int_0^\infty f(x)e^{-x}dx \approx \sum_{n=1}^{N} f(\xi_n)w_n,$$

where the points ξ_n and weights w_n are free parameters—that is, a quadrature formula of order N will integrate the polynomial

$$f(x) = a_0 + a_1x + \ldots + a_{2N-1}x^{2N-1}$$

exactly. To integrate a function that behaves like e^{-Ax}, we simply rescale: $t = Ax$.

Table 1 lists values of the integral Equation 6 for several large values of A and $\lambda = 7$, computed in real arithmetic using adaptive quadrature. Table 1 also lists values of the transformed integral Equation 7 computed by using adaptive (Simpson's rule) quadrature. It also shows the number of evaluations of the integrand. Table 2 compares the evaluation of the transformed integral by adaptive quadrature with the results of a 10-point Gauss-Laguerre rule. The absolute error criterion for the adaptive quadrature was 10^{-12}; as Table 1 shows, both the real and complex adaptive methods agree with each other to this precision; the 10-point Gauss-Laguerre method agrees with both to this precision, and it requires only 10 evaluations of the integrand, so it's the method of choice.

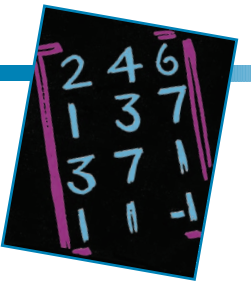
Deforming the integration contour of an oscillatory integral into the complex plane is a trick I've long found useful. Even when the integrand involves the solution of a differential equation, the method still works because most such equations are sufficiently analytic that they can be continued (numerically) into the complex plane. The key issue is to make sure that in deforming the contour of integration, we cross no singularities. CiSE

References

1. W.H. Press et al., *Numerical Recipes in C*, 2nd ed., Cambridge Univ. Press, 1992, Appendix C.
2. D. Goldberg, "What Every Computer Scientist Should Know about Floating-Point Arithmetic," *ACM Computing Surveys*, vol. 23, no. 1, 1991, pp. 5–48.
3. *C99 Draft Standard*, Joint Tech. Committee ISO/IEC JTC1, Information Technology, Subcommittee SC22, 6 May 2005; www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf.
4. J.V. Noble, "Gauss-Legendre Principal Value Integration," *Computing in Science & Eng.*, vol. 2, no. 1, 2000, pp. 92–95.
5. E.T. Copson, *An Introduction to the Theory of Functions of a Complex Variable*, Clarendon Press, 1955.
6. E.T. Whittaker and G.N. Watson, *A Course of Modern Analysis*, 4th ed., Cambridge Univ. Press, 1963.
7. M. Abramowitz and I.A. Stegun, *Handbook of Mathematical Functions*, Dover Publications, 1965, p. 360.
8. G.A. Evans and J.R. Webster, "A Comparison of Methods for the Evaluation of Highly Oscillatory Integrals," *J. Computational and Applied Mathematics*, vol. 112, 1999, pp. 55–69.
9. M. Abramowitz and I.A. Stegun, *Handbook of Mathematical Functions*, Dover Publications, 1965, p. 890.

Julian V. Noble was professor emeritus of physics at the University of Virginia's Department of Physics. His interests were eclectic, both in and out of physics. His philosophy of teaching computational methods was "no black boxes." Noble passed away on 11 March in Charlottesville, Va.

Editors: George K. Thiruvathukal, gkt@cs.luc.edu
Konstantin Läufer, laufer@cs.luc.edu



MATPLOTLIB: A 2D GRAPHICS ENVIRONMENT

By John D. Hunter

Matplotlib is a 2D graphics package used for Python for application development, interactive scripting, and publication-quality image generation across user interfaces and operating systems.

I began learning Python in 2001, mainly as a way to procrastinate during the final stages of preparing my dissertation. I was hooked in short order. My numerical and statistical workflow at the time was a mix of Fortran, C++, and Matlab; I used the file system to communicate between the three. Python's ability to integrate seamlessly and transparently with high-performance compiled code dramatically simplified this workflow, but I couldn't quite wean myself from the breadth, quality, and ease of use that the Matlab environment offered for graphics.

