Python Unleashed on Systems Biology

Researchers at Cornell University have built an open source software system to model biomolecular reaction networks. SloppyCell is written in Python and uses third-party libraries extensively, but it also does some fun things with on-the-fly code generation and parallel programming.

A central component of the emerging field of systems biology is the modeling and simulation of complex biomolecular networks, which describe the dynamics of regulatory, signaling, metabolic, and developmental processes in living organisms. (Figure 1 shows a small but representative example of such a network, describing signaling by G protein-coupled receptors.[1] Other networks under investigation by our group appear online at www.lassp.cornell.edu/sethna/GeneDynamics/.) Naturally, tools for inferring networks from experimental data, simulating network behavior, estimating model parameters, and quantifying model uncertainties are all necessary to this endeavor.

Our research into complex biomolecular networks has revealed an additional intriguing property—namely, their *sloppiness*. These networks are vastly more sensitive to changes along some directions in parameter space than along others.[2–5] Although many groups have built tools for simulating biomolecular networks (www.sbml.org), none support the types of analyses that we need to unravel this sloppiness phenomenon. Therefore, we've implemented our own software system—called SloppyCell—to support our research (http://sloppycell.sourceforge.net).

Much of systems biology is concerned with understanding the dynamics of complex biological networks and in predicting how experimental interventions (such as gene knockouts or drug therapies) can change that behavior. SloppyCell augments standard dynamical modeling by focusing on inference of model parameters from data and quantification of the uncertainties of model predictions in the face of model sloppiness, to ascertain whether such predictions are indeed testable.

The Python Connection

SloppyCell is an open source software system written in Python to provide support for model construction, simulation, fitting, and validation. One important role of Python is to glue together many diverse modules that provide specific functionality. We use NumPy (www.scipy.org/NumPy) and SciPy (www.scipy.org) for numerics—particularly, for integrating differential equations, optimizing parameters by least squares fits to data, and analyzing the Hessian matrix about a best-fit set of parameters. We use matplotlib for plotting (http://matplotlib.sourceforge.net). A Python interface to the libSBML library (http://sbml.org/software/libs

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

Christopher R. Myers, Ryan N. Gutenkunst, and James P. Sethna
Cornell University

bml/) lets us read and write models in a standardized, XML-based file format, the Systems Biology Markup Language (SBML),[6] and we use the `pypar` wrapper (http://datamining.anu.edu.au/~ole/pypar/) to the message-passing interface (MPI) library to coordinate parallel programs on distributed memory clusters. We can generate descriptions of reaction networks in the `dot` graph specification language for visualization via Graphviz (www.graphviz.org). Finally, we use the `smtplib` module to have simulation runs send email with information on their status (for those dedicated researchers who can't bear to be apart from their work for too long).

Although Python serves admirably as the glue, we focus here on a few of its powerful features—the ones that let us construct highly dynamic and flexible simulation tools.

Code Synthesis and Symbolic Manipulation

Researchers typically treat the dynamics of reaction networks as either continuous and deterministic (modeling the time evolution of molecular concentrations) or as discrete and stochastic (by simulating many individual chemical reactions via Monte Carlo). In the former case, we construct systems of ordinary differential equations (ODEs) from the underlying network topology and the kinetic forms of the associated chemical reactions. In practice, these ODEs are often derived by hand, but they need not be: all the information required for their synthesis is embedded in a suitably defined network, but the structure of any particular model is known only at runtime once we create and specify an instance of a `Network` class.

With Python, we use symbolic expressions (encoded as strings) to specify the kinetics of different reaction types and then loop over all the reactions defined in a given network to construct a symbolic expression for the ODEs that describe the time evolution of all chemical species. (We depict the reactions as arrows in Figure 1; we can query each reaction to identify those chemicals involved in that reaction [represented as shapes], as well as an expression for the instantaneous rate of the reaction based on model parameters and the relevant chemical concentrations.) This symbolic expression is formatted in such a way that we can define a new method, `get_ddv_dt(y, t)`, which is dynamically attached to an instance of the `Network` class using the Python `exec` statement. (The term "`dv`" in the method name is shorthand for "dynamical variables"—that is, those chemical species whose time evolution we're solving for.) We then use this dynamically generated method in conjunction with ODE integrators (such as `scipy.integrate.odeint`, which is a wrapper around the venerable LSODA integrator,[7,8] or with the variant LSODAR,[9] which we've wrapped up in SloppyCell to integrate ODEs with defined events). We refer to this process of generating the set of ODEs for a model directly from the network topology as "compiling" the network.

