

ALSO

The Return of the
Books Dept. p. 3

Making the
Complex Simple p. 84

An Ice-Free
Arctic? p. 65



May/June 2007

Computing

in **SCIENCE & ENGINEERING**

Computing in Science & Engineering is a peer-reviewed, joint publication of the IEEE Computer Society and the American Institute of Physics



May/June 2007

Hajm

PYTHON: BATTERIES INCLUDED



AMERICAN
INSTITUTE
OF PHYSICS
<http://cise.aip.org>

IEEE
computer
society
www.computer.org/cise/

Computing

in **SCIENCE & ENGINEERING**

PYTHON: BATTERIES INCLUDED

Guest Editor's Introduction

Paul F. Dubois

7

Python for Scientific Computing

Travis E. Oliphant

10

IPython: A System for Interactive Scientific Computing

Fernando Pérez and Brian E. Granger

21

Computational Physics Education with Python

Arnd Bäcker

30

Python Unleashed on Systems Biology

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

34

Reaching for the Stars with Python

Perry Greenfield

38

A Python Module for Modeling and Control Design of Flexible Robots

Ryan W. Krauss and Wayne J. Book

41

Python in Nanophotonics Research

Peter Bienstman, Lieven Vanholme, Wim Bogaerts, Pieter Dumon, and Peter Vandersteegen

46

Using Python to Solve Partial Differential Equations

Kent-Andre Mardal, Ola Skavhaug, Glenn T. Lines, Gunnar A. Staff, and Åsmund Ødegård

48

Analysis of Functional Magnetic Resonance Imaging in Python

K. Jarrod Millman and Matthew Brett

52

Python for Internet GIS Applications

Xuan Shi

56

Quantum Chaos in Billiards

Arnd Bäcker

60

INTERNATIONAL POLAR YEAR

An Ice-Free Arctic? Opportunities for Computational Science

L. Bruno Tremblay, Marika M. Holland, Irina V. Gorodetskaya, and Gavin A. Schmidt

65

Statement of Purpose

Computing in Science & Engineering aims to support and promote the emerging discipline of computational science and engineering and to foster the use of computers and computational techniques in scientific research and education. Every issue contains broad-interest theme articles, departments, news reports, and editorial comment. Collateral materials such as source code are made available electronically over the Internet. The intended audience comprises physical scientists, engineers, mathematicians, and others who would benefit from computational methodologies.

All articles and technical notes in CISE are peer-reviewed.



Printed on 100% recycled paper

Cover illustration: Dirk Hagner



MAY/JUNE 2007

computing

in **SCIENCE & ENGINEERING**

COLUMNS

The First Word

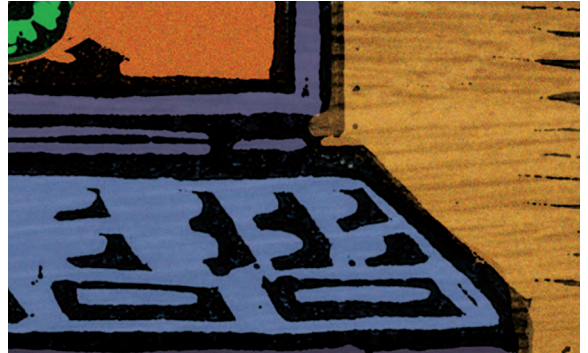
Norman Chonacky, Editor in Chief
You're Recommending What?!

2

The Last Word

Francis Sullivan
Wrong Again!

96



DEPARTMENTS

3

Books

Michael Jay Schillaci
Computationally Complete

75

Education

Christopher R. Myers and James P. Sethna
Python for Education:
Computational Methods for Nonlinear Systems

80

Your Homework Assignment

Dianne P. O'Leary
A Partial Solution to Beetles, Cannibalism, and Chaos:
Analyzing a Dynamical System Model

84

Computing Prescriptions

Julian V. Noble
Making the Complex Simple

90

Scientific Programming

John D. Hunter
Matplotlib: A 2D Graphics Environment

How to Contact *CiSE*, p. 6

Computer Society Membership Info, p. 9

AIP Membership Info, p. 29

Advertiser/Product Index, p. 95

WWW.COMPUTER.ORG/CISE/
[HTTP://CISE.AIP.ORG](http://CISE.AIP.ORG)

YOU'RE RECOMMENDING WHAT?!

By Norman Chonacky, Editor in Chief



THIS MAGAZINE'S EDITORIAL BOARD JUST CONCLUDED ITS ANNUAL MEETING AT THE AMERICAN CENTER FOR PHYSICS IN COLLEGE PARK, MARYLAND. DURING THE PROCEEDINGS, I INTRODUCED SOME NEW BOARD MEMBERS AND ANNOUNCED THE RETIREMENT OF SOME OTHERS.

I'll take some space in the next issue to express my thanks to those who invested a piece of themselves in the life of *CiSE* and its predecessors. Naturally, the publication's evolution involves a concomitant evolution of its editorial board, but sometimes these retirements arise sadly and unexpectedly. Julian V. Noble, co-editor of our Computing Prescriptions department and a long-time professor of physics (recently professor emeritus) at the University of Virginia, very recently died (see p. 84); we will miss him.

It's at times of such transitions in life that I reflect about transitions in other contexts—in this case, the life of our publication. Gathered around the meeting table were both the young and the not-so-young, embodying a range of assets from energy to wisdom. Within the “younger” set are those people growing up in an information galaxy that sometimes seems to me to reside in another universe.

One of the big issues on the table was how to expand our ability to create a community among groups of scientists, engineers, and mathematicians who are drawn together by their common intellectual investment in computing. To the “youngsters,” the obvious answer is the Web. I tend to view the Web as a source of information rather than a medium for building a community, especially among people who don't know one another; but “that's the whole point,” these energetic enthusiasts exclaim. As I understand it, our mission is to serve a collection of people who have a common need but not an ethos. Yet, the assumption at the meeting is that a sizeable chunk of our putative community belongs to this Web generation: its members don't hesitate to exchange ideas and critiques in near real time with others about whose professional (let alone personal) character and competence they have almost no idea—it's the buzz that brings them together.

I must admit that I don't really “get it,” but in my ripe old

age I'm also not inclined to obstruct new possibilities even if they don't make complete sense to me at the time. And I deeply believe that we have a responsibility to build a community whose members are in need, perhaps serious need, of some ethos of shared intellectual interests. So you can count on seeing an experiment in *CiSE*-sponsored blogging in the not-too-distant future.

How soon? Well, one metric was evident during our meeting. While I was expounding about the algorithm for determining *CiSE*'s share of the income earned from the IEEE's digital library (Xplore) and asking for strategic moves that might ameliorate its decline, the “youngsters” were busy online buying a domain name for blogging purposes ... right there in front of me! I doubt it'll be very long at all before you have the opportunity to share your thoughts.

In the meantime, enjoy this issue, which is full of Pythonista enthusiasm—this, I *do* get—and continue to enjoy our tales from the far North with our International Polar Year theme track.

HOW YOU CAN CONTRIBUTE

1. Send a Letter to the Editor

Have we published something that caught your eye? Would you like to comment on any of our articles, or share an opinion about any of our departments? Please email senior editor Jenny Stout (jstout@computer.org).

2. Review for Us

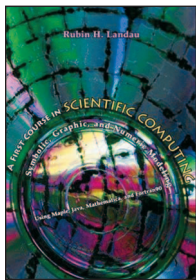
Email cise@computer.org with your vita and experience.



COMPUTATIONALLY COMPLETE

By Michael Jay Schillaci

R. Landau, *A First Course in Scientific Computing: Symbolic, Graphic, and Numeric Modeling Using Maple, Java, Mathematica, and Fortran90*, Princeton Univ. Press, 2005, ISBN: 0691121834, 472 pages.



Aimed directly at the undergraduate student, the stated goal of Ruben Landau's *A First Course in Scientific Computing: Symbolic, Graphic, and Numeric Modeling Using Maple, Java, Mathematica, and Fortran90*, is "[t]o provide them with tools and knowledge they can utilize throughout their college careers." Adopting a tutorial approach and

an "over the shoulder" style that lets students work independently and confidently, Landau addresses almost all the shortcomings of earlier computational physics books and produces a text that will certainly stand the test of time.

Drawing on both a long career of teaching excellence and from his prolific body of research, Landau wisely guides his readers through the scientific computing world with what at times feels like kind, grandfatherly advice. Indeed, in the preface—a part of the book that many students might unfortunately skip—he deftly focuses the reader's attention on the simple observation that, "[t]he basic ideas behind scientific computing are language independent, yet the details are not." With this guiding principle, he sensibly chooses Maple and Java as exemplars of the vast array of tools available to students. Together with the accompanying CD, which includes alternate code for Mathematica and Fortran90, the text's intended role is to be "closer to a workbook than a reference book." In this regard, Landau does very well. With a short chapter on LaTeX, he also addresses the ever-increasing role of electronic document production in the Web-based communication of scientific works and so provides the student with a complete set of tools for the task at hand.

Maple

To begin, although Landau's use of the command-line mode is appropriate because of platform variability concerns, the addition of a "walk through" of Maple's features would serve to give students a literal view of what's to come and could provide witness to the way in which students could integrate different programming environments. By demonstrating

Maple's ability to typeset mathematical equations, plot functions, and export graphics and pages as HTML or LaTeX, for example, students could quickly appreciate its usefulness. Instructors could then point to the examples as greater explanation and motivation for the course's scope and content.