Around that time, I began a fairly substantial project that involved writing an application to analyze human electrocorticography (ECoG) signals registered with 3D medical image data, and I debated long and hard about using Python or Matlab. On balance—and after much hand wringing—Matlab won, primarily because of its excellent graphics and secondarily because of its widespread use in the ECoG community. The application quickly grew in complexity, ultimately incorporating 3D medical image visualizations, 2D ECoG displays, spectral and time-series analyses, and the data structures required to represent human subject data. The networking support included data files served up over HTTP, metadata served up over MySQL, and some Web Common Gateway Interface (CGI) forms thrown into the mix for good measure. Not everyone knows that Matlab embeds its own Java Virtual Machine (JVM), which makes it possible to handle all of these things, but making them work together became increasingly painful, and I eventually hit the wall and decided to start all over again in Python.

The first step was to find a suitable replacement for the Matlab 2D graphics engine (the Visualization Toolkit [VTK] in Python provided 3D-visualization support that was more than adequate for my purposes). Although a score of graphics packages were and are readily available for Python, none met all my needs: they had to be embeddable in a GUI for application development, support different platforms, offer extremely high-quality raster and vector (primarily PostScript) hardcopy output for publication, provide support for mathematical expressions, and work interactively from the shell.

I wrote matplotlib to satisfy these needs, concentrating initially on the first requirement so I could get up and running with my ECoG application (the `pbrain` component of the "neuroimaging in Python project" at http://nipy.scipy.org; also, see p. 52 in this issue) and then gradually adding support for the others, with generous contributions from the matplotlib community. Because I was intimately familiar with Matlab and happy with its graphics environment, I followed the advice of Edward Tufte ("copy the great architectures") and T.S. Elliot ("talent imitates, but genius steals") and reverse-engineered the basic Matlab interface. Figure 1 shows a screenshot of the `pbrain` ECoG viewer I wrote in matplotlib.

The latest release of matplotlib runs on all major operating systems, with binaries for Macintosh's OS X, Microsoft Windows, and the major Linux distributions; it can be embedded in GUIs written in GTK, WX, Tk, Qt, and FLTK; has vector output in PostScript, Scalable Vector Graphics (SVG), and PDF; supports TeX and LaTeX for text and mathematical expressions; supports major 2D plot types and interactive graphics, including *xy* plots, bar charts, pie charts, scatter plots, images, contouring, animation, picking, event handling, and annotations; and is distributed under a permissive license based on the one from the Python Software Foundation. Along with a large community of users and developers, several institutions also use and support matplotlib development, including the Space Telescope Science Institute and the Jet Propulsion Laboratory.

Getting Started: A Simple Example

Matplotlib has a Matlab emulation environment called

PyLab, which is a simple wrapper of the matplotlib API. Although many die-hard Pythonistas bristle at PyLab's Matlab-like syntax and its `from pylab import *` examples, which dump the PyLab and NumPy functionality into a single namespace for ease of use, this feature is an essential selling point for many teachers whose students aren't programmers and don't want to be: they just want to get up and running. For many of these students, Matlab is the only exposure to programming they've ever had, and the ability to leverage that knowledge is often a critical point for teachers trying to bring Python into the science classroom. In Figure 2 , I've enabled the `usetex` parameter in the matplotlib configuration file so LaTeX can generate both the text and the equations.

Let's look at a sample session from IPython, the interactive Python shell that is matplotlib-aware in PyLab mode (also see p. 21 in this issue):

```
> ipython -pylab
IPython 0.7.3 — An enhanced Interactive Python.
   Welcome to pylab, a matplotlib-based
   Python environment. For more information,
   type 'help(pylab)'.

In [1]: subplot(111)
In [2]: t = arange(0.0,3.01,0.01)
In [3]: s = sin(2*pi*t)
In [4]: c = sin(4*pi*t)
In [5]: fill(t, s, 'blue', t, c, 'green',
   alpha=0.3);
In [6]: title(r'\TeX\ is
No.$\displaystyle\sum_{n=1}^\infty
   \frac{-e^{i\pi}}{2^n}$!')
```

IPython detects which GUI windowing system you want to use by inspecting your matplotlib configuration, imports the PyLab namespace, and then makes the necessary threading calls so you can work interactively with a GUI mainloop such as GTK's.

