

Computational Physics Education with Python

Educators at an institution in Germany have started using Python to teach computational physics. The author describes how graphical visualizations also play an important role, which he illustrates here with a few simple examples.

In recent years, various universities have supplemented their usual courses on theoretical and experimental physics with courses on computational physics, and the physics department at the Technische Universität Dresden is no exception. In this article, I give a short overview of our experience in establishing such a course with Python as a programming language. The course's main goal is to enable students to solve problems in physics with the help of numerical computations. In particular, we want them to use graphical and interactive exploration to gain a more profound understanding of the underlying physical principles and the problem at hand.

Starting the Course

After setting up a list of possible topics for the course, one of the first decisions we had to make was our choice of programming language. Several points drove our decision to pick Python:

- It's freely available for Linux, Macintosh, and Windows systems.

- For a full programming language, it's easy to learn (even for beginners).
- Its readable and compact code allows for a shorter development time, which gives students more time to concentrate on the physics problem itself.
- It can be used interactively, including for plotting.
- Object-oriented programming (OOP) is possible, but it isn't required.

We started the computational physics course in 2002 and have run it every summer term since then, with student numbers increasing from roughly 20 to 70 students total—up to 50 percent of each year's physics students. Two-hour tutorials and exercise sheets accompany the two-hour weekly lectures that cover both the physical and numerical aspects of elementary numerical methods (differentiation, integration, zero finding), differential equations, random numbers, stochastic processes, Fourier transformation, nonlinear dynamics, and quantum mechanics. Students hand in their solutions by email, and instructors print out, test, correct, grade, and return their programs in the next tutorial session to provide individual feedback. In addition, we post extensively commented sample solutions on the course's Web page.

Student Experience

Over the years since the course's inception, we've found our students' knowledge of programming

```

from pylab import *                                # plotting routines
from scipy.integrate import odeint                 # routine for ODE integration

def derivative(y, t):
    """Right hand side of the differential equation.

    Here y = [phi, v].
    """
    return array([y[1], sin(y[0]) + 0.25* cos(t)]) # (\dot{\phi}, \dot{v})

def compute_trajectory(y0):
    """Integrate the ODE for the initial point y0 = [phi_0, v_0]"""
    t = arange(0.0, 100.0, 0.1)                   # array of times
    y_t = odeint(derivative, y0, t)                # integration of the equation
    return y_t[:, 0], y_t[:, 1]                   # return arrays for phi and v

# compute and plot for two different initial conditions:
phi_a, v_a = compute_trajectory([1.0, 0.9])
phi_b, v_b = compute_trajectory([0.9, 0.9])
plot(phi_a, v_a)
plot(phi_b, v_b, "r--")
xlabel(r"$\varphi$")
ylabel(r"$v$")
show()

```

Figure 1. Python program to compute and visualize a driven pendulum's time evolution.

languages to be rather diverse, ranging from no experience at all to detailed expertise in C++. With a few exceptions, we rarely found previous knowledge of Python. To provide the necessary basics, we give a short introduction to the language, which makes extensive use of Python's interactive capabilities (with IPython). This introduction covers the use of variables, simple arithmetic, loops, conditional execution, small programs, and subroutines. Of particular importance for numerical computations is the use of arrays provided by NumPy, which lets the students write efficient code without explicit loops. Again, this makes the code compact but also enhances its readability and programming speed. Students can use SciPy for additional numerical routines, such as solving ordinary differential equations or computing special functions.

Finally, we give a short introduction to plotting via matplotlib. After starting IPython with support for interactive plotting via `ipython -pylab`, students can do

```

# Array of x values
x = linspace(0.0, 2.0*pi, 100)
# plot graph of sin(x) vs. x:

```

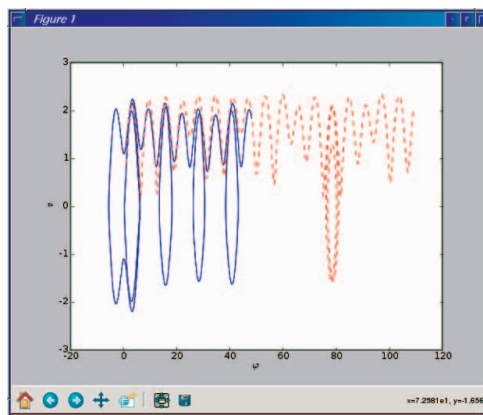


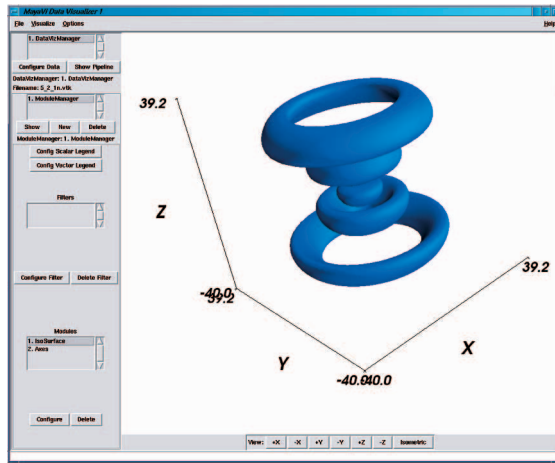
Figure 2. Pendulum example. Using matplotlib, we can visualize a driven pendulum's dynamics (described in Equation 1) for two different initial conditions (bold and dashed curves).

```

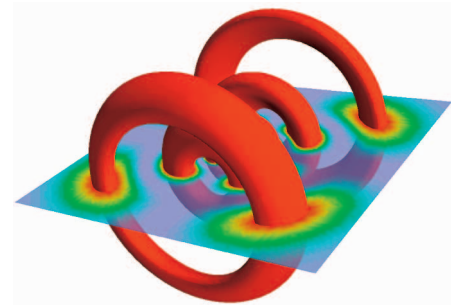
plot(x, sin(x))
# add another graph
plot(x, cos(2*x))

```

and then zoom into the resulting plot via the mouse.



(a)



(b)

Figure 3. Isosurface example. The (a) screenshot of a MayaVi visualization shows a surface of constant value of $|\Psi_{n,l,m}(x, y, z)|^2$ for the hydrogen atom, where $n, l, m = 5, 2, 1$. With minor variations, we get (b) the same type of plot, but with a scalar cut-plane and a different lookup table.

Two Examples

A typical example we use in the classroom is a driven pendulum, described by

$$\ddot{\phi} = \sin \phi + (1/4)\cos t,$$

which we can rewrite as a coupled differential equation:

$$\begin{aligned} \dot{\phi} &= v \\ \dot{v} &= \sin \phi + (1/4)\cos t \end{aligned} \quad (1)$$

for which the simple program in Figure 1 computes the time evolution.