Partly due to the workbook approach, the Maple discussion's overall organization is somewhat terse with many sections, subsections, and sub-subsections that don't always seem particularly pertinent to the task of "introducing the relevant mathematics in the course of solving realistic problems." This is especially true when using the book in courses designed to include students from curricula other than physics. For example, the first problem in the text draws on the results of special relativity, and Landau takes great care to entice the student saying, "[a]lthough the theory of special relativity does have its subtleties [...] [t]his should give you a good working knowledge of some tools." Given the text's audience, the first example should allow all students, not just physics majors, to fully grasp the phenomena's physical significance. Moreover, nearly 20 pages separate the introduction of relativistic equations with the plots of their basic behavior, with much of the intervening material coming from a computer science perspective. This is avoidable in an electronic format, but in written form, an organizational scheme that sets the tutorial information and exercises in sidebars or boxes would be preferable.

Although the mathematical and computational techniques Landau covers include the detailed use of integration and differentiation as well as matrix algebra and advanced plotting commands, too much is made of Maple's limitations. A measure of this is necessary, but in some cases this approach might challenge students' confidence in Maple—for example, Maple sometimes allows operators to be omitted (as when employing its version of scientific notation); a statement that using operators will decrease your debugging time would be useful in this circumstance. Moreover, when illustrating Maple's computer algebra system, the text focuses on the nature of commands such as `simplify` and `collect`, but it fails to give a detailed treatment of ex-

computing

in SCIENCE & ENGINEERING

EDITOR IN CHIEF

Norman Chonacky
cise-editor@aip.org

ASSOCIATE EDITORS IN CHIEF

Jim X. Chen, George Mason Univ.,
jchen@cs.gmu.edu

Denis Donnelly, Siena College, donnelly@siena.edu

Steven Gottlieb, Indiana Univ., sg@indiana.edu

Douglass E. Post, Carnegie Mellon Univ.,
post@ieee.org

EDITORIAL BOARD MEMBERS

Klaus-Jürgen Bathe, MIT, kjb@mit.edu

Antony Beris, Univ. of Delaware, beris@che.udel.edu

Michael W. Berry, Univ. of Tennessee, berry@cs.utk.edu

Bruce Boghosian, Tufts Univ., bruce.boghosian@tufts.edu

Hans-Joachim Bungartz, Institut für Informatik,
bungartz@in.tum.de

George Cybenko, Dartmouth College, gvc@dartmouth.edu

Jack Dongarra, Univ. of Tennessee, dongarra@cs.utk.edu

Rudolf Eigenmann, Purdue Univ., eigenman@ecn.purdue.edu

David Eisenbud, Mathematical Sciences Research Inst.,
de@msri.org

William J. Feiereisen, Los Alamos Nat'l Lab, wjf@lanl.gov

Geoffrey Fox, Indiana Univ., gcf@grids.ucs.indiana.edu

Sharon Glotzer, Univ. of Michigan, sglotzer@umich.edu

Anthony C. Hearn, RAND, hearn@rand.org

Charles J. Holland, Darpa, charles.holland@darpa.mil

M.Y. Hussaini, Florida State Univ., myh@cse.fsu.edu

Rachel Kuske, Univ. of British Columbia, rachel@math.ubc.ca

David P. Landau, Univ. of Georgia, dlandau@hal.physast.uga.edu

R. Bowen Loftin, Texas A&M University, Galveston,
loftin@tamug.edu

B. Vincent McKoy, California Inst. of Technology,
mckoy@its.caltech.edu

Jill P. Mesirov, Whitehead/MIT Ctr. for Genome Research,
mesirov@genome.wi.mit.edu

Constantine Polychronopoulos, Univ. of Illinois,
cdp@csrd.uiuc.edu

William H. Press, Los Alamos Nat'l Lab., wpress@lanl.gov

John Rice, Purdue Univ., jrr@cs.purdue.edu

John Rundle, Univ. of California, Davis,
rundle@physics.ucdavis.edu

Ahmed Sameh, Purdue Univ., sameh@cs.purdue.edu

Henrik Schmidt, MIT, henrik@mit.edu

Greg Wilson, Univ. of Toronto, gvwilson@third-bit.com

CONTRIBUTING EDITORS

Francis Sullivan, The Last Word, fran@super.org

Paul F. Dubois, Café Dubois, paul@pfdubois.com

NOT SO NEW, BUT IMPROVED

In early 2006, the *CiSE* editorial board evaluated the Books department. The outcome of this discussion was that the department emerged as an important section of the magazine that we needed to rework and update. With *CiSE*'s larger constituency, the Books department seeks to have "something for everyone," so our goal is to feature at least one book review per issue, which will come from these general categories:

- programming (design, tools, techniques, languages);
- computational sciences (engineering, biology, chemistry, physics, and so on);
- engineering and science applications (CAD/CAE, productivity, modeling, simulation);
- algorithms (high-performance and parallel computing, numerical, symbolic);
- data (mining, databases, analyses);
- textbooks (computer science or science with computation);
- laboratory/experimental (acquisition, control systems, imaging, sensors); and
- hardware/networks (special purpose, distributed systems, communications).

However, the Books department is only as good as the books we receive, and the reviewers who review them. If you know of a good book worthy of review, or want to review a book in one of these categories, please contact me at mabelloni@davidson.edu.

We kick off this revamped department with Michael Jay Schillaci's review of Rubin Landau's book, *A First Course in Scientific Computing*. Much debate exists as to what a first course in scientific computing should be (see for example, the September/October 2006 issue of *CiSE*), and Landau's approach has been at the forefront of much of this discussion. In addition to our review, the *American Journal of Physics* recently published a comparison review of three first-course books (vol. 74, no. 7, 2006). Our reviewer also has his own first-course approach, which you can download as a PDF from his Web site (www.evis.org/download.html).

tremely important commands like `unapply`. Consequently, the opportunity to impart high-order heuristics to the student is lost. By including short tables or intermittent summaries of important built-in functions and commands in the text (in addition to those already in the appendix) and with a more judicious use of the command-line help system, the delivery and impact of the examples could be improved.

When tackling more complex ideas, the text's colloquial approach often comes across as abrupt. For instance, in dis-

Discussing how a 3D plot structure is rendered on a 2D surface, Landau says, “we do that by rotating the object, shading it, employing parallax, and so forth.” Arguably, one of the most appealing aspects of software packages such as Maple or Mathematica is that the nontechnical masses can use them to blindly produce complex and beautiful graphics, but one of a scientific computing course’s many goals is to give students a deeper understanding of the principles involved. For this reason, it might have been more beneficial in this context if the text demonstrated how the student could add plot options such as viewpoint and shading to the basic command structure to alter the object’s appearance. Then, the student could use the matrix rotation techniques to construct a simple animation sequence, directly illustrating the effects and providing a concrete and in-depth example. Indeed, with a similar approach to each of the chapter’s problems, Landau could enhance the coverage of Maple methods and programming.

Java

The thorough treatment of Java as a paradigm example of a modern object-oriented programming (OOP) language, complete with plotting and Web-based applications, is where Landau’s text makes its real contribution. Although his first example might seem a bit mundane—calculating a circle’s area—it goes beyond the traditional “hello world” program and introduces the language’s method-based structure. In particular, Landau’s discussion of classes and methods and when to avoid thinking too deeply about the required syntax of declarations is very refreshing. To quickly demonstrate Java’s power and appeal, he then shows how easy it can be to produce graphical output.

Despite the fact that some of the physical examples covered in the Maple section weren’t fully developed, Landau’s use and expansion of these same examples later in the text (most notably, the detailed simulation of a large city’s electricity usage) should let students move confidently into modern scientific computing’s more technical aspects. Moreover, his addition of “new” problems such as frictionless projectile motion provide students with a demonstration of how theory, algorithm development, and logic must all come together to produce working simulations. The comparison of Java and Maple solutions to differential equations helps further drive home this point and also provides the unified approach that’s lacking in some of the text’s earlier portions.