Images, Color Mapping, Contouring, and Color Bars

In addition to simple line plots, you can fairly easily create more sophisticated graphs, including color-mapped images with contouring and labeling, in just a few lines of Python. For pseudocolor images, matplotlib supports various image-interpolation and color-mapping schemes. For interpolation, you can choose "nearest" (which does a nearest neighbor interpolation for those who just want to see their raw data), "bilinear," "bicubic," and 14 other interpolation methods for smoothing data. For color mapping, all the classic color maps from Matlab are available (`gray`, `jet`, `hot`, `copper`, `bone`, and so on) as well as scores more. You can also define custom color maps.

Let's look at a Python script that computes a bivariate Gaussian distribution plotted as a grayscale image and then overlays contour lines using the heated object scale `hot` color map:

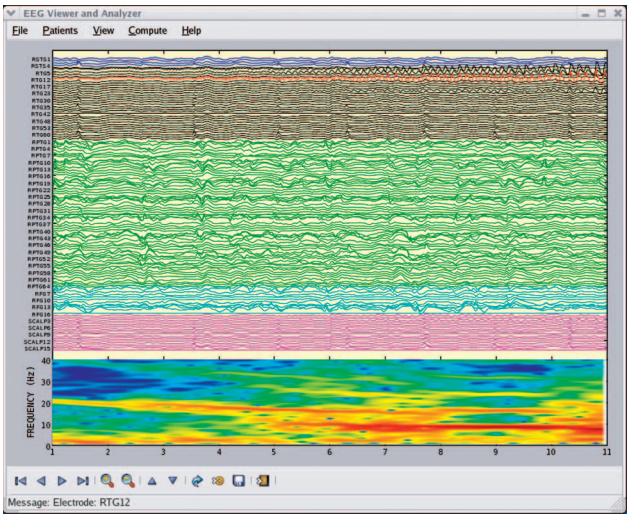


Figure 1. The PBrain project. An electrocorticography (ECoG) viewer, written in matplotlib and embedded in a `pygtk` application.

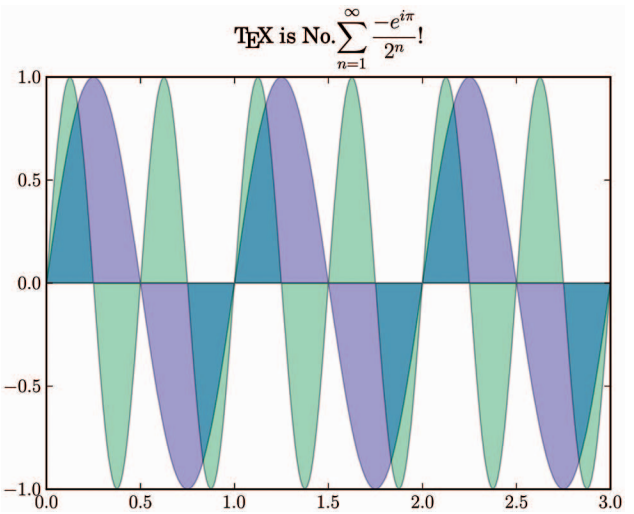


Figure 2. LaTeX support. Setting the `usetex` option in the matplotlib configuration file enables LaTeX generation of the text and equations in a matplotlib figure, as this screenshot shows.