Figure 2 shows the resulting plot of v versus ϕ as a screenshot; in this system, we see how a moderate change in the initial condition leads to quite different behavior after a short time period.

A more advanced example is the visualization of quantum probability densities for the hydrogen atom's wave functions, which, in spherical coordinates, reads

$$\begin{aligned} \Psi_{n,l,m}(r, \nu, \phi) &= \\ & Y_{lm}(\nu, \phi) \\ & \cdot \sqrt{\frac{(n-l-1)!(2/n)^3}{2n[(n+l)!]}} \\ & \cdot (2r/n)^l e^{-r/n} L_{n-l-1}^{2l+1}(2r/n), \end{aligned} \quad (2)$$

where n, l , and m are the quantum numbers that characterize the wave function, Y is the spherical harmonics (numerically determined from `scipy.special.spharm`), and L is the associated La-

guerre polynomial (determined from `scipy.special.assoclaguerre`).

Because $\psi(x, y, z)$ is a scalar function defined on \mathbb{R}^3 , we can either use a density plot or plot equi-energy surfaces. Instead of writing the corresponding (nontrivial) visualization routines from scratch, we use the extremely powerful Visualization Toolkit (VTK; www.vtk.org), whose routines are also accessible from Python. The first step is to generate a data file suitable for VTK by choosing a rasterization in Cartesian coordinates (x, y, z) on $[-40, 40]^3$ with 100 points in each direction,

```
# vtk DataFile Version 2.0
data.vtk hydrogen wave function data
ASCII

DATASET STRUCTURED_POINTS
DIMENSIONS 100 100 100
ORIGIN -40.0 -40.0 -40.0
SPACING 0.8 0.8 0.8
POINT_DATA 1000000
SCALARS scalars float
LOOKUPTABLE default
... 1000000 real numbers corresponding
to |\psi_{5,2,1}(x,y,z)|^2...
```

We can visualize this data set with MayaVi by starting at the shell prompt

```
mayavi -d data.vtk -m IsoSurface -m Axes
```

This command line reads the data file `data.vtk`,

loads the isosurface module, and adds axes to the plot. By modifying the isosurface's value, we get the plot in Figure 3a for $n, l, m = 5, 2, 1$. Adding a scalar-cut-plane, a different color lookup table, and varying the view takes us to Figure 3b. This short example shows that with only a moderate level of effort, highly instructive and appealing data visualizations are possible.

At the end of the course, students give a short presentation on a small project of their choice. In this task, the students' creativity isn't limited by Python because it helps them create highly illustrative dynamical visualizations, some of them even in 3D (with VPython; www.vpython.org).

Our initial attempt at using Python for teaching computational physics has proven to be highly successful. In fact, several students have continued to use

Python for other tasks, such as data analysis in experimental physics courses or during a diploma thesis outside our group. The plan is to fully integrate the computational physics course into the compulsory curriculum.

Ci
SE

Acknowledgments

I thank Roland Ketzmerick, with whom the concept of this computational physics course was developed jointly. I also thank the people involved in setting up and running the course. Finally, a big thanks to all the authors and contributors to Python and the software mentioned here.

Arnd Bäcker is a scientific researcher at the Institut für Theoretische Physik, Technische Universität Dresden. His research interests include nonlinear dynamics, quantum chaos, and mesoscopic systems. Bäcker has a PhD in theoretical physics from the Universität Ulm. Contact him at baecker@physik.tu-dresden.de.

AAPT Topical Conference Computational Physics for Upper Level Courses

27–28 July 2007

DAVIDSON

Davidson College
Davidson, NC



For further information go to www.opensourcephysics.org/CPC

The purpose of this conference is to identify problems where computation helps students understand key physics concepts. Participants are university and college faculty interested in integrating computation at their home institutions. Some participants already teach or have taught computational physics to undergraduates and some are looking for ways to integrate computational physics into their existing physics curriculum.

Participants will contribute and discuss algorithms and curricular material for teaching core subjects such as mechanics, electricity and magnetism, quantum mechanics, and statistical and thermal physics. Participants will prepare and edit their material for posting on an AAPT website such as ComPADRE. Visiting experts will give talks on how computational physics may be used to present key concepts and current research to undergraduates.

Participants are invited to prepare a poster describing how they incorporate computational physics into their teaching, what projects they have assigned to students at different levels, and how computation has enhanced their curriculum. Posters will remain up throughout the conference.

Invited speakers include Amy Bug (Swarthmore College), Norman Chonacky (*CiSE* editor), Francisco Esquembre (University of Murcia, Spain), Robert Swendsen (Carnegie-Mellon University), Steven Gottlieb (Indiana University), Rubin Landau (Oregon State University), Julien C. Sprott (University of Wisconsin), Angela Shiflet (Wofford College), and Eric Warren (Evernham Motorsports).

The organizing committee consists of Wolfgang Christian, Jan Tobochnik, Rubin Landau, and Robert Hilborn.

**Partial funding for this conference is being provided by Computing in Science & Engineering, the Shodor Foundation, the TeraGrid project, and NSF grant DUE-442581.*

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 sup-

port 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 `py-par` 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

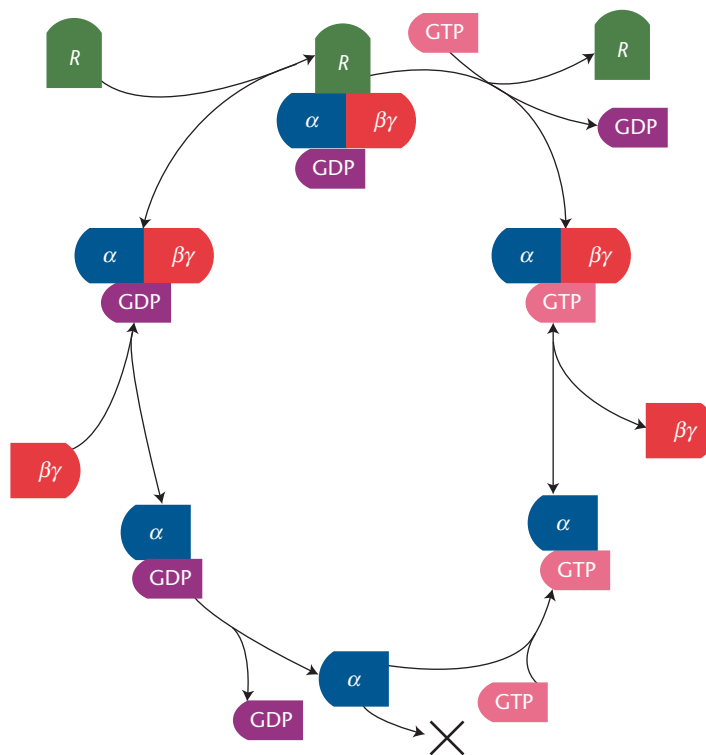


Figure 1. Model for receptor-driven activation of heterotrimeric G proteins.¹ The α signaling protein is inactive when bound to guanosine diphosphate (GDP) and active when bound to guanosine triphosphate (GTP). After forming a complex with a $\beta\gamma$ protein, binding to the active receptor R allows 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.