Landau employs a “just enough” approach when it comes to his discussions of OOP concepts, but deals with the ideas of encapsulation and inheritance sufficiently, making the

DEPARTMENT EDITORS

Books: Mario Belloni, Davidson College, mabelloni@davidson.edu
Computing Prescriptions: Isabel Beichl, Nat’l Inst. of Standards and Tech., isabel.beichl@nist.gov
Computer Simulations: Muhammad Sahimi, University of Southern California, moe@iran.usc.edu, and Dietrich Stauffer, Univ. of Köln, stauffer@thp.uni-koeln.de
Education: David Winch, Kalamazoo College, winch@TaosNet.com
News: Rubin Landau, Oregon State Univ., rubin@physics.oregonstate.edu
Scientific Programming: Konstantin Läuffer, Loyola University, Chicago, klauffer@cs.luc.edu, and George K. Thiruvathukal, Loyola University, Chicago, gkt@cs.luc.edu
Technology: James D. Myers, jimmyers@ncsa.uiuc.edu
Visualization Corner: Claudio T. Silva, University of Utah, csilva@cs.utah.edu, and Joel E. Tohline, Louisiana State University, tohline@rouge.phys.lsu.edu
Your Homework Assignment: Dianne P. O’Leary, Univ. of Maryland, oleary@cs.umd.edu

STAFF

Senior Editor: Jenny Stout, jstout@computer.org
Group Managing Editor: Steve Woods
Staff Editors: Kathy Clark-Fisher, Rebecca L. Deuel, and Brandi Ortega
Contributing Editor: Joan Taylor
Production Editor: Monette Velasco
Publications Coordinator: Hazel Kosky, cise@computer.org
Technical Illustrator: Alex Torres
Publisher: Angela Burgess, aburgess@computer.org
Associate Publisher: Dick Price
Advertising Coordinator: Marian Anderson
Marketing Manager: Georgann Carter
Business Development Manager: Sandra Brown

AIP STAFF

Circulation Director: Jeff Bebee, jbebee@aip.org
Editorial Liaison: Charles Day, cday@aip.org

IEEE ANTENNAS AND PROPAGATION SOCIETY LIAISON

Don Wilton, Univ. of Houston, wilton@uh.edu

IEEE SIGNAL PROCESSING SOCIETY LIAISON

Elias S. Manolakos, Northeastern Univ., elias@neu.edu

CS PUBLICATIONS BOARD

Jon Rokne (chair), Mike Blaha, Angela Burgess, Doris Carver, Mark Christensen, David Ebert, Frank Ferrante, Phil Laplante, Dick Price, Don Shafer, Linda Shafer, Steve Tanimoto, Wenping Wang

CS MAGAZINE OPERATIONS COMMITTEE

Robert E. Filman (chair), David Albonesi, Jean Bacon, Arnold (Jay) Bragg, Carl Chang, Kwang-Ting (Tim) Cheng, Norman Chonacky, Fred Douglass, Hakan Erdogmus, David A. Grier, James Hendler, Carl E. Landwehr, Sethuraman (Panch) Panchanathan, Maureen Stone, Roy Want

EDITORIAL OFFICE

COMPUTING in SCIENCE & ENGINEERING
 10662 Los Vaqueros Circle, Los Alamitos, CA 90720 USA
 phone +1 714 821 8380; fax +1 714 821 4010; www.computer.org/cise/



move to Web computing easier to appreciate. Because of a small typographical error (probably placed there by Landau to encourage students and instructors to type in their own code!), the first Web application didn't run "out of the box." By extending this basic applet project to include user control, he implicitly reminds us that much of scientific computing's lineage was based on copious use of legacy code and libraries. However, the follow-through absent in the Maple section is also evident here as Landau stops just short of a full-blown Web application using JavaScript to allow for direct user interaction via form input.

LaTeX

Despite the fact that the text is formally broken into three

sections, the third section—entitled, "The LaTeX Survival Guide"—is very short. Nevertheless, it does an adequate job of introducing the essential LaTeX commands and environments that a student would need to produce high-quality and content-rich documents suitable for laboratory reports, refereed journal submissions, or Web postings. The reasons for including LaTeX in a first course in scientific computing are clear in that the superior mathematical typesetting ability and the resulting electronic documents (that is, EPS or PDF) are ubiquitous. However, LaTeX's integration and connection to the extant material in the text and course isn't evident in Landau's treatment. Specifically, no discussion of styles and packages is included—which is essential if you want to compile the LaTeX source from a Maple worksheet. This level of detail isn't often found in the many TeX primers available online but adds a decidedly more complete view of the subtleties of electronic document production. Instead of viewing this as a liability, you could rearrange the text material to cover LaTeX elements first and then move to Maple (or Mathematica). This would have the added benefit of allowing nontraditional students to learn and refresh their programming and debugging skills while working with less cognitively demanding material.

It's arguable that an introductory scientific computing course ought to be accessible to all students of the broader sciences. A conscientious instructor willing to reorganize and extend some material to make it more suitable and appealing to a multidisciplinary student body could use *A First Course in Scientific Computing* to cast this wider net. Indeed, the colloquial and tutorial approach might help alleviate the many practical problems associated with incorporating computational applications into a more traditional lecture environment. The text provides many concrete and programming examples in action and illustrates how much you can accomplish with a few well-chosen tools. All in all, students impressed with the text's workbook style and reference-book quality will add it to their bookshelves and return to it often.

Michael Jay Schillaci is managing director of the McCausland Center for Brain Imaging at the University of South Carolina. His research interests include computational physics and curriculum development, and models of human cognition using magnetic resonance imaging and electroencephalographic data. Schillaci has a PhD in physics from the University of Arkansas at Fayetteville. He is a member of the American Physical Society and the Cognitive Neurosciences Society. Contact him at mjs@sc.edu.



Writers

Visit www.computer.org/cise/author.htm.

Letters to the Editors

Send letters to Jenny Stout, Lead Editor, jstout@computer.org. Provide an email address or daytime phone number.

On the Web

Access www.computer.org/cise/ or <http://cise.aip.org>.

Subscribe

Visit https://www.aip.org/forms/journal_catalog/order_form_fs.html or www.computer.org/subscribe/.

Subscription Change of Address (IEEE/CS)

Send an email to address.change@ieee.org. Please specify CiSE.

Subscription Change of Address (AIP)

Send general subscription and refund inquiries to subs@aip.org.

Missing or Damaged Copies

Contact help@computer.org. For AIP subscribers, contact claims@aip.org.

Reprints of Articles

For price information or to order reprints, email cise@computer.org or fax +1 714 821 4010.

Reprint Permission

Contact William Hagen, IEEE Copyrights and Trademarks Manager, at copyrights@ieee.org.

www.computer.org/cise/

Python: Batteries Included



This issue's special theme is the computer programming language Python and the increasing role it plays in scientific projects. Free and universally available, Python comes with a vast standard library containing support for nearly every area of com-

puter science. An even more extensive set of third-party tools and modules covers additional tasks, from managing a Web site to doing a fast Fourier transform to distributed or parallel programming. Python's motto, "batteries included," is meant to convey the idea that Python comes with everything you need.

Interpreted Doesn't Mean Slow or only Interactive

Python is an interpreted language, and it can be used interactively. Some might assume that this limits its uses—for example, that an interpreted

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

PAUL F. DUBOIS
Contributing Editor

Table 1. Basic Python resources.

Description	Tool	URL
Python with standard library	Python	http://python.org
SciPy project (which includes Python, NumPy, and f2py)	SciPy/NumPy	http://scipy.org
Enhanced interactive Python	IPython	http://ipython.scipy.org
Two-dimensional graphics	Matplotlib	http://matplotlib.sf.net
Three-dimensional graphics	MayaVi	http://mayavi.sf.net
Connect Python to C/C++	SWIG	http://swig.org
	Boost/Python	http://boost.org
	PyCXX	http://cxx.sf.net
Connect Python to Fortran	f2py	(in SciPy)
Python Cheese Shop	2,000 more packages	http://cheeseshop.python.org

language can't possibly be fast enough for scientific programming—but as you'll see, this isn't true. Others might assume that an interactive language can't be used in a large code, or in a batch or parallel system, but that's not right either.

Python's numerical extension NumPy adds an array language similar in power to the one in modern Fortran, in which operations are performed in compiled code. Together with modules for numerical mathematics and graphics, Python by itself is a powerful computational tool. Moreover, it's easy to make your own compiled code callable from Python (and able to call Python itself). Various tools help you make this connection quickly and easily, and once you connect to Python, you have access to Python's "batteries."

Steering

My own interest in Python focuses on using it for computational steering: Python serves as the input language to a scientific application, and the actual computations are performed both in Python itself and in compiled extensions. This approach gives users the chance to be creative, serves as a built-in symbolic debugger and interactive graphics capability, and reduces development time. Nobody I know who has experienced it has ever been willing to be without it in the future.

I first wrote a system for producing steered code in 1984 at Lawrence Livermore National Laboratory. This system, called Basis, was such a new idea that it was difficult at the time to explain it to people, but it proved very successful and developers have written at least 200 applications with it. Some of the larger ones are still in use today, and Basis is still an active project (<http://basis.llnl.gov>). The key to its success is that the interpreted language I wrote for the steering was an array language quite similar to what was eventually in Fortran 95. By using the array operations, I could do real work in

the interpreter (besides calling the compiled routines to do the bulk of the work). Most importantly, the language was simple—it was enough like Fortran that users could easily read it and learn to write it.

However, trouble was coming. Basis supported Fortran 77, and I could see that not only was Fortran going to evolve but the object-oriented revolution was upon us. So in the early 1990s, I contemplated my "Act II." I even designed an object-oriented interpreter and implemented a prototype. We held periodic meetings with Basis users to discuss their requirements.

One day I found Python and saw that it had a great similarity to my prototype, was better thought out, and much further along. The one thing it lacked was an array-language extension, but a special-interest group was already looking into that. At the next meeting, I mentioned it favorably, and David Grote, a member of the group, said that he, too, had just discovered it and thought it would do the job. I decided to throw my efforts into helping design the array extension that became Numerical Python. Jim Hugunin volunteered to write the code; he later moved on to create Jython and IronPython, the Java and .NET versions of Python. I took over as the project's coordinator, and five years later, I passed the torch to Perry Greenfield (whose article about telescopes appears in this issue). Now the project is led by Travis Oliphant, who describes it more fully on p.10.

The happy ending here is that we made a good choice, and LLNL now has many Python-based efforts built from scratch or wrapped around legacy codes, and others that evolved from Basis codes: hundreds of thousands of lines of C++, Python, and Fortran 95, all working together just as we hoped, doing compute-intensive calculations on massively parallel computers.