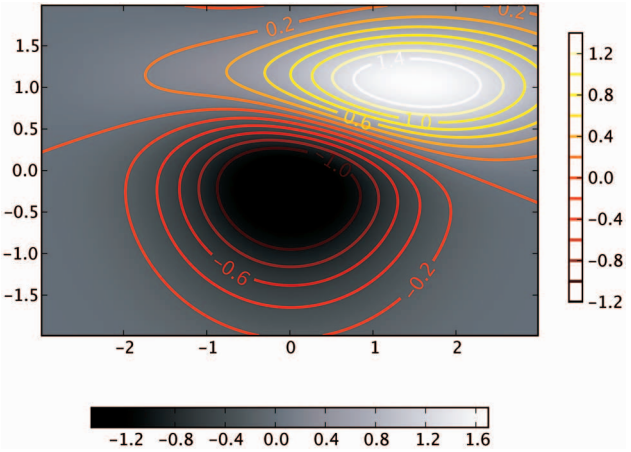


Figure 3. Images, contours, and color mapping. This screenshot from matplotlib illustrates how to add contour lines to luminosity images; note that the use of multiple color maps (`gray` and `hot`) and color bars (`continuous` and `discrete`) are supported.

```
from pylab import figure, cm, nx, show
from matplotlib.mlab import meshgrid, \
   bivariate_normal

delta = 0.025
x = nx.arange(-3.0, 3.0, delta)
y = nx.arange(-2.0, 2.0, delta)
X, Y = meshgrid(x, y)
Z1 = bivariate_normal(X, Y, 1.0, 1.0, 0.0, 0.0)
Z2 = bivariate_normal(X, Y, 1.5, 0.5, 1, 1)
Z = 10.0 * (Z2 - Z1) # difference of Gaussians

fig = figure()
ax = fig.add_subplot(111)

# make a grayscale image
im = ax.imshow(Z, interpolation='bilinear',
   cmap=cm.gray, extent=(-3,3,-2,2),
   origin='lower')
levels = nx.arange(-1.2, 1.6, 0.2)

# do a contour using a "hot" colormap for
# the lines
cs = ax.contour(Z, levels, linewidths=2,
   cmap=cm.hot, extent=(-3,3,-2,2),
   origin='lower')

# label every 2nd contour inline
ax.clabel(cs, levels[1::2], inline=1,
   fmt='%1.1f')

# make a colorbar for the contour lines
cbar = fig.colorbar(cs, shrink=0.8,
   extend='both')

# we can still add a colorbar for the image, too.
cbarim = fig.colorbar(im, orientation=
   'horizontal', shrink=0.8)

# This makes the original colorbar look a bit
# out of place, so let's improve its position.
l,b,w,h = ax.get_position()
ll,bb,ww,hh = cbar.ax.get_position()
cbar.ax.set_position([ll, b+0.1*h, ww, h*0.8])

show()
```

Figure 3 shows the output of this Python script, with multiple color maps and color bars supported in a single axes, as well as continuous (the horizontal `gray` bar) and discrete color bars (the vertical `hot` bar).

Interactive Plotting

To facilitate interactive work, matplotlib provides access to basic GUI events, such as `button_press_event`, `mouse_motion_event`, `key_press_event`, `draw_event`, and so on; you can also register with these events to receive callbacks. In addition to the GUI-provided information, we attach matplotlib-specific data—if you connect to the `button_press_event`, for example, you can get the button press's *x* and *y* location in the display space, the `xdata` and `ydata` coordinates in the user space, which axes the click occurred in, and the underlying GUI event that generated the callback.

Event Handling

Matplotlib abstracts GUI event handling across the five major GUIs it supports, so event-handling code written in matplotlib works across many different GUIs. Let's look at a simple example that reports the *x* and *y* locations in the display and user spaces with a mouse click:

```
from pylab import figure, show
fig = figure()
ax = fig.add_subplot(111)
ax.plot([1,2,3])

def onpress(event):
  if not event.inaxes: return
  print 'click'
  print '\tuser space: x=%1.3f, y=%1.3f'%(
```

```
      event.xdata, event.ydata)
  print '\tdisplay space: x=%1.3f, y=%1.3f'%(
      event.x, event.y)

fig.canvas.mpl_connect('button_press_event',
   onpress)
show()
```

Basic events like button and key presses, mouse motion, and canvas drawing are supported across the five GUIs that matplotlib supports, thus matplotlib code written for one GUI will port without changes to another.