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,

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. 

Acknowledgments

We thank Fergal Casey, Joshua Waterfall, Robert Kuczynski, 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.

Reaching for the Stars with Python

The author describes how Python has helped scientists calibrate and analyze data from the Hubble Space Telescope, first as a means of scripting legacy applications, and, more recently, as a way of developing new applications in Python itself.

The Hubble Space Telescope (HST), a 2.4-meter optical, ultraviolet, and infrared telescope, has orbited Earth for more than 16 years. Ground-based telescopes suffer the effects of the atmosphere. In particular, the atmosphere blurs the image, blocks ultraviolet and infrared photons, and scatters light—whether from the moon or your local streetlight—resulting in brighter backgrounds and greater difficulty in observing faint sources. Being above the atmosphere means that the HST makes sharper images of fainter sources at wavelengths not visible from the ground. As a result, the HST can obtain exquisite observations and make many important discoveries that are otherwise unobtainable.

The Space Telescope Science Institute (STScI) is responsible for the HST's scientific operation, including scheduling observations and processing and disseminating the resulting data to both the public and astronomers. Our group at STScI develops the software used to calibrate and analyze the data. This article describes our experiences with the Python language and how we've incorporated it into our work.

Using Python

Our initial use of Python was in developing an alternate scripting environment for a popular astronomical analysis system called IRAF (for Imaging Reduction and Analysis Facility), which has its own custom compiled language for its applications and its own scripting language and command-line environment. Although IRAF had great initial success and longevity (the US National Optical Astronomy Observatory developed it in the early 1980s), it has proved to be an increasingly constrained development environment as time goes on. To remedy this situation, we developed a way to script IRAF applications using Python so that we could run IRAF tasks more robustly and combine them with the wide variety of libraries and tools available in Python. This new scripting environment—called PyRAF—proved easier to develop in Python than we expected and let us add more powerful capabilities than we had originally hoped (see www.python.org/workshops/2000-01/proceedings/papers/white/pyrafpaper.pdf). Python had fewer limits on what it could accomplish—outside of efficiency concerns—and offered broad library support, powerful yet easy-to-read language features (particularly its numerous language hooks for extending features to new objects), and the ability to call out to C when necessary (for which the need was rare). More important, Python's interactive nature made it much easier to prototype, test, and debug applications.

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

PERRY GREENFIELD
Space Telescope Science Institute

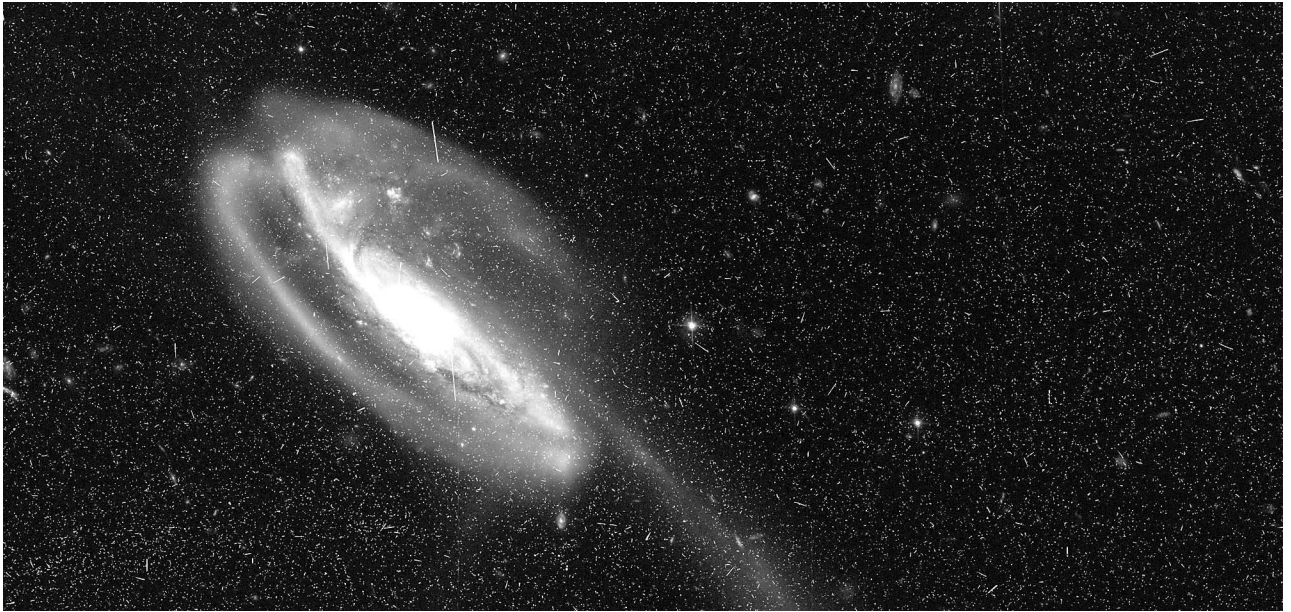


Figure 1. One raw exposure of the Tadpole galaxy (UGC10214). Each raw exposure consists of two $4,096 \times 2,048$ pixel charge-coupled device images (one of which is shown). This 840-second exposure was taken with a wide-band filter centered at 606 nm (orange-red). Most of the tens of thousands of star-like images, specks, and streaks aren't stars—they're the result of cosmic rays, and thus don't represent detected photons. This figure (and Figure 2) uses a nonlinear intensity transfer function to emphasize the fainter features in the image.

Because of our success with PyRAF, we decided to develop as many of our applications in Python as possible. This seemed realistic because many astronomers have successfully developed useful tools and applications with the commercial image-processing package IDL (Interactive Data Language), an interactive array manipulation language that lets users easily handle images and spectra in mathematical expressions via numerical and visualization tools. Because it looked as if Python could do much of what IDL could, we began building tools to enable application development in Python. We started with an enhanced array package (Numarray; www.python.org/pycon/papers/numarray.html). The successor to Numarray, NumPy (see Travis Oliphant's article on p. 10 of this issue), incorporates all of its features; we're in the process of converting all our programs to use it. We've also developed other libraries such as PyFITS (www.stsci.edu/resources/software_hardware/pyfits/Users_Manual_1.pdf), a module that can read and write the standard astronomical Flexible Image Transport System (FITS) data format and also contributed to matplotlib (see the Scientific Programming department on p. 90).