In this Issue

We begin the issue with a basic introduction to Python in general and the SciPy project in particular. SciPy gathers the high-performance array extension together with many modules for doing common mathematical and statistical functions. Our next major article introduces the advanced interactive interpreter IPython and the matplotlib graphics package. IPython is the computational tool of choice for some people, used in much the same way as commercial products such as Matlab but with access to the full Python world and at no cost. Matplotlib is rapidly becoming accepted as the standard two-dimensional graphics utility for Python, and the Scientific Programming department on p. 90 discusses it in even greater detail.

After the larger introductory articles, we have a series of shorter pieces that present specific scientific, engineering, and educational applications. To show you a wide variety, we tried to extract the basic material on the language and its tools into the first two articles, so we suggest you read those first after sneaking a peek at the pretty pictures in the application pieces. An extra article on Python in the classroom appears in the Education department.

Although I asked the authors to state briefly why they find Python helpful, I also asked them not to extensively argue for it over some other technology choice. As in *Field of Dreams*, we think we've built it and that you will come once you see it.

I hope you enjoy our special issue and will try the Python approach to scientific computing. Table 1 should get you started, with a list of basic Python resources that are open source and available without charge. There are many, many more; start your hunt at the Python Cheese Shop (<http://cheeseshop.python.org>). ☺

Paul F. Dubois is retired and lives in Pleasanton, California, where he contributes to open source projects and writes for CiSE. His column "Cafe Dubois" will return next issue. Contact him at paul@pfdubois.com.

Coming up in our next issue

July/August: New Directions

The articles in this special issue cover a wide range of topics, from problem-solving environments to algorithm design to physics curricular material.

IEEE computer society

PURPOSE: The IEEE Computer Society is the world's largest association of computing professionals and is the leading provider of technical information in the field.

MEMBERSHIP: Members receive the monthly magazine *Computer*, discounts, and opportunities to serve (all activities are led by volunteer members). Membership is open to all IEEE members, affiliate society members, and others interested in the computer field.

COMPUTER SOCIETY WEB SITE: www.computer.org

OMBUDSMAN: Email help@computer.org.

Next Board Meeting: 18 May 2007, Los Angeles

EXECUTIVE COMMITTEE

President: Michael R. Williams*

President-Elect: Rangachar Kasturi; * **Past President:** Deborah M. Cooper; * **VP, Conferences and Tutorials:** Susan K. (Kathy) Land (1ST VP); * **VP, Electronic Products and Services:** Sorel Reisman (2ND VP); * **VP, Chapters Activities:** Antonio Doria; * **VP, Educational Activities:** Stephen B. Seidman; † **VP, Publications:** Jon G. Rokne; † **VP, Standards Activities:** John Walz; † **VP, Technical Activities:** Stephanie M. White; * **Secretary:** Christina M. Schober; * **Treasurer:** Michel Israel; † **2006–2007 IEEE Division V Director:** Oscar N. Garcia; † **2007–2008 IEEE Division VIII Director:** Thomas W. Williams; † **2007 IEEE Division V Director-Elect:** Deborah M. Cooper; * **Computer Editor in Chief:** Carl K. Chang †

* voting member of the Board of Governors † nonvoting member of the Board of Governors

BOARD OF GOVERNORS

Term Expiring 2007: Jean M. Bacon, George V. Cybenko, Antonio Doria, Richard A. Kemmerer, Itaru Mimura, Brian M. O'Connell, Christina M. Schober

Term Expiring 2008: Richard H. Eckhouse, James D. Isaak, James W. Moore, Gary McGraw, Robert H. Sloan, Makoto Takizawa, Stephanie M. White

Term Expiring 2009: Van L. Eden, Robert Dupuis, Frank E. Ferrante, Roger U. Fujii, Anne Quiroz Gates, Juan E. Gilbert, Don F. Shafer

EXECUTIVE STAFF

Associate Executive Director: Anne Marie Kelly; **Publisher:** Angela R. Burgess;

Associate Publisher: Dick J. Price; **Director, Administration:** Violet S.

Doan; **Director, Finance and Accounting:** John Miller

COMPUTER SOCIETY OFFICES

Washington Office. 1730 Massachusetts Ave. NW, Washington, DC 20036-1992

Phone: +1 202 371 0101 • Fax: +1 202 728 9614

Email: hq.ofc@computer.org

Los Alamitos Office. 10662 Los Vaqueros Circle, Los Alamitos, CA 90720-1314

Phone: +1 714 821 8380 • Email: help@computer.org

Membership and Publication Orders:

Phone: +1 800 272 6657 • Fax: +1 714 821 4641

Email: help@computer.org

Asia/Pacific Office. Watanabe Building, 1-4-2 Minami-Aoyama, Minato-ku, Tokyo 107-0062, Japan

Phone: +81 3 3408 3118 • Fax: +81 3 3408 3553

Email: tokyo.ofc@computer.org

IEEE OFFICERS

President: Leah H. Jamieson; **President-Elect:** Lewis Terman; **Past President:** Michael R. Lightner; **Executive Director & COO:** Jeffrey W. Raynes; **Secretary:** Celia Desmond; **Treasurer:** David Green; **VP, Educational Activities:** Moshe Kam; **VP, Publication Services and Products:** John Baillieul; **VP, Regional Activities:** Pedro Ray; **President, Standards Association:** George W. Arnold; **VP, Technical Activities:** Peter Staecker; **IEEE Division V Director:** Oscar N. Garcia; **IEEE Division VIII Director:** Thomas W. Williams; **President, IEEE-USA:** John W. Meredith, P.E.

revised 23 Mar. 2007



Python for Scientific Computing

By itself, Python is an excellent “steering” language for scientific codes written in other languages. However, with additional basic tools, Python transforms into a high-level language suited for scientific and engineering code that’s often fast enough to be immediately useful but also flexible enough to be sped up with additional extensions.

Python is an interpreted language with expressive syntax that some have compared to executable pseudocode. This might be part of the reason why I fell in love with the language in 1996, when I was seeking a way to prototype algorithms on very large data sets that overwhelmed the capabilities of the other interpreted computing environments I was familiar with. My enjoyment of programming with Python increased as I quickly learned to express complicated ideas in the syntax and objects available with it.

The idea that coding in a high-level language can greatly enhance productivity isn’t new. Many scientists and engineers are typically exposed to one or more interpreted scientific computing environments early in their careers because they help them write nontrivial computational programs without getting too bogged down in syntax and compilation time lags. Python can be used in exactly this way, but its unique features offer an environment that makes it a better choice for scientists and engineers seeking a high-level language for writing scientific applications. In the rest

of this special issue’s articles, you’ll find a feast of reasons why Python excels as a platform for scientific computing. As a small appetizer, consider this list of general features:

- A liberal open source license lets you sell, use, or distribute your Python-based application as you see fit—no extra permission necessary.
- The fact that Python runs on so many platforms means you don’t have to worry about writing an application with limited portability, which also helps avoid vendor lock-in.
- The language’s clean syntax yet sophisticated constructs let you write in either a procedural or fully object-oriented fashion, as the situation dictates.
- A powerful interactive interpreter allows real-time code development and live experimentation, thus eliminating the time-consuming and productivity-eating compile step from the code-then-test development process.
- The ability to extend Python with your own compiled code means that Python can be taught to do anything as fast as your hardware will allow.
- You can embed Python into an existing application, which means you can instantly add an easy-to-use veneer on top of an older, trusted application.
- The ability to interact with a wide variety of other software on your system helps you leverage the software skills you’ve already acquired.

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

TRAVIS E. OLIPHANT
Brigham Young University

- Its large number of library modules (both installed in the standard library and through additional downloads) means you can quickly construct sophisticated programs such as a Web server that solves partial differential equations, a distributed image-processing library with automatic load-balancing, or an application that collects data from the Internet, posts results to a database, and emails you with progress reports.
- The existence of Python bindings to all standard GUI toolkits means you can apply rapid development techniques when building a user interface.
- The Python community is famous for delivering quick, useful responses to user inquiries on the mailing lists, newsgroups, and IRC channels devoted to Python.
- A repository of categorized Python modules is available at www.python.org/pypi along with easy-to-install “eggs” that simplify software management.

With this small taste of Python’s usefulness, let’s dive in and try to uncover a little bit about the language itself. This overview is necessarily terse. Because Python is a general-purpose programming language, a wealth of additional information is available in several books and on Internet sites dedicated to it (see the “Useful References” sidebar). Python’s `help` function also provides additional information.