Picking

All matplotlib `Artist` primitives define a method `pick` that supports picking so that users can interactively select objects in the plot scene via mouse clicks. Users can also define threshold tolerance criteria in distance units—for example, to define a hit if the mouse click is within five printer's points of the graphical object—or provide a custom hit-testing function. Let's print the data points that fall within five points of the mouse-click location:

```
from pylab import figure, nx, show
fig = figure()
ax1 = fig.add_subplot(111)

# 5 points tolerance
line, = ax1.plot(nx.mlab.rand(100), 'o',
   picker=5)

def onpick(event):
  line = event.artist
  xdata = line.get_xdata()
  ydata = line.get_ydata()
  ind = event.ind
  print 'data: ', zip(nx.take(xdata, ind),
     nx.take(ydata, ind))

fig.canvas.mpl_connect('pick_event', onpick)
show()
```

More elaborate examples, such as defining custom hit-testing functions, are located in the examples directory of the matplotlib source distribution.

Matplotlib Toolkits

Matplotlib supports toolkits for domain-specific plotting functionality that's either too big or too narrow in purpose for the main distribution. Jeffrey Whitaker of the US National Oceanic and Atmospheric Association offers the excellent `basemap` toolkit for plotting data on map projections (http://matplotlib.sourceforge.net/matplotlib.toolkits.basemap.basemap.html). Some of the available projections include cylindrical equidistant, mercator, lambert conformal conic, lambert azimuthal equal area, albers equal area conic, stereographic, and many others. The `basemap` toolkit ships with the `proj4` library and Python wrappers to do the projections. Coastlines, political boundaries, and rivers are available in four resolutions: crude, low, intermediate, and high. The toolkit also supports contouring and annotations. Figure 4 shows an orthographic map projection of the Earth from the perspective of a satellite looking down at 50N, 100W using low-resolution coastlines and from this code:

```
from matplotlib.toolkits.basemap import Basemap
from pylab import nx, show

# don't plot features that are smaller than
# 1000 square km.
map = Basemap(projection='ortho',lat_0=50,
   lon_0=-100, resolution='l',
   area_thresh=1000.)

# draw coastlines, country boundaries, fill
# continents.
map.drawcoastlines()
map.drawcountries()
map.fillcontinents(color='coral')

# draw the edge of the map projection region
# (the projection limb)
map.drawmapboundary()

# draw lat/lon grid lines every 30 degrees.
map.drawmeridians(nx.arange(0,360,30))
map.drawparallels(nx.arange(-90,90,30))
show()
```

Many more sophisticated examples including annotations, alternate projections, and detailed political and geographic boundaries are available in the examples directory of the `basemap` toolkit, which also ships with shape files for the boundaries.

Figure 4. Low-resolution satellite view of the Earth using the matplotlib `basemap` toolkit. Higher-resolution political and geographic boundaries, as well as a wealth of map projections, are available as configuration options in the toolkit.

The Matplotlib API

At its highest level, the matplotlib API has three basic classes: `FigureCanvasBase` is the canvas onto which the scene is painted, analogous to a painter's canvas; `RendererBase` is the object used to paint on the canvas, analogous to a paintbrush; and `Artist` is the object that knows how to use a renderer to paint on a canvas. `Artist` is also where most of the interesting stuff happens; basic graphics primitives such as `Line2D`, `Polygon`, and `Text` all derive from this base class. Higher-level artists such as `Tick` (for creating tick lines and labels) contain layout algorithms and lower-level primitive artists to handle the drawing of the tick line (`Line2D`), grid line (`Line2D`), and tick label (`Text`). At the highest level, the `Figure` instance itself is an `Artist` that contains one or more `Axes` instances—the `subplot` command in Figure 2 creates an `Axes` instance.

The basic drawing pipeline is fairly straightforward. For concreteness, let's look at the `Agg` back end. `Agg` is the core matplotlib raster back end that uses the antigrain C++ rendering engine to create pixel buffers with support for antialiasing and alpha transparency (see www.antigrain.com). `FigureCanvasAgg` creates the pixel buffer, and `RendererAgg` provides low-level methods for drawing onto the canvas—for example, with `draw_lines` or `draw_polygon`. The canvas is created with a reference to `Figure`, which is the top-level `Artist` that contains all other artists. This provides a rigid segregation between `Figure` and the output formats. Let's look at a complete example that uses the `Agg` canvas to make a PNG output file:

```
from matplotlib.backends.backend_agg import \
  FigureCanvasAgg as FigureCanvas
from matplotlib.figure import Figure

fig = Figure()
canvas = FigureCanvas(fig)
ax = fig.add_subplot(111)
ax.plot([1,2,3])
fig.savefig('agg_demo.png')
```