This direction has proved to be very productive. We're now able to develop and modify applications much faster in Python than we could have with

IRAF. When it's necessary to develop algorithms that require a compiled language, it's straightforward to write routines with C (via "vanilla" Python extensions) to handle the small percentage of cases that require it, and still use the routines from Python. Today, we develop nearly all our new software in Python.

An Example Application

A heavily used application, both for calibration pipelines and interactive analysis, is Multidrizzle.¹ This application takes several spatially offset (but overlapping) exposures of the sky and combines them into one image. This might seem like a simple operation, but it's actually quite complex—because of telescope and camera distortion, we can't simply offset image arrays spatially and add them together. Rather, we must resample the pixels and apply distortion corrections before combining. The task is further complicated by the presence of cosmic rays, which are high-energy particles that cause spurious star-like objects in telescopic images. Because they don't appear in multiple images, we want to compare overlapping regions to see if these features appear in only one of them and reject them if they don't appear in all of them.

The issue of determining proper registration of these different images is complex and involves au-

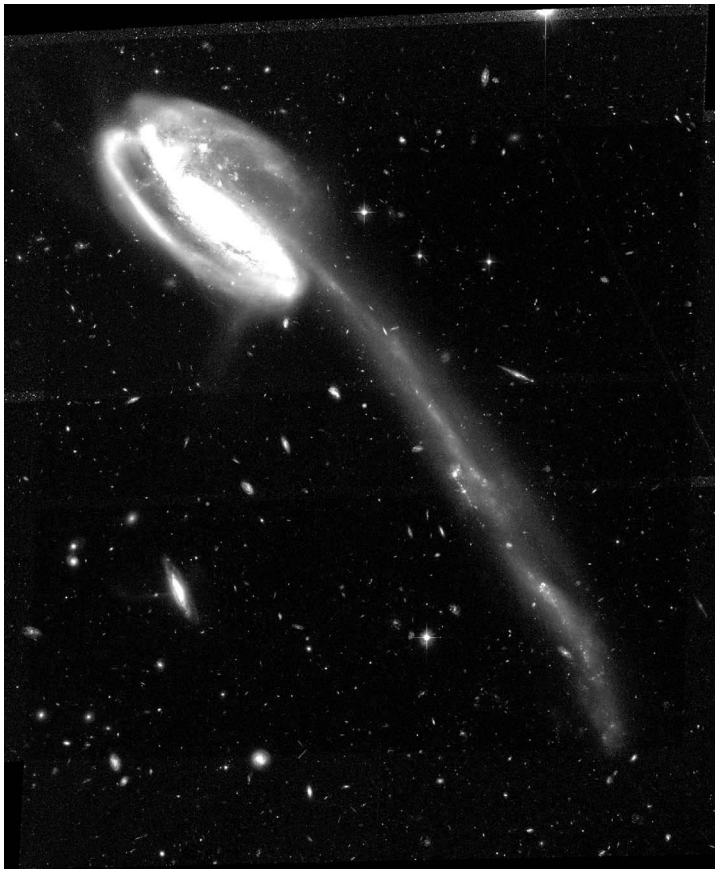


Figure 2. The final image produced by combining six individual exposures with Multidrizzle. Nearly all the cosmic rays are removed here. The odd outline at the edges results from the different overlaps used to cover the observed field. Nearly all the remaining small objects visible are galaxies. Higher noise is apparent in the regions covered with fewer exposures. The resulting image's size is larger because of the offsets—it's now $4,582 \times 4,879$ pixels (the displayed image is slightly cropped on the left and right sides).

tomatic object identification and matching. Naturally, the amount of data to which this technique is applied is large—each exposure is $4,000 \times 4,000$ pixels, and we might have to combine many of them. Figure 1 shows an image (one of several) as it appears before using Multidrizzle to process it, and Figure 2 shows the result after processing.

Multidrizzle calls a Fortran routine for the pixel resampling (and a few C-extension routines for other operations), but most of the code is Python. An associated Python program called `tweakshifts` performs automatic image registration by using other packages to identify objects. Multidrizzle can automatically process thousands of data sets in the HST data-processing pipelines, but it's also distributed to the astronomical community for interactive use. Moreover, PyRAF lets us run Multidrizzle as though

it were an IRAF task even though it isn't. In other words, users can run it from a familiar command environment with an interface similar to IRAF tasks.

We're impressed with Python because it works well as a glue for integrating existing software tools, from libraries to stand-alone executables to entire legacy systems. Python is much more accessible to scientists than languages such as C, C++, Java, or Fortran—indeed, a staff astronomer, not a professional software developer, wrote the initial Multidrizzle prototype.

We've also used Python to process data from an experiment to measure the structural stability of the mirror-support structure of the next large space telescope in development. The James Webb Space Telescope (JWST) will have a 6.5-meter primary mirror made up of 18 hexagonal mirror segments, and has a planned launch date in 2013. JWST's mirror-support structure must remain stable to within tens of nanometers for modest changes in its temperature (which is very cold, just tens of degrees above absolute zero). We've developed Python applications to acquire and process 10 Tbytes of data over the experiment's run, and we expect to write most of our JWST data-processing and reduction applications in Python as well.

But Python's use at STScI isn't limited to data processing and analysis: we're using it in many other areas, including telescope scheduling and planning. It has even found wider use in astronomy outside of STScI and appears to be the most commonly adopted scripting language for large projects in several scientific domains. For example, other major legacy astronomical data analysis software systems have added Python interfaces: PyMidas for MIDAS (Munich Image Data Analysis System), Parseltongue for AIPS (Astronomical Image Processing System), and PySL for CIAO (Chandra Interactive Analysis of Observations); the next major radio astronomy project, ALMA (Atacama Large Millimeter Array), uses Python as its scripting language.

Reference

1. A.M. Koekemoer et al., "MultiDrizzle: An Integrated PyRaf Script for Registering, Cleaning and Combining Images," *2002 HST Calibration Workshop*, Space Telescope Science Inst., 2002; www.stsci.edu/largefiles/hst/HST_overview/documents/calworkshop/workshop2002/CW2002_Papers/koekemoer_multidrizzle.pdf.

Perry Greenfield is a science analysis tools project lead at the Space Telescope Science Institute. He has a PhD in physics from MIT. Contact him at perry@stsci.edu.