Clean Syntax

A significant factor in Python’s utility as a computing language for scientists and engineers is its clear syntax, which can make code easy to understand and maintain. Some of this syntax’s specifics include code blocks defined by indentation, extensive use of namespaces (modules), easy-to-read looping constructs (including list comprehensions), exception handling, and documentation strings. Consider the following code, which computes the `sinc` function on a list of inputs:

```
from math import sin, pi
def sinc(x):
    '''Compute the sinc function:
       sin(pi*x)/(pi*x)'''
    try:
        val = (x*pi)
        return sin(val)/val
    except ZeroDivisionError:
        return 1.0
output = [sinc(x) for x in input]
```

With only a little explanation, this code is com-

USEFUL REFERENCES

- D. Ascher et al., *Numerical Python*, tech. report UCRL-MA-128569, Lawrence Livermore Nat’l Lab., 2001; <http://numpy.scipy.org>.
- P.F. Dubois, K. Hinsin, and J. Hugunin, “Numerical Python,” *Computers in Physics*, vol. 10, no. 3, 1996, pp. 262–267.
- E. Jones et al., “SciPy: Open Source Scientific Tools for Python,” 2001; www.scipy.org.
- T.E. Oliphant, *Guide to NumPy*, Trelgol, 2006; www.trelgol.org.
- M. Pilgrim, *Dive into Python*, 2004; www.diveintopython.org.
- G. van Rossum and F.L. Drake, eds., *Python Reference Manual*, Python Software Foundation, 2006; <http://docs.python.org/ref/ref.html>.

pletely understandable. It shows the use of namespaces because the `sin` function and the `pi` constant are kept in a separate module and must be “imported” to be used. (Alternatively, we could have used `import math` along with `math.sin` and `math.pi` to refer to the objects). Modules are either files (with a “.py” suffix) with Python code or compiled shared libraries specifically built for import into Python. Each module contains a namespace that allows grouping of similar functions, classes, and variables in a manner that lets the code scale from simple scripts to complicated applications.

This code also demonstrates exception handling: the `try/except` syntax allows for separate handling when division fails. Notice, too, that the code block defining the function is indented, but otherwise offers no indication of the code’s beginning or end. This feature ensures that both the computer and the human reader have the same concept of code level without spurious characters taking up precious onscreen real estate.

The string in the code immediately after the function definition is the function *docstring*, which is useful when generating help messages or documenting functions in the output of automatic documentation tools such as `pydoc` and `epydoc`. The output list is generated from the input sequence via a compact looping construct called a *list comprehension*. This readable construct looks almost like executable English because it creates a new list whose elements are the output of `sinc(x)` for every `x` in the input sequence.

One of Python’s key features is dynamic typing. Notice that the type of the `sinc` function’s input is never specified—all we need is for the `sin` function to work on it. In this case, the `math` module’s `sin` function doesn’t handle complex numbers, but the

`cmath` module does, so we can use it to replace the `math` module if desired.

Useful Built-In Objects

Everything in Python is an object of a particular kind (or type); the standard way to construct a type is with a function call. Some types, however, are constructed from built-in, simplified syntax. The built-in scalar types are integer (infinitely large), floating point (double precision), and complex (double-precision real and imaginary parts). The syntax automatically constructs the scalars, as evidenced by the following interactive interpreter session:

```
>>> type(1), type(1.0), type(1.0j),
      type('one')
(<type 'int'>, <type 'float'>, <type
 'complex'>, <type 'str'>)
```

The '`>>>`' marker indicates that the interpreter is ready to receive code. Separating items with a comma automatically constructs an immutable (unchangeable once constructed) sequence called a *tuple*. In this example, I've also demonstrated the creation of a string object, which is another immutable object used extensively in Python code. An additional sequence in Python is a mutable (can be altered) list. A list can contain a sequence of any Python object (including extra lists); thus, we can use lists to construct simple multidimensional arrays:

```
>>> a = [['an', 'ecletic', 3], [5.0,
      'nothing']]
>>> print a[1][0]
5.0
```

This is an efficient, quick way to store and work with small arrays. For larger arrays, however, NumPy is better suited for managing memory and speed requirements (I'll discuss NumPy in more detail later).

Python's dictionary provides a very useful container object that lets us look up a value using a key. For example,

```
>>> a = {2: 'two', 'one': 1}
>>> print a['one'], a[2]
1 two
```

The object before the ':' here is the *key* and the object after the ':' is the *value*. The key must be immutable so that the lookup step happens quickly. Dictionaries are useful for storing data for later retrieval with a key, which is a common need in scien-

tific computing—we can even construct a very simple sparse-matrix storage scheme with dictionaries.

File objects are also important for most scientific computing projects. We use the `open` command to construct such objects:

```
>>> fid = open('simple.txt', 'w')
>>> fid.write("This is a string written
      to the file.")
>>> fid.close()
```

This example shows how to open a file, write a simple string to it, and then close it again (we can also close it by deleting the `fid` variable). To read from a file, we use the file object's `read` method to return a string with the file's bytes. In this example, I've also demonstrated that both double (") and single (') quotes can delimit a string constant as long as we open and close a particular string with the same quote style.

Functions and Classes

Besides offering clean syntax, Python also contributes to the construction of maintainable code by separating code into logical groups such as modules, classes (new object definitions), and functions. As I mentioned earlier, a module is a collection of Python code grouped together into a single file (with the ".py" extension), and it usually contains many related functions or classes. After using the `import` command on a module, we can use dot notation to access the functions, classes, and variables defined within the module:

```
>>> import numpy as N
>>> print N.linalg.inv([[1,2], [1,3]])
[[ 3., -2.],
 [-1., 1.]])
```

This code segment calls the `inv` function from the `linalg` submodule of the `numpy` module (which I renamed to the local variable `N` for the current session). The `inv` function works on any nested sequence and finds the "matrix" inverse of a two-dimensional array.

Functions are normally defined with the syntax `def funcname(...): <indented block>`. For simple functions, we can also use a `lambda` expression, which lets us write one-line (anonymous) functions. Here's a `lambda` expression that uses a recursive formula to compute a factorial:

```
f = lambda x: (x<1) or x*f(x-1)
```

This function takes one argument and returns one

object that should be the factorial for integer input. It works because the `or` operation doesn't compute the second operand if the first is true. Anonymous functions are occasionally useful, but the standard way to define functions is to use the `def` syntax:

```
def sum_and_mean(x, sumfunc=sum,
                norm=None):
    if norm is None:
        norm = len(x)
    y = sumfunc(x)
    return y, y/float(norm)
```

This function also illustrates the use of keyword arguments to define default values for optional function arguments, as well as the use of a tuple to return more than one result from the function. The resulting tuple can be unpacked when the function is called, giving the appearance that the function returns more than one argument:

```
tot, ave = sum_and_mean([1, 4, 10, 3.0])
```

Python fits the brain of many different kinds of people partly because it supports both procedural and object-oriented programming styles. In each module, you can define functions to implement behavior or create new objects by defining classes. These new objects can have methods and attributes, including special methods that teach Python how to interpret object-specific syntax (such as infix operators). Here's a simple class that inherits from the `list` object but redefines the `'+'`, `'-'`, and `'*'` infix operators to represent element-by-element addition, subtraction, and multiplication:

```
class vector(list):
    def __add__(self, other):
        res = [x+y for x,y in zip(self,
                                other)]
        return vector(res)
    def __sub__(self, other):
        res = [x-y for x,y in zip(self,
                                other)]
        return vector(res)
    def __mul__(self, other):
        res = [x*y for x,y in zip(self,
                                other)]
        return vector(res)
    def tolist(self):
        return list(self)
```

This class uses list comprehension (inline looping) and the `zip` built-in function to iterate over all elements of the two inputs that perform the re-

quested operation on each input pair. The result is a list from the list-comprehension syntax, which is converted to a vector before being returned:

```
>>> v = vector([1,2,3])
>>> print v+v
[2, 4, 6]
>>> print v*v
[1, 4, 9]
>>> print v-[3,2,1]
[-2, 0, 2]
```

Classes contain attributes accessed via dot notation (`object.attribute`), and methods are attributes that can be called like a function. They're defined inside a class block to take the object itself as the first argument. This object isn't needed when calling the method on an instance of the class, so to call the `tolist()` method of a vector class's object (which is bound to the name "v"), you would write `v.tolist()`.

Special methods are prefixed and postfixed with the characters `"_"` to indicate that the Python interpreter automatically calls them at special times—for example, `v+v` is translated by the interpreter to `v.__add__(v)`.

Standard Library

Another reason why Python is so useful is that it comes prepackaged with a wealth of general-purpose libraries, popularizing the notion that it comes with "batteries included." Let's review a few modules you might want to use in your own computing projects:

- `re`, a powerful regular expression matching that significantly enhances Python's already powerful string-processing capabilities;
- `datetime`, date-time objects and tools for their manipulation;
- `decimal`, support for real numbers with user-settable precision;
- `random`, functions for random-number generation;
- `pickle`, portable serialized (stringified) representations of Python objects (let's you save Python objects to disk and load them on a different system);
- `email`, routines for parsing, handling, and generating email messages;
- `csv`, assistance for automatically reading and writing files with comma-separated values;
- `gzip`, `zlib`, and `bz2`, functions for reading and writing compressed files;
- `zipfile` and `tarfile`, functions for extracting and creating file containers;

- `mmap`, an object allowing the usage of memory-mapped files;
- `urllib`, routines for opening arbitrary URLs;
- `ctypes`, a foreign-function interface that lets Python open a shared library and directly call its exported functions;
- `os`, a cross-platform interface to the functionality that most operating systems provide; and
- `sys`, objects maintained by the interpreter that interact strongly with it (such as the module-search path and the command-line arguments passed to the program).

All these libraries let you use Python right out of the box for almost all programming projects. Besides my scientific uses of Python, for example, I've also been able to use it to manage an email list, run a Web server, and create nice-looking personalized birthday calendars for my family.