This sort of technique helps us do more than just synthesize ODEs for the model itself. Similar techniques let us construct *sensitivity equations* for a given model, so that we can understand how model trajectories vary with model parameters. To accomplish this, we developed a small package that supports the differentiation of symbolic Python math expressions with respect to specified variables,

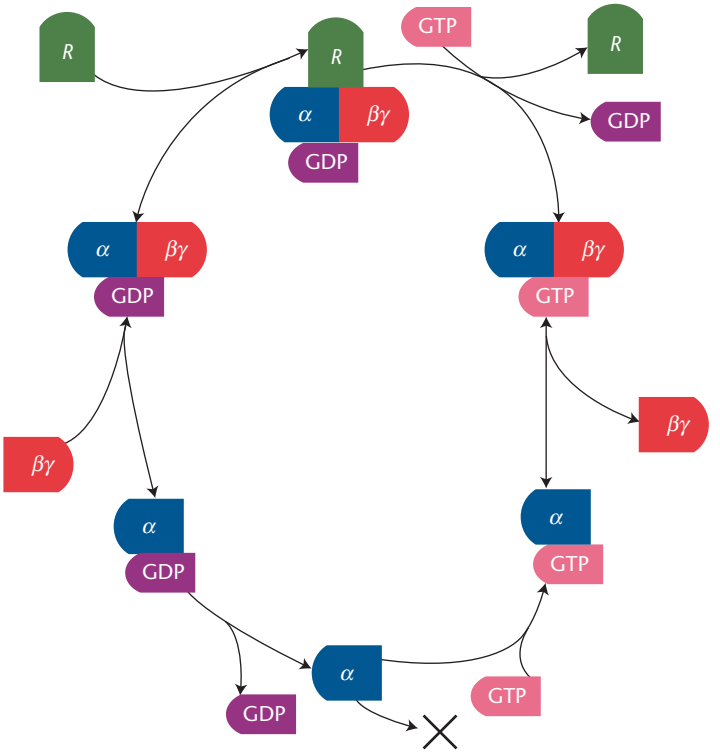

Figure 1. Model for receptor-driven activation of heterotrimeric G proteins.[1] The α signaling protein is inactive when bound to guanosine diphosphate (GDP) and active when bound to guanosine triphosphate (GTP). After forming a complex with a $\beta\gamma$ protein, binding to the active receptor *R* allows the α protein to release its GDP and bind GTP. The complex then dissociates into *R*, $\beta\gamma$, and activated α. The activated α protein goes on to signal downstream targets, whereas the $\beta\gamma$ protein is free to bring new inactive α to the receptor.

by converting mathematical expressions to abstract syntax trees (ASTs) via the Python `compiler` module. This lets us generate another new method, `get_d2dv_dovdt(y, t)`, which describes the derivatives of the dynamical variables with respect to both time and optimizable variables (model parameters whose values we're interested in fitting to data). By computing parametric derivatives analytically rather than via finite differences, we can better navigate the ill-conditioned terrain of the sloppy models of interest to us.

The ASTs we use to represent the detailed mathematical form of biological networks have other benefits as well. We also use them to generate LaTeX representations of the relevant systems of equations—in practice, this not only saves us from error-prone typing, but it's also useful for debugging a particular model's implementation.

Colleagues of ours who are developing PyDSTool (http://pydstool.sourceforge.net)—a Python-based package for simulating and analyzing dynamical systems—have taken this type of approach to code synthesis for differential equations a step further. The approach we described earlier involves generating Python-encoded right-hand sides to differential equations, which we use in conjunction with compiled and wrapped integrators. For additional performance, PyDSTool supports the generation of C-encoded right-hand sides, which it can then use to dynamically compile and link with various integrators using the Python `distutils` module.

Parallel Programming in SloppyCell

Because of the sloppy structure of complex biomolecular networks, it's important not to just simulate a model for one set of parameters but to do so over large families of parameter sets consistent with available experimental data. Accordingly, we use Monte Carlo sampling to simulate a model with many different parameter sets and thus estimate the model uncertainties (error bars) associated with predictions. Parallel computing on distributed memory clusters efficiently enables these sorts of extensive parameter explorations. Moreover, several different Python packages provide interfaces to MPI libraries, and we've found `pypar` to be especially useful in this regard.

Whereas message passing on distributed memory machines is inherently somewhat cumbersome and low level, `pypar` raises the bar by exploiting built-in Python support for the *pickling* of complex objects. Message passing in a low-level programming language such as Fortran or C typically requires constructing appropriately sized memory buffers into which we must pack complex data structures, but `pypar` uses Python's ability to serialize (or pickle) an arbitrary object into a Python string, which can then be passed from one processor to another and unpickled on the other side. With this, we can pass lists of parameters, model trajectories returned by integrators, and so on; we can also send Python exception objects raised by worker nodes back to the master node for further processing. (These can arise, for example, if the ODE integrator fails to converge for a particular set of parameters.)