A Python Module for Modeling and Control Design of Flexible Robots

This article discusses the creation of a Python module for object-oriented modeling and control design of flexible robots using the transfer matrix method (TMM). The authors overcame several theoretical hurdles to apply the TMM to practical flexible robots and have experimentally validated the Python module's modeling capabilities.

Flexible robots offer the benefits of being lighter, faster, and cheaper to actuate than their rigid counterparts. However, robots with flexible links or joints also pose a considerable challenge because they might be composed of discrete and distributed parameter elements, many links, complicated actuators, and multiple feedback loops. The example system analyzed in this article poses two additional challenges: it's hydraulically actuated with two feedback loops in which the sensors and actuators aren't precisely collocated.

Existing modeling approaches for flexible structures are inadequate for designing controllers for flexible robots. The transfer matrix method (TMM) might be an excellent approach if some theoretical hurdles are overcome and if some new software package makes the approach more accessible and user friendly.¹⁻⁴

This article discusses how we expanded the TMM's capabilities and developed a Python software module to make the TMM an excellent tool

for the modeling and control design of practical flexible robots.⁵

System Description

Figure 1 shows a picture of the flexible robot we used in the experimental part of our work. The robot is called SAMII, which stands for small, articulated manipulator II. SAMII is a hydraulically actuated robot with rigid links mounted on the end of a cantilevered beam. The cantilevered beam represents a larger robot that would give SAMII a larger workspace and move it into the general position desired. Once SAMII is in position, the large robot's joints might be locked. Thus, the large robot must have long links to give SAMII a large workspace, even though this length inherently implies flexibility and vibration problems. We thus need modeling approaches and control schemes to deal with this flexibility and suppress or avoid vibrations.

Problem Statement

Figure 2 shows a block diagram of the control scheme. The goal is to develop transfer functions for G_θ and G_a , where G_θ specifies how the system will respond to commands to move to a desired position θ_d and is referred to as the motion control portion of the control scheme, and G_a is the vibration suppression controller. We want to find the optimal G_θ and G_a so that the robot moves quickly while also suppressing vibration.

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

RYAN W. KRAUSS

Southern Illinois University Edwardsville

WAYNE J. BOOK

Georgia Institute of Technology

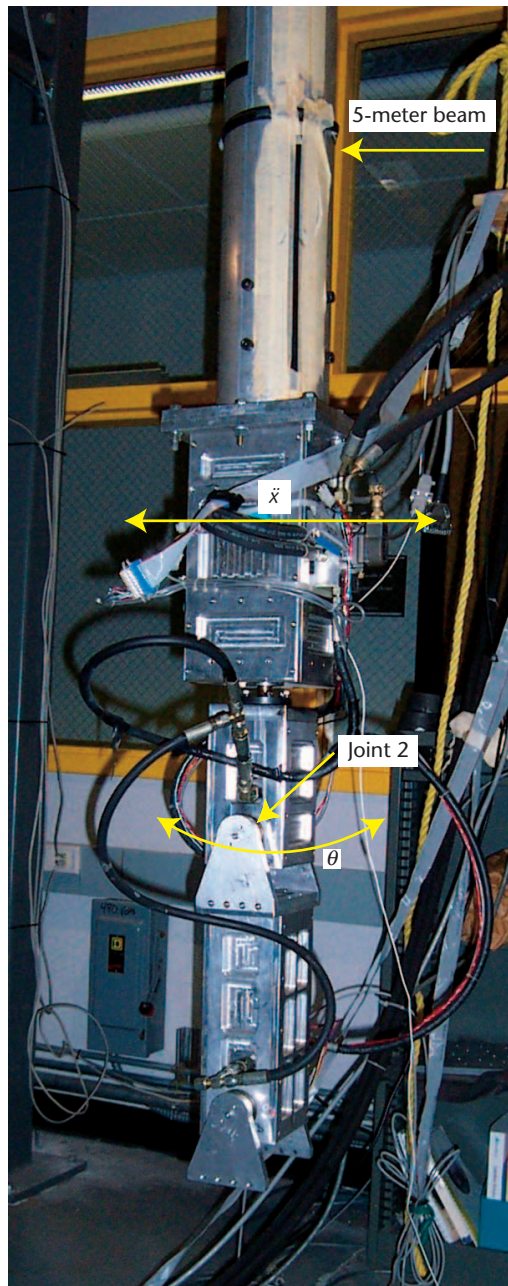


Figure 1. Picture of SAMII. Joint 2's angular position is θ , and \ddot{x} is the acceleration of SAMII's base (that is, the end of the cantilever beam).

The Need for Better Modeling Tools

Many approaches exist for modeling flexible structures, but two of the most prominent are finite element analysis (FEA) and the assumed modes method (AMM).

FEA is widely used in the analysis of flexible structures, but it isn't the ideal tool for control design. It's difficult, for example, to find an FEA software package that can model multiple feedback

loops and hydraulic actuators. Moreover, even if we found a suitable package that could model the closed-loop response of hydraulically actuated flexible robots, we couldn't use such a model for control design without modal discretization. It would be a clumsy approach for very flexible robots in which the feedback controller affects the system's mode shapes.

Other researchers have applied the AMM to robotics,⁶ but the approach quickly grows unwieldy as the number of links increases. Correctly handling element-connectivity conditions as we add more links to the model is quite burdensome, and it would quickly become very complicated if we applied this approach to the robot analyzed here. Additionally, this approach is clumsy for extremely flexible robots.

Expanding the TMM's Capabilities

The TMM has the potential to overcome other methods' shortcomings. It doesn't grow unwieldy as more links are added to the model, and it handles element-connectivity conditions exactly and automatically. It can also handle distributed parameter elements without discretization, so very flexible robots don't pose any additional challenges. The TMM lends itself to control design because the method outputs Bode plots (that is, magnitude and phase plots of the complex valued transfer functions between system inputs and outputs) very naturally, and it's easy to incorporate feedback.

However, we had two large obstacles to overcome before we could use the TMM to model SAMII: hydraulic actuators and non-collocated feedback.