Ease of Extension

Although Python is often fast enough for many calculation needs, multidimensional `for` loops will still leave you wanting some way to speed up the code. Fortunately, Python is easily extended with a large C-API for calling Python functionality from C, connecting to non-Python compiled code, and extending the language itself by creating new Python types (or classes) in C (or C++).

An extension module is a shared library that completely mimics a Python module with variable, function, and class attributes. It's created with a single entry-point function, which Python calls when the module is imported. This entry point sets up the module by adding constants, any new defined types, and the module function table, which translates between module function names and actual C functions to call.

Developers have created many automated tools over the years to make the process of writing Python modules that use compiled code almost trivial—`f2py` (<http://cens.ioc.ee/projects/f2py2e/>), for example, allows automated calling of Fortran code; `weave` (www.scipy.org/Weave) allows calling of C/C++ code; `Boost.Python` (www.boost.org/libs/python/doc/) allows seamless integration of C++ code into Python; and `Pyrex` (www.cosc.canterbury.ac.nz/greg.ewing/python/Pyrex/) allows the writing of an extension module in a Python-like language.

However, what really makes Python excel as a language for scientists and engineers is the NumPy extension (which you download separately). It supplies Python with a multidimensional array object whose items are stored exactly as they would be in compiled code. The extension also provides universal functions

that operate rapidly over the multidimensional array on an element-by-element basis as well as additional fast functions for calculations on the array. NumPy has its own C-API, so you can write your own extensions with it. Such extensions form the foundation of SciPy, which is a collection of toolboxes to further extend Python's scientific-computing capabilities.

NumPy

NumPy is freely available at <http://numpy.scipy.org> and offered under a very generous license. It grew out of an original module called Numeric (and sometimes also called `numpy`) written in 1995. Numeric established most of NumPy's features, but the way Numeric handled data types wasn't flexible enough for a group of scientists at the Space Science Telescope Institute, so they built a replacement system called Numarray that introduced significant new features. However, despite a degree of compatibility between the two array systems, developers wrote various extensions that could use only one package or the other, which created a divide in the fledgling community.

In 2005, I began the process of trying to bridge this divide by creating NumPy, which essentially took the ideas that Numarray had pioneered and ported them back to the Numeric code base, significantly enhancing Numeric in the process. In October 2006, we released version 1.0 of NumPy, which has all the features of Numeric and Numarray (including full support for both of their C-APIs) plus some additional features. This article barely scratches the surface of what NumPy provides, but a full account of the package is available at www.treglolo.com.

Array Objects

First and foremost, NumPy provides a homogeneous, multidimensional array of a particular data type. Although the array's main goal is to hold items of the same kind, one of its available built-in data types is called an "object." An array of it can hold an arbitrary Python object at each element, thus it's effectively a heterogeneous multidimensional array.

Data types. The data type an array can hold is quite arbitrary. NumPy's array internally supports all the fundamental C data types, including 10 different kinds of signed and unsigned integers, three kinds of floats, three kinds of complex numbers, and a Boolean type. In addition, arrays can hold strings or unicode strings, and you can even define your own data type that's equivalent to a C structure (sometimes called a *record*). Thus, for example, you can have an array whose elements consist of a 20-

byte record whose first field is a 4-byte integer, second field is a 12-character string, and last field is an array of four 1-byte unsigned integers. Consider the following example, which defines such a data type to track an array of students:

```
>>> import numpy as N
>>> dt = N.dtype([('id', 'i4'),
('name', 'S12'), ('scores', 'u1', 4)])
>>> a = N.array([(1001, 'James',
[100,98,97,60]), (1002, 'Kathy',
[100,100,85,98]), (1003, 'Michael',
[84,75, 98,100]), (1004, 'John',
[84,76,82,92])], dtype=dt)
>>> a['name']
array(['James', 'Kathy', 'Michael',
'John'], dtype='<S12')
>>> a['scores']
array([[100, 98, 97, 60],
[100, 100, 85, 98],
[ 84, 75, 98, 100],
[ 84, 76, 82, 92]], dtype=uint8)
```

This example shows how to extract a record array's fields as arrays of another data type. Records can even be nested so that a particular field itself contains another record data type. Record arrays are a useful way to group data and are essential if the array is constructed from a complicated memory-mapped data file.

Given the potential complexity of the array data type, it's useful to think about an array in NumPy as a collection of items consuming exactly the same number of bytes. A data type object describes each element in the array, whereas the array itself provides the information regarding the array's shape.

Attributes and methods. All arrays have several attributes and methods. Some attributes can be set to alter the array's characteristics—for example, we could reshape the previously created array of students into a 2×2 array by setting its `shape` attribute:

```
>>> print a.shape, a.ndim
(4,) 1
>>> a.shape = (2,-1)
>>> print a.shape, a.ndim
(2,2) 2
```

The `-1` entry in the shape-setting tuple indicates that the second dimension's shape should be whatever is necessary to use all the array elements. In this case, the missing entry is 2.

The array's methods allow quick computation or

manipulation of its elements—in our example, we can use them to *sort* the one-dimensional array of students. Using the `name` field's `argsort` method returns an array of indices that sort the array. Providing this set of indices to the `take` method creates a new 1D array in the sorted order:

```
>>> b = a.take(a['name'].argsort())
>>> print b['id']
[1001 1004 1002 1003]
```

Indexing. Another useful array feature is the ability to extract specific subregions via *indexing* by using the `a[obj]` notation on an array object. There are basically four kinds of array indexing: field extraction, element selection, view-based slicing, and copy-based indexing. We've already seen an exam-

What really makes Python excel as a language for scientists and engineers is the NumPy extension (which you download separately).

ple of indexing in which `obj` indicates which array field to extract; element selection occurs when `obj` is such that we extract a single element of the array. In this case, the indexing notation returns a new Python scalar object that contains the data at that location in the array. For example, let's define `a` to be an array of Gaussian random numbers and extract a particular number from it:

```
>>> import numpy as N
>>> a = N.random.randn(50,25)
>>> print a.shape, a[10,15]
(50, 25) 0.5295135653
```

View-based indexing occurs when `obj` is a slice object or a tuple of slice objects. In this case, the indexing notation returns a new array object that points to the same data as the original array. This is an important optimization that saves unnecessary memory copying but must be mentally noted to avoid unexpected data alteration. Given the previously defined array, for example,

```
>>> b=a[10:15:2, 8:13:2]; b
array([[ 0.35238367, -0.40288084,
 0.10110947],
[-0.91742114,  1.13308636,  0.00602061],
[-0.57394525, -2.00959791, -0.3262831
]])
```

The notation `10:15:2` inside the indexing brackets tells Python to begin at element 10, end before element 15, and get every two elements—in other words, to extract elements 10, 12, and 14 from the first dimension of the array. Indexing along one dimension of a 2D array extracts a 1D array. As a result, the second range indicates that for each 1D array, we should take elements 8, 10, and 12 to form the output array. Remember this output array is just a view of the underlying data—changing elements of `b` will change the newly formed array as well.

Copy-based indexing always returns a copy of the data and occurs when `obj` is an array (or a tuple containing arrays) of a Boolean or integer data type. Boolean indexing allows masked selection of all the array's elements, which is particularly useful for setting certain elements to a particular value. The code `a[a > 0.1] -= 0.1`, for example, will decrease every element of `a` that's larger than 0.1 by 0.1. This works because arrays redefine subtraction to perform element-by-element subtraction and comparison operators to return Boolean arrays with the comparison implemented element by element.

When the indexing array has an integer data type, it's used to extract an array of specific entries. For fully specified index arrays, the returned array's shape is the same as the shape of the input indexing arrays—for example,

```
>>> b=a[ [10,12,14], [13,15,17] ]; b
array([ 1.55922631, 0.93609952,
       -0.10149853])
```

Notice that the returned 1D array has elements `a[10,13]`, `a[12,15]`, and `a[14,17]` and not the cross-product array, as some would expect. We can get the cross-product by using either `b=a[[10],[12],[14]],[13,15,17]]` or `b=a[N.ix_([10,12,14],[13,15,17])]`.

Universal Functions

Exceptional opportunities for array manipulation and extraction are only part of what makes NumPy useful for scientific computing. It also provides universal function objects (`ufuncs`), which make it simple to define functions that take N inputs and return M outputs by performing some underlying function, element by element. The basic mathematical operations of arrays are all implemented with universal functions that take two inputs and return one output—for example, when either `b` or `c` is an array object, `a=b+c` is equivalent to `a=N.add(b,c)`.

More than 50 mathematical functions are defined in the `numpy` module as universal functions, but it's easy to define your own universal functions either in compiled code (for very fast `ufuncs`) or based on a Python function (which will have `ufuncs` features but will operate more slowly). Let's look more closely at these features.

Broadcasting. `ufuncs` operate element by element, which seems to imply that all input arrays must have the same shape. Fortunately, NumPy provides a concept known as *broadcasting*, which is a specific method for arrays that don't have the same shape to try and act as if they do. Broadcasting has two rules in the form of steps:

- Make sure all arrays have the same number of dimensions by pre-pending a 1 to the shape attribute of arrays whose number of dimensions is too small. Thus, if an array of shape $(2, 5)$ and an array of shape $(5,)$ were input into a `ufunc`, the $(5,)$ -shaped array would be interpreted as a $(1, 5)$ -shaped array.
- Interpret the length of any axis whose length is 1 as if it were the size of the other non-unit-length arrays in the operation. Thus, if we use an array of shape $(3, 6)$ and an array of shape $(6,)$, the second array would first be interpreted as a $(1, 6)$ -shaped array and then as a $(3, 6)$ -shaped array.