To create a different output format, such as PostScript, we need only change the first line to `from matplotlib.backends.backend_ps import FigureCanvasPS as FigureCanvas`. The method `canvas.draw()` in our code example creates a back-end-specific renderer and forwards the `draw` call to `Figure`. The `draw` method looks like this:

```
class FigureCanvasAgg(FigureCanvasBase):
  def draw(self):
    renderer = RendererAgg(self.width,
       self.height, ...)
    self.figure.draw(renderer)
```

Every `Artist` must implement the `draw` method; the call to `Figure.draw` calls `Axes.draw` for each `Axes` in the `Figure`. In turn, `Axes.draw` calls `Line2D.draw` for every line in the `Axes`, and so on, until all the matplotlib `Artists` contained in the figure are drawn. Let's look at the code for the top-level `Line2D.draw` method, which closes the circle between the high-level matplotlib `Artists` and the low-level primitive `renderer` methods:

```
class Line2D(Artist):
  def draw(self, renderer):
    x, y = self.get_transformed_xy()
    gc = renderer.new_gc()
    gc.set_foreground(self._color)
    gc.set_antialiased(self._antialiased)
    gc.set_linewidth(self._linewidth)
    gc.set_alpha(self._alpha)
    renderer.draw_lines(gc, x, y)
```

This code illustrates the encapsulation of the back-end renderer from the matplotlib `Artist`: the `Line2D` class knows

Circulation: *Computing in Science & Engineering* (ISSN 1521-9615) is published bimonthly by the AIP and the IEEE Computer Society. IEEE Headquarters, Three Park Ave., 17th Floor, New York, NY 10016-5997; IEEE Computer Society Publications Office, 10662 Los Vaqueros Circle, PO Box 3014, Los Alamitos, CA 90720-1314, phone +1 714 821 8380; IEEE Computer Society Headquarters, 1730 Massachusetts Ave. NW, Washington, DC 20036-1903; AIP Circulation and Fulfillment Department, 1NO1, 2 Huntington Quadrangle, Melville, NY 11747-4502. 2007 annual subscription rates: $45 for Computer Society members (print plus online), $76 (sister society), and $100 (individual nonmember). For AIP society members, 2007 annual subscription rates are $45 (print plus online). For more information on other subscription prices, see www.computer.org/subscribe/ or https://www.aip.org/forms/journal_catalog/order_form_fs.html. Computer Society back issues cost $20 for members, $96 for nonmembers; AIP back issues cost $22 for members.

Postmaster: Send undelivered copies and address changes to *Computing in Science & Engineering*, 445 Hoes Ln., Piscataway, NJ 08855. Periodicals postage paid at New York, NY, and at additional mailing offices. Canadian GST #125634188. Canada Post Corporation (Canadian distribution) publications mail agreement number 40013885. Return undeliverable Canadian addresses to PO Box 122, Niagara Falls, ON L2E 6S8 Canada. Printed in the USA.

that the `Renderer` instance has a `draw lines` method, but doesn't know what kind of output it renders to, be it a PostScript canvas or GTK `DrawingArea`. Likewise, the renderer and canvas don't know anything about the matplotlib coordinate system or figure hierarchy or primitive geometry types; instead, they rely on the individual artists to do the layout and transformation and make the appropriate primitive renderer calls. Although this design isn't always ideal—for example, it doesn't exploit some of the features in a given output specification—it does keep the back ends reasonably simple and dumb, allowing us to quickly add support for a new format.

Matplotlib has achieved many of its early goals; its current objectives include improving performance for real-time plotting and offering better support for an arbitrary number of scales for a given axis, user coordinate systems, and basic 3D graphics (http://matplotlib.sf.net/goals.html). The latter goal is one of the most frequent requests from matplotlib users.