The TMM models each element in a system with a matrix that transfers a state vector from one end of the element to the other. Each matrix is multiplied by the state vector at the end of the preceding element; the system transfer matrix comes from multiplying the element transfer matrices together. For example, SAMII's open-loop system transfer matrix is given by

$$\mathbf{U}_{\text{sys}} = \mathbf{U}_{\text{bs}} \mathbf{U}_{\text{beam}} \mathbf{U}_{j0} \mathbf{U}_{j1} \mathbf{U}_{j1} \mathbf{U}_{\text{act}} \mathbf{U}_{j2} \mathbf{U}_{j3} \mathbf{U}_{j3-6}, \quad (1)$$

where \mathbf{U}_{bs} is the transfer matrix for the basespring, \mathbf{U}_{beam} is the transfer matrix for the beam, and so on for each element in Figure 3's schematic. Because of this approach, we can't use a state from several elements back in the model, which prohibits accurate representation of the physical system. For practical reasons, our vibration-suppression scheme is based on accelerometers mounted on the end of the cantilever beam. The

accelerometer signal is fed back into the control scheme through the actuator of joint 2, which is several elements away in the TMM model. We developed a transfer matrix for non-collocated feedback based on symbolically inverting a transfer matrix from the sensor location to the actuator location. This transfer matrix enables the TMM to determine the sensor state based on the states at the actuator location (that is, the states that will multiply the actuator transfer matrix). We've experimentally verified these matrices for non-collocated feedback in a model that correctly predicts the system's closed-loop response.

We also developed a transfer matrix model for a hydraulic actuator interacting with the flexible structure. Similarly, we experimentally verified the model and found that it accurately captures the interaction between the actuator and the structure at resonance.

A Python Module for Analyzing Flexible Robots

We created a Python module for object-oriented analysis of flexible structures via the TMM. The object-oriented nature of the software leads to clean, clear, and easy-to-maintain code and provides a framework for user extensibility through inheritance. A user can define a new transfer matrix element by deriving from the base class, which clearly defines what properties and methods the new element must have to be valid.

The module uses two primary classes for TMM analysis: `TMMElement` and `TMMSystem`. Examples of `TMMElement` include flexible links (beam elements), rigid links, torsional springs, hydraulic actuators, and feedback elements (possibly non-collocated); users can develop them by deriving from the base class. The `TMMElement` is the TMM model's primary building block, and it has two primary methods: `GetMat` and `GetHT`. The former takes the frequency variable s as an input and returns the element transfer matrix, which the `TMMSystem` then uses to form the system transfer matrix according to Equation 1. `GetHT` returns the homogeneous transformation matrix for the element used in the 3D visualization of the mode shapes.

A `TMMSystem` consists of a list of serially connected `TMMElements` along with a specification of the system boundary conditions and the output signals to be calculated. The `TMMSystem` class has methods for finding a system's natural frequencies and mode shapes, generating Bode plots of specific outputs, system identification, and so on.

Figure 3 shows a picture of SAMII along with a

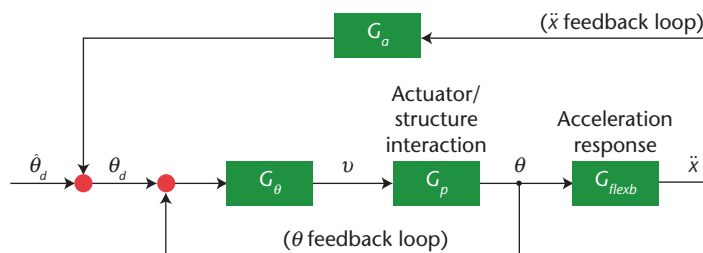


Figure 2. Block diagram. The system requires motion control (θ feedback) and vibration suppression (\ddot{x} feedback).

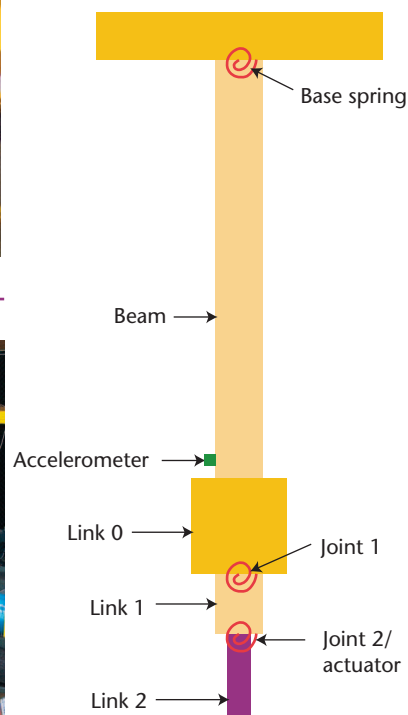
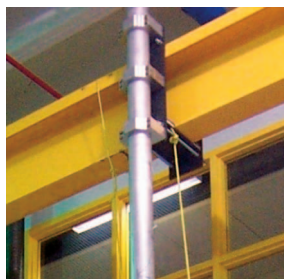


Figure 3. SAMII. Both the picture and schematic illustrate the transfer matrix model.

schematic; each element in the schematic is an object in the code from Figure 4. Each of Lines 2 through 9 in Figure 4 creates a `TMMElement` object from a derived class (`TorsionalSpring-Damper4x4`, `samiiBeam`, `samiiLink0`, and so on). Each object models a specific physical piece of the robot by calculating a transfer matrix that captures that piece's dynamics. Line 7 creates an `AVSwThetaFB` element that models a hydraulic actuator under θ feedback, including the interaction between the structure and the actuator. Line 8 creates a `SAMIIAccelFB` element that models

```

1 def accfbsamiimodel(xf,Ga=1):
2     basespring = TorsionalSpringDamper4x4({'k':xf[0],'c':xf[1]},
        unknownparams=['k','c'])
3     beam = samiiBeam()
4     link0 = samiiLink0()
5     j1spring = TorsionalSpringDamper4x4({'k':xf[2],'c':xf[3]},
        unknownparams = ['k','c'])
6     link1 = samiiLink1()
7     claws = AVSwThetaFB({'Ka':xf[4],'tau':xf[5],'ks':xf[6],'c':xf[7],'
        Gc':180.0/pi},unknownparams=['Ka','tau','ks','c'])
8     accfb = SAMIIAccelFB(link0,j1spring,link1,claws,Ga = Ga)
9     link2 = samiiLink2()
10    bodeout1 = bodeout(input='j2dhat',output='j2a',type='diff',ind=[
        claws,link1],dof=1)
11    bodeout2 = bodeout(input='j2dhat',output='a1',type='abs',ind=beam,
        post='accel',dof=0,gain=xf[8])
12    return ClampedFreeTMMSystem([basespring,beam,link0,j1spring,link1,
        accfb,claws,link2],bodeouts=[bodeout1,bodeout2])

```

Figure 4. Python code to create the transfer matrix method (TMM) model of SAMII corresponding to Figure 3.