If applying these rules fails to produce arrays of exactly the same shape, an error occurs because element-by-element operation isn't defined. As an example of `ufunc` behavior, consider the following code, which computes the outer product of two 1D arrays:

```
>>> a,b = N.array([1,2,3],[10,20,30])
>>> c = a[:,N.newaxis] * b; print c
[[10, 20, 30],
 [20, 40, 60],
 [30, 60, 90]]
```

The first line uses Python's ability to map a sequence object to multiple objects in one line, which is equivalent to `temp=N.array(...)` followed by `a=temp[0]` and `b=temp[1]`. This produces two 1D arrays of shape $(3,)$. The indexing manipulation in the next line selects all the elements of `a` and adds a new axis to its end, making a $(3, 1)$ -shaped array. Broadcasting then interprets the multiplication of a $(3,)$ -shaped array as multiplication by a $(1, 3)$ -shaped array that produces a $(3, 3)$ -shaped result, which is $c_{ij} = a_i b_j$ in index notation. Note that broadcasting never copies any data to perform its

dimension upgrading—rather, copying is handled by reusing the repeated values internally.

Output arrays. All `ufuncs` take optional arguments for output arrays. Sometimes, to speed up calculations, you might want the `ufunc` to place its result in an already allocated array rather than have a fresh new allocation occur. This is especially true in a complicated calculation that has many temporaries—for example, the code `a=(b+4)*c*d` involves the creation of two strictly unnecessary temporary arrays to hold the results of intermediate calculations. If these arrays are large, we can save the significant overhead of creating and deleting the temporary arrays by writing this as

```
a=b+4
N.multiply(a, c, a)
N.multiply(a, d, a)
```

Clearly, this isn't as easy to read, but it could be essential for large simulations.

Memory-saving type conversion. When `ufuncs` are created, they must specify the data types of the required inputs and outputs. A typical `ufunc` usually has several low-level routines registered to match specific signatures. If the pattern of inputs doesn't match one of the internally supported signatures, the `ufunc` machinery will upcast any inputs (in size-limited chunks) as needed to match an available signature so the calculation can proceed. This same chunked-casting occurs if an output array is provided that isn't the same type the calculation produces.

Suppose `a` is an array of 4-byte integers, but `b` is an array of 4-byte floats, and you want to add them together. The `add` `ufunc` has 18 registered low-level functions for implementing the `add` operation on identical input data types that produce the same data type. For this mixed data-type operation, the `ufunc` machinery chooses 8-byte floats for the underlying operation (the “lowest” type to which we can convert both 4-byte integers and 4-byte floats without losing precision). The conversion to 8-byte floats occurs behind the scenes in chunks no larger than a user-settable buffer size. If an output array is provided, then the 8-byte result is coerced back into the output.

Array-like object wrapping. All `ufuncs` provide the ability to handle any object as an input. The only requirement is that it be converted to an array (because it has an `__array__` method, defines the array interface, or is a sequence object). If the input arrays also have the special `__array_wrap__`

method defined, then that method will be called on each result of the `ufunc`. The method call's output is returned as the `ufunc`'s output. This lets user-defined objects define an `__array_wrap__` method that takes an input array and returns the user-defined object and have those objects passed seamlessly through NumPy's `ufuncs`.

Hardware error handling. On platforms that support it, NumPy lets you query the result of hardware error flags during the computation of any `ufunc` and issue a warning, issue an error, or call a user-provided function. Hardware error flags include underflow, overflow, divide-by-zero, and invalid result. By default, all errors either give a warning or are ignored, but the `seterr` function lets you alter that behavior:

```
>>> array([1,2])/0. # emits a warning
>>> old = N.seterr(divide='raise')
>>> array([1,2])/0. # now it will raise
an error
>>> N.seterr(**old)
```

Methods. All `ufuncs` that take two inputs and return one output can also use the `ufunc` methods `reduce`, `accumulate`, and `reduceat`, which, respectively

- perform the function repeatedly using successive elements along a particular dimension of the array and then store the result in an output variable;
- perform the function repeatedly using successive elements along a particular dimension of the array and then store each intermediate result; and
- perform the function repeatedly using successive elements along specific portions of a particular dimension of the array.

These methods admit simple interpretations for well-known operations—for example, the `sum` method of an array that sums all the elements along a particular axis is implemented using `add.reduce`. Similarly, the array's `prod` method is implemented with `multiply.reduce`.

Basic Libraries

In addition to the fundamental array object and the many implemented `ufuncs`, NumPy comes with several standard subclasses, a masked-array implementation, a simple polynomial class, set operations on 1D arrays, and a host of other useful functions. Routines for implementing Fourier transforms, basic linear algebra operations, and random-number generation are also available. Let's

look at these features more closely, starting with the provided objects:

- `matrix` is an array subclass that works with 2D arrays and redefines the `'*'` operator to be matrix-multiplication and the `'**'` operator to be matrix-power;
- `memmap` is an array subclass in which the memory is a memory-mapped file;
- `recarray` is an array subclass that allows field access by using attribute lookup and returning arrays of strings as `chararrays`;
- `chararray` is an array of strings that provide standard string methods (which operate element-by-element) as additional array methods; and
- `ma` is an additional array type called a *masked array*, which stores a mask along with each array to ignore specific values during computation.

Many functions exist for creating, manipulating, and operating on arrays, but let's focus here on a small sampling of commonly used functions:

- `convolve` curls together two 1D input sequences;
- `diag` constructs a 2D array from its diagonal or extract a diagonal from a 2D array;
- `histogram` creates a histogram from data;
- `choose` constructs an array using a choice array and additional inputs;
- `dot` sums over the last dimension of the first argument and the second-to-last dimension of the second (this extension of matrix-multiplication exploits system libraries if available at compile time);
- `empty`, `zeros`, and `ones` create arrays of a certain shape and data type filled with nothing, 0, and 1, respectively;
- `fromfunction` creates an N -d array from a function with N inputs, which are assumed to be index positions in the array;
- `vectorize` takes a callable object (such as a function or a method) that operates on scalar inputs and return a callable object that works on arbitrary array inputs element-by-element; and
- `lexsort` returns an array of indices showing how to sort a sequence of arguments (these arguments must all be arrays of the same shape).

The Fourier transform is an essential tool to many algorithms for arbitrary-dimensioned data. The `numpy.fft` package has the following routines:

- `fft` and `ifft`, 1D fast Fourier transform and inverse (uses $1/N$ normalization on inverse);
- `fft2` and `ifft2`, 2D fast Fourier transform and inverse; and

- `fftn` and `ifftn`, ND fast Fourier transform and inverse.

Additional routines specialize for real-valued data—for example, `rfft` and `irfft` are real-valued Fourier transforms that take real-valued inputs and return nonredundant complex-valued outputs by taking 1D Fourier transforms along a specified dimension. All the `fft` routines take N -dimensional arrays, can pad to specified lengths, and operate only along the axes specified.

Linear algebra routines (they're all under the `numpy.linalg` namespace) are built against a default basic linear algebra system (BLAS) and a stripped-down linear algebra package (LAPack) implementation (vendor-specific linear algebra libraries can be used if provided at build-time). Among the numerical routines available are those for finding a matrix inverse (`inv`), solving a system of linear equations (`solve`), finding the determinant of a matrix (`det`), finding eigenvalues and eigenvectors (`eig`), and finding the pseudo inverse of a matrix (`pinv`). Routines for the Cholesky (`cholesky`), QR (`qr`), and SVD (`svd`) decompositions are also available.

Random-number generation is an important part of most scientific computing exercises, so NumPy comes equipped with a substantial list of fast random-number generators of both continuous and discrete type. All these random-number generators reside in the `numpy.random` namespace, allow for array inputs, and produce array outputs. Each random-number generator also takes a `size=keyword` argument that specifies the shape of the output array to be created. Two convenience functions, `rand` and `randn`, produce standard uniform and standard normal random numbers using their input arguments to determine the output shape:

```
>>> a = N.random.rand(5,10,20); print
a.shape, a.std()
(5, 10, 20) 0.289855415313
>>> b = N.random.randn(6,12,22); print
b.shape, b.var()
(6, 12, 22) 1.01300101504
```

Additional random-number generators are available for producing variates from roughly 50 different distributions; some of the continuous distributions include exponential, chi-square, Gumbel, multivariate normal, noncentral F, triangular, and gamma, to name a few. The discrete distributions available include binomial, geometric, hypergeometric, Poisson, and multinomial.

f2py

As I mentioned earlier, NumPy comes installed with a powerful tool called f2py, which can parse Fortran files and construct an extension module that contains all the subroutines and functions in those files as methods. Suppose I have a file called `example.f` that has two simple Fortran routines for subtracting the element of one array from another in two different precisions:

```
SUBROUTINE DSUB(A,B,C,N)
  DOUBLE PRECISION A(N)
  DOUBLE PRECISION B(N)
  DOUBLE PRECISION C(N)
  INTEGER N
CF2PY INTENT(OUT) :: C
CF2PY INTENT(HIDE) :: N
  DO 20 J = 1, N
    C(J) = A(J) + B(J)
  20 CONTINUE
```

With equivalent code called `SSUB` that uses `REAL` instead of `DOUBLE PRECISION`, I can make and compile an extension module (called `example`) that contains two functions (`ssub` and `dsub`). All I need to do is run

```
f2py -m example -c example.f
```

The `CF2PY` directives in the code (which are interpreted as Fortran comments) make the interface to both routines receive two input arguments and return one output argument. The argument providing the array size is hidden and passed along automatically:

```
>>> import example
>>> example.dadd([1,2,3],[4,5,6])
array([5., 7., 9.])
>>> example.sadd([1,2,3],[4,5,6])
array([5., 7., 9.], dtype=float32)
```

Notice that f2py converts input arrays and returns the output in the expected precision.