Additionally, Python's built-in `eval` statement makes it easy to create a very flexible worker that can execute arbitrary expressions passed as strings by the master (requiring only that the inputs and return value are pickle-able). The following code snippet demonstrates a basic error-tolerant master–worker parallel computing environment for arbitrarily complex functions and arguments defined in some hypothetical module named `our_science`:

```
import pypar
# our_science contains the functions
# we want to execute
import our_science

if pypar.rank() != 0:
    # The workers execute this loop.
    # (The master has rank == 0.)
    while True:
        # Wait for a message from
        # the master.
        msg = pypar.receive(source=0)

        # Exit python if sent a
        # SystemExit exception
        if isinstance(msg, SystemExit):
            sys.exit()

        # Evaluate the message and
        # send the result back to
        # the master.
        # If an exception was raised,
        # send that instead.
        command, msg_locals = msg
        locals().update(msg_locals)
        try:
            result = eval(command)
        except X:
            result = X
        pypar.send(result, 0)

# The code below is only run by
```

```
# the master.

# Evaluate our_science.foo(bar) on each
# worker, getting values for bar from
# our_science.todo.

command = 'our_science.foo(bar)'
for worker in range(1, pypar.size()):
    args = {'bar': \
            our_science.todo[worker]}
    pypar.send((command, args), worker)

# Collect results from all workers.
results = [pypar.receive(worker) \
           for worker in \
           range(1, pypar.size())]

# Check if any of the workers failed.
# If so, raise the resulting Exception.
for r in results:
    if isinstance(r, Exception):
        raise r

# Shut down all the workers.
for worker in range(1, pypar.size()):
    pypar.send(SystemExit(), worker)
```

We've very briefly described a few of the fun and flexible features that Python provides to support the construction of expressive computational problem-solving environments, such as those needed to tackle complex problems in systems biology. Although any programming language can be coaxed into doing what's desired with sufficient hard work, Python encourages researchers to ask questions that they might not have even considered in less expressive environments. CiSE

Acknowledgments

We thank Fergal Casey, Joshua Waterfall, Robert Kuczenski, and Jordan Atlas for their help in developing and testing SloppyCell, and we acknowledge the insights of Kevin Brown and Colin Hill in developing predecessor codes, which have helped motivate our work. Development of SloppyCell has been supported by NSF grant DMR-0218475, USDA-ARS project 1907-21000-017-05, and an NIH Molecular Biophysics Training grant to Gutenkunst (no. T32-GM-08267).

References

1. V.Y. Arshavsky, T.D. Lamb, and E.N. Pugh, "G Proteins and Phototransduction," *Ann. Rev. Physiology*, vol. 64, no. 1, 2002, pp. 153–187.
2. K.S. Brown and J.P. Sethna, "Statistical Mechanical Approaches to Models with Many Poorly Known Parameters," *Physical Rev. E*, vol. 68, no. 2, 2003, p. 021904; http://link.aps.org/abstract/PRE/v68/e021904.
3. K.S. Brown et al., "The Statistical Mechanics of Complex Signaling Networks: Nerve Growth Factor Signaling," *Physical Biology*, vol. 1, no. 3, 2004, pp. 184–195; http://stacks.iop.org/1478-3975/1/184.
4. J.J. Waterfall et al., "Sloppy-Model Universality Class and the Vandermonde Matrix," *Physical Rev. Letters*, vol. 97, no. 15, 2006, pp. 150601–150604; http://link.aps.org/abstract/PRL/v97/e150601.
5. R.N. Gutenkunst et al., "Universally Sloppy Parameter Sensitivities in Systems Biology," 2007; http://arxiv.org/q-bio.QM/0701039.
6. M. Hucka et al., "The Systems Biology Markup Language (SBML): A Medium for Representation and Exchange of Biochemical Network Models," *Bioinformatics*, vol. 19, no. 4, 2003, pp. 524–531; http://bioinformatics.oxfordjournals.org/cgi/content/abstract/19/4/524.
7. A.C. Hindmarsh, "Lsode and Lsodi, Two New Initial Value Ordinary Differential Equation Solvers," *ACM-SIGNUM Newsletter*, vol. 15, no. 4, 1980, pp. 10–11.
8. L.R. Petzold, "Automatic Selection of Methods for Solving Stiff and Nonstiff Systems of Ordinary Differential Equations," *SIAM J. of Scientific and Statistical Computing*, vol. 4, no. 1, 1983, pp. 137–148.
9. A.C. Hindmarsh, "Odepack, A Systematized Collection of ODE Solvers," *Scientific Computing*, North-Holland, 1983, pp. 55–64.

Christopher R. Myers *is a senior research associate and associate director in the Cornell Theory Center at Cornell University. His research interests lie at the intersection of physics, biology, and computer science, with particular emphases on biological information processing, robustness and evolvability of natural and artificial networks, and the design and development of software systems to probe complex spatiotemporal phenomena. Myers has a PhD in physics from Cornell. Contact him at myers@tc.cornell.edu; www.tc.cornell.edu/~myers.*

Ryan N. Gutenkunst *is a graduate student in the Laboratory of Atomic and Solid State Physics at Cornell University. He uses SloppyCell to study universal properties of complex biological networks, and is interested in how these properties affect both practical model development and the dynamics of evolution. Gutenkunst has a BS in physics from the California Institute of Technology. Contact him rng7@cornell.edu; http://pages.physics.cornell.edu/~rgutenkunst/.*

James P. Sethna *is a professor of physics at Cornell University, and is a member of the Laboratory of Atomic and Solid State Physics. His recent research interests include common, universal features found in nonlinear optimization problems with many parameters, such as sloppy models arising in the study of biological signal transduction. Sethna has a PhD in physics from Princeton University. Contact him at sethna@lassp.cornell.edu; www.lassp.cornell.edu/sethna.*