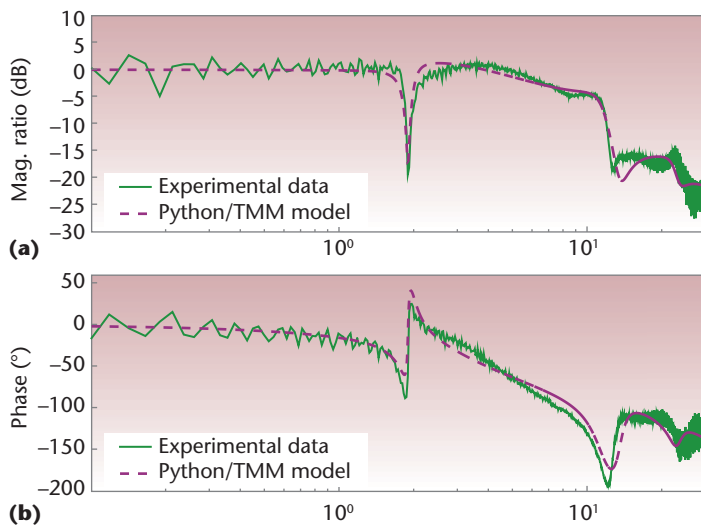


Figure 5. Closed-loop actuator Bode plot $\theta/\hat{\theta}_d$. We're comparing experimental data to the transfer matrix model for the system that has θ and \ddot{x} feedback (vibration suppression).

the non-collocated accelerometer feedback, where the sensor is at the end of the beam and the actuator is at joint 2. Lines 10 and 11 specify the output signals to calculate, and Line 12 returns an object derived from `TMMSystem`, in which the system boundary conditions are specified (clamped-free). The entire system model is created in just 12 lines.

Capabilities

Once we've created a system model, the software provides many capabilities for analysis, system identification, and control design. Examples include finding the system's natural frequencies and mode shapes, automated system identification, Bode analysis with user-defined outputs, and control design and optimization. We can do control design by optimizing multiple Bode plots or optimizing the closed-loop pole locations.

The software also has the ability to find closed-form symbolic expressions for the closed-loop system response by using Python to automatically write an input script to Maxima, which does the symbolic analysis and outputs its results to Fortran files. We can import these Fortran files into Python in two ways: compile them and use `f2py` to connect to the compiled code, or automatically parse the Fortran code into Python modules that we can directly import. All of this can happen without forcing the user to learn Maxima or Fortran—or even knowing that they're being used.

Example Usage and Results

Now that we've created the system model, it's very straightforward to generate Bode plots for the system's open- or closed-loop response. As an example of the software's predictive capabilities, Figures 5 and 6 show Bode plots of the system with both the motion control and vibration-suppression loops closed (that is, like the closed-loop system illus-

trated in Figure 2). We find excellent agreement between model and experiment.

We chose Python for our work for two main reasons: Python makes object-oriented programming easy, and many scientific and engineering modules are available for us to build on. We also found that our coding time was greatly reduced through interactive development with IPython (<http://ipython.scipy.org>). We plan to continue this work by developing Python modules for simulating interconnected dynamic systems and for rapidly implementing the control schemes on real-time embedded hardware.

References

1. E.C. Pestel and F.A. Leckie, *Matrix Methods in Elastomechanics*, McGraw-Hill, 1963.
2. W.J. Book, *Modeling, Design and Control of Flexible Manipulator Arms*, PhD dissertation, Dept. of Mechanical Eng., Massachusetts Inst. of Tech., Apr. 1974.
3. W.J. Book, O. Maizza-Neto, and D.E. Whitney, "Feedback Control of Two Beam, Two Joint Systems with Distributed Flexibility," *J. Dynamic Systems, Measurement and Control*, vol. 97, no. 4, 1975, pp. 424-431.
4. R.W. Krauss, O. Bruls, and W.J. Book, "Two Competing Linear Models for Flexible Robots: Comparison, Experimental Validation, and Refinement," *Proc. 2005 Am. Control Conf.*, IEEE CS Press, 2005, pp. 1963-1968; http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=1470257.
5. R.W. Krauss, *An Improved Technique for Modeling and Control of Flexible Structures*, PhD dissertation, Dept. of Mechanical Eng., Georgia Inst. of Tech., Aug. 2006; <http://etd.gatech.edu/theses/available/etd-06202006-185450/>.
6. A. Deluca and B. Siciliano, "Closed-Form Dynamic-Model of Planar Multilink Lightweight Robots," *IEEE Trans. Systems Man and Cybernetics*, vol. 21, no. 4, 1991, pp. 826-839.

Ryan W. Krauss is an assistant professor in the mechanical engineering department at Southern Illinois University Edwardsville. His research interests include control of flexible structures, applied controls, automotive crashworthiness, and technical computing. Krauss has a PhD in mechanical engineering from the Georgia Institute of Technology. Contact him at rkrauss@siue.edu.

Wayne J. Book is the Husco/Ramirez Distinguished Professor in Fluid Power and Motion Control in the George W. Woodruff School of Mechanical Engineering at the Georgia Institute of Technology. His research interests include fluid power, motion control, haptics, and hardware-in-the-loop simulations. Book has a PhD in mechanical engineering from MIT. He is a fellow of the American Society of Mechanical Engineers and the IEEE. Contact him at wayne.book@me.gatech.edu.

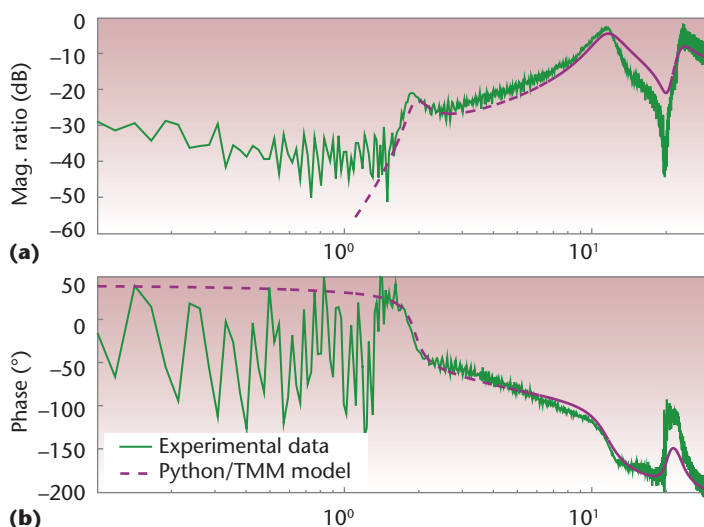


Figure 6. Closed-loop flexible base Bode plot $\ddot{x} / \hat{\theta}_d$. We're comparing experimental data to the transfer matrix model for the system that has θ and \ddot{x} feedback (vibration suppression).

FREE Visionary Web Videos
about the Future of Multimedia.

Listen to premiere
multimedia experts!

Post your own views and demos!

Visit www.computer.org/multimedia

Python in Nanophotonics Research

The authors describe how they use Python for nanophotonics research—specifically, they describe using it for electromagnetic modeling, mask design, and process simulation.