The requirement for high-quality 3D graphics and visualization in Python is mostly solved via the wrapping of VTK; MayaVi2, which is part of the Enthought Toolsuite (http://code.enthought.com), provides a convenient Matlab-like interface to VTK's fairly complex functionality. Matplotlib provides some rudimentary 3D graphics, and we want to improve them so that our users can rely on basic functionality for surfaces, meshes, and 3D scatterplots without having to depend on the VTK installation. But for anything more sophisticated, we prefer to stick to our core competency—high-quality interactive scientific 2D graphs—rather than trying to replicate the already fantastic graphics provided by VTK and the MayaVi wrappers. CiSE

John D. Hunter is Senior Research Programmer and Analyst at Tradelink. His research interests include scientific visualization and event-driven trading strategies. Hunter has a PhD in neurobiology from the University of Chicago. Contact him at jdh2358@gmail.com.

WRONG AGAIN!

By Francis Sullivan

EVERYONE WHO'S EVER WRITTEN A PROGRAM MUST HAVE WONDERED AT SOME POINT HOW SO VERY MANY REALLY SNEAKY BUGS MANAGED TO CREEP INTO THE CODE. SOME ERRORS, OF COURSE, ARE MERELY TYPING MISTAKES—EVEN A MODERATE-SIZED PROGRAM CONSISTS OF SEVERAL THOUSAND characters, each of which must be correct. But others are of a much different and more subtle character and can persist for years before being exposed.

I recall working on a code for a molecular dynamics simulation. Every few time steps of integration of the governing differential equations required a recalculation of each particle's local neighborhood because the particles moved as the simulation proceeded. Because this recalculation was so frequent, not too much was expected to change between updates. I tried to take advantage of this fact by using Edsger Dijkstra's elegant smoothsort algorithm, which runs in linear time on lists that are almost sorted. After the program we'd written had been in use for several years and served as the basis of a few publications, I noticed that I'd made an error by coding an "improved" smoothsort! The bug remained hidden because my implementation was correct unless the number of particles grew quite large. When I explained this to the users, I learned that they'd quietly replaced my version of smoothsort with a correct quicksort years earlier because they didn't understand how smoothsort was supposed to work.

Ever since this strange episode, I've wondered how and why such things can happen. Of course, if the pseudocode is well structured, it's possible to write a correct program without having *any* understanding of the algorithm. It reminds me of the song "Lobachevsky" by Tom Lehrer:

> I am never forget the day I am given first original paper to write. It was on Analytic and Algebraic Topology of Locally Euclidean Metrization of Infinitely Differentiable Riemannian Manifold. Bozhe moi! This I know from nothing. But I think of great Lobachevsky and I get idea—haha!

Lobachevsky's idea, of course, was to plagiarize. However, instead of merely plagiarizing from smoothsort's pseudocode, I decided to do something more dangerous: I decided to think. The error was typical of what can happen when you "almost" understand an algorithm and then make a small change to express things differently. In other words, the mistake was in my interpretation of what I thought was smoothsort logic.

Philosophers have written extensively about logic errors, focusing on the question of how they could occur if everything is done according to logic. These authors usually mean "true" errors rather than something like a typing mistake, but it's difficult to tell if they assume that, for every proposition, either it or its negation could be proved. In the case of computer programs, Dijkstra and his colleagues strongly advocated a technology of formal verification, which can be useful in small cases because it provides clarity, discipline, and structure. If I'd done a formal analysis of my modification, I might have found the flaw. But then, of course, I'd have to somehow check the formal verification, perhaps by doing a higher-level even more formal verification of the first formal verification, and so on. It begins to feel like the halting problem.

So what is to be done? I suspect a completely reliable, formal, and rigorous self-checking method for expressing algorithms is impossible. This doesn't mean we shouldn't try for clarity of expression, but in the end, the acid test is to run the program on real data—lots and lots of real data. During the Punic wars, the Roman senator Cato said *Carthago delenda est*! (Carthage must be destroyed!) With apologies to all readers expert in Latin, I'm adopting as a motto, *data vera utenda sunt*! (Real data must be used!)

Francis Sullivan is the director of the IDA Center for Computing Sciences in Bowie, Maryland. Contact him at fran@super.org.