In our photonics research group at Ghent University, we're very active in the field of *nanophotonics*, which takes a complex optical system the size of a large table and shrinks it so that it fits onto a photonic chip of a few mm². Miniaturization and integration have done wonders for electronics in the past few decades, and the hope is that a similar strategy can work for photonics, too, leading to highly performant optical chips for fields as diverse as high-speed telecommunications, optical computing, and biosensors.

The electronics industry has spent billions of dollars perfecting fabrication technology in silicon, so it seems like a smart idea to piggyback on their mature processes for photonics applications. This is why our group is working on photonic structures made in silicon-on-insulator, which we're fabricating with the same state-of-the-art deep-UV lithography that produced the latest computer chip. These structures are typically less than a micron in size and can guide light along narrow waveguides, make very tight bends, or squeeze light into extremely small volumes.

In getting to a working device, however, we face several challenges and rely heavily on Python to overcome them.

Challenges and Advantages

A first step for any photonics research involves fig-

uring out how light behaves in complicated structures. For our particular studies, we use an in-house-developed Maxwell solver (<http://camfr.sourceforge.net>). Its core is written in C++, and it uses a few legacy Fortran routines, but its interaction with our simulator (to define the structure to be simulated, the quantities to be calculated, and so forth) happens via Python scripts, glued to C++ via Boost.Python. This C++ library makes it easy to expose C++ code to Python, is very powerful, and provides support for advanced C++ options. Its drawback, though, is that compilation times and memory requirements can be quite heavy. Figure 1 shows an example of a nanolaser's optical field, as calculated with our electromagnetics solver CAMFR.

Using Python

Once we come up with a good design, we still have to fabricate it, which involves designing a mask. Python scripts can help us define these masks: because Python is a full-fledged programming language, it's easy to parameterize the design or create repetitive structures using loops. Once the mask is finished, we place it in a deep-UV stepper to project the design on a photosensitive resist spun on a silicon wafer. Unfortunately, the pattern that ends up on the wafer isn't the same as the one on the mask, due to the projected light's diffraction, peculiarities in the etching process, and so forth. To get around this, we used NumPy or SciPy to write a process simulator that can calculate various effects.

As Figure 2 illustrates, Python can take us from electromagnetic design to mask layout and process technology simulation. This ability also lets us close the loop and, for example, recalculate the electromagnetic properties of the actual resulting geometry as predicted by the technology simula-

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

PETER BIENSTMAN, LIEVEN VANHOLME, WIM BOGAERTS,
PIETER DUMON, AND PETER VANDERSTEEGEN

Ghent University

tor, compare it to the nominal design, and make some modifications and precorrections, if needed.

All Python tools have a single aspect in common: they must be able to handle a structural definition (in terms of geometric primitives). For our research, we designed a generic class library that deals with geometric prototypes and creates a “little” language on top of Python to define a structure (such as the one in Figure 3):

```
air = Material(1) # Air has a refrac-
tive index of 1.
mat = Material(3) # Our material has a
refractive index of 3.
```

```
g = Geometry(air) # Air is the back-
ground material.
```

```
# Now add some shapes.
```

```
g += Rectangle(Point(0.0, 1.0),
Point(2.0, 2.0), mat)
g += Triangle(Point(2.0, 2.0),
Point(2.0, 1.0), Point(3.0, 1.5), mat)
g += Circle(Point(4.5, 1.5), 0.5, mat)
```

Although we could probably implement similar approaches in different languages, Python’s increased productivity makes it a very attractive option for us.

We’re currently extending and formalizing the definition of our “little” language, such that it will be powerful enough to use as input for our Maxwell solver and to automatically generate a mask description from it. We also plan to write a wrapper around other third-party simulation software, such that these generic structure definitions can be used as inputs for a wide variety of tools.

Peter Bienstman is an associate professor at Ghent University, Belgium. His research interests include nanophotonics and scientific computing. Contact him at Peter.Bienstman@UGent.be.

Lieven Vanholme works in the Photonics Research Group at Ghent University. His research interests include programming and physics. Contact him at Lieven.Vanholme@UGent.be.

Wim Bogaerts is a postdoc in the photonics group at Ghent University. His research interests include silicon nanophotonics. Contact him at Wim.Bogaerts@UGent.be.

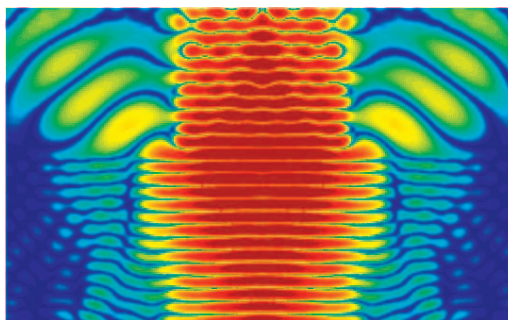


Figure 1. Optical field in a nanolaser, as calculated by our electromagnetics solver CAMFR.

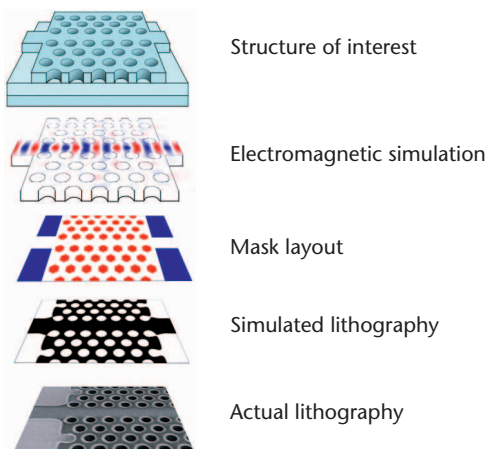


Figure 2. Python in the process.

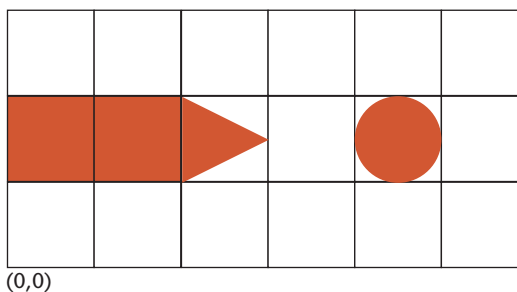


Figure 3. Example of a simple geometry.

Pieter Dumon is a PhD student in electronic engineering at Ghent University. His research interests include simulation and fabrication of silicon nanophotonic components. Contact him at Pieter.Dumon@UGent.be.

Peter Vandersteegen is a PhD student in the photonics group at Ghent University. His research focuses on organic LEDs and simulation methods. Contact him at Peter.Vandersteegen@UGent.be.