SciPy


Quite a bit of calculation and computational ability exists with just Python and the NumPy package installed, but if you're accustomed to other computational environments, you might notice a few missing tools, such as those for optimization, special functions, and image processing. SciPy builds on top of NumPy to provide such advanced tools. To do this, SciPy resurrects quite a bit of the well-tested code available at public-domain repositories such as netlib:

- The *input/output* (`io`) subpackage provides raw routines for reading and writing binary files as well as simplified routines for reading and writing files for popular data formats.
- The *linear algebra* (`linalg`) subpackage provides extended interfaces to the BLAS and LAPack libraries and has additional decompositions such as LU (`lu`) and Schur (`schur`), as well as a selection of matrix functions such as matrix exponential (`expm`), matrix square root (`sqrtm`), and matrix logarithm (`logm`).
- The *statistics* (`stats`) subpackage provides a wide variety of distribution objects for not only creating random variates but also evaluating the `pdf`, `cdf`, and inverse `cdf` of many continuous and discrete distributions.
- The *optimization* (`optimize`) subpackage provides a collection of constrained and unconstrained multivariate optimizers and function solvers.
- The *integration* (`integrate`) subpackage provides tools for integrating both functions and ordinary differential equations, including a general-purpose integrator (`quad`), a Gaussian quadrature integrator (`quadrature`), and a method that uses Romberg interpolation (`romberg`).
- The *interpolation* (`interpolate`) subpackage includes cubic splines and linear interpolation in several dimensions.
- *Weave* (`weave`) is a very useful module for calling inline C code from Python, but it's also helpful for building extension modules (by just writing the actual C code that implements the functionality).
- The *Fourier transforms* (`fftpack`) subpackage provides Fourier transforms implemented using a different wrapper to the `fftpack` library for single and double precision as well as Hilbert and inverse Hilbert transforms.
- The *special functions* (`special`) subpackage provides more than 250 special-function calculation engines, most of which are available as universal functions (`ufuncs`).
- The *sparse* (`sparse`, `linsolve`) subpackage provides sparse matrices in several different storage schemes as well as direct and iterative solution schemes.
- The *Nd-image* (`ndimage`) subpackage provides a large collection of image- and array-processing capabilities for *N*-dimensional arrays, including fast B-spline interpolation, morphology, and various filtering operations.
- The *signals and systems* (`signal`) subpackage provides routines for signal and image processing, including *N*-dimensional convolution,

fast B-spline functions, order filtering, median filtering 1D finite impulse response (FIR) and infinite impulse response linear filtering, filter design techniques, waveform generation, and various linear time invariant (LTI) system functions.

- The *maximum entropy models* (`maxentropy`) subpackage contains two classes for fitting maximum entropy models subject to linear constraints on the expectations of arbitrary feature statistics (one class is for small discrete sample spaces, whereas the other is for sample spaces that are too large to sum over).
- The *clustering* (`cluster`) subpackage contains an implementation of the K-means clustering algorithm for generating a smaller set of observations that best fit a larger set.

Naturally, this list covers only the bare bones of what SciPy's subpackages can do—you can find more information at www.scipy.org. In addition, you can use Python's `help` command on the SciPy package and all of its subpackages (using `help(scipy.<name>)`) in an interactive session once `import scipy` has been executed.

Due to space constraints, I've barely explained all the features that Python provides to the practitioner of scientific computing—for example, I've only hinted at the myriad tools available for making it easy to wrap compiled code. Likewise, I haven't really discussed NumPy's extensive C-API, which helps you build extension modules. You can glean information about these and much more by going to <http://numpy.scipy.org> or www.scipy.org. For general Python information, visit www.python.org—it has additional tools not necessarily integrated into NumPy or SciPy for doing computational work. Hopefully, you'll investigate and see how much easier and more efficient your daily computational work will become thanks to this powerful language. 

Travis E. Oliphant is an assistant professor of electrical and computer engineering at Brigham Young University. He's a principal author of both SciPy and NumPy, and his research interests include microscale impedance imaging, MRI reconstruction in inhomogeneous fields, and any biomedical inverse problem. Oliphant has a PhD in biomedical engineering from the Mayo Graduate School. Contact him at oliphant@ee.byu.edu.

CISE Computational Physics Challenge for Undergraduate Physics Students

For further information go to www.siena.edu/physics/

Entrants will pick their own problems.

Judging will be based on the following 50-point scoring system:

1. A well-defined and interesting problem (10)
2. A description of the model including the algorithm (5)
3. An explanation of the numerical methods used (5)
4. The code (5)
5. The results (10)
6. A discussion of the results in the light of the problem being explored (10)
7. A critique of the work (5)

Sponsored by

computing
in SCIENCE & ENGINEERING

SHODOR
A NATIONAL RESOURCE FOR
COMPUTATIONAL SCIENCE EDUCATION

SIENA college
SCIENCE

Deadline for submission is 15 May 2007

CASH PRIZES FOR THE WINNERS

IPython: A System for Interactive Scientific Computing

Python offers basic facilities for interactive work and a comprehensive library on top of which more sophisticated systems can be built. The IPython project provides an enhanced interactive environment that includes, among other features, support for data visualization and facilities for distributed and parallel computation.

The backbone of scientific computing is mostly a collection of high-performance code written in Fortran, C, and C++ that typically runs in batch mode on large systems, clusters, and supercomputers. However, over the past decade, high-level environments that integrate easy-to-use interpreted languages, comprehensive numerical libraries, and visualization facilities have become extremely popular in this field. As hardware becomes faster, the critical bottleneck in scientific computing isn't always the computer's processing time; the scientist's time is also a consideration. For this reason, systems that allow rapid algorithmic exploration, data analysis, and visualization have become a staple of daily scientific work. The Interactive Data Language (IDL) and Matlab (for numerical work), and Mathematica and Maple (for work that includes symbolic manipulation) are well-known commercial environments of this kind. GNU Data Language, Octave, Maxima and Sage provide their open source counterparts.

All these systems offer an interactive command line in which code can be run immediately, without having to go through the traditional edit/compile/execute cycle. This flexible style matches well the spirit of computing in a scientific context, in which determining what computations must be performed next often requires significant work. An interactive environment lets scientists look at data, test new ideas, combine algorithmic approaches, and evaluate their outcomes directly. This process might lead to a final result, or it might clarify how to build a more static, large-scale production code.

As this article shows, Python (www.python.org) is an excellent tool for such a workflow.¹ The IPython project (<http://ipython.scipy.org>) aims to not only provide a greatly enhanced Python shell but also facilities for interactive distributed and parallel computing, as well as a comprehensive set of tools for building special-purpose interactive environments for scientific computing.

Python: An Open and General-Purpose Environment

The fragment in Figure 1 shows the default interactive Python shell, including a computation with long integers (whose size is limited only by the available memory) and one using the built-in complex numbers, where the literal `1j` represents $i = \sqrt{-1}$.

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

FERNANDO PÉREZ

University of Colorado at Boulder

BRIAN E. GRANGER

Tech-X Corporation

```

$ python # $ represents the system prompt
Python 2.4.3 (Apr 27 2006, 14:43:58)
[GCC 4.0.3 (Ubuntu 4.0.3-1ubuntu5)] on linux2
Type "help", "copyright", "credits" or "license"
for more information.
>>> print "This is the Python shell."
This is the Python shell.
>>> 2**45+1 # long integers are built-in
35184372088833L
>>> import cmath # default complex math library
>>> cmath.exp(-1j*cmath.pi)
(-1-1.2246063538223773e-16j)

```

Figure 1. Default interactive Python shell. In the two computations shown—one with long integers and one using the built-in complex numbers—the literal $1j$ represents $i = \sqrt{-1}$.

This shell allows for some customization and access to help and documentation, but overall it's a fairly basic environment.

However, what Python lacks in the sophistication of its default shell, it makes up for by being a general-purpose programming language with access to a large set of libraries with additional capabilities. Python's standard library includes modules for regular expression processing, low-level networking, XML parsing, Web services, object serialization, and more. In addition, hundreds of third-party Python modules let users do everything from work with Hierarchical Data Format 5 (HDF5) files to write graphical applications. These diverse libraries make it possible to build sophisticated interactive environments in Python without having to implement everything from scratch.

IPython

Since late 2001, the IPython project has provided tools to extend Python's interactive capabilities beyond those shipped by default with the language, and it continues to be developed as a base layer for new interactive environments. IPython is freely available under the terms of the BSD license and runs under Linux and other Unix-type operating systems, Apple OS X, and Microsoft Windows.

We won't discuss IPython's features in detail here—it ships with a comprehensive user manual (also accessible on its Web site). Instead, we highlight some of the basic ideas behind its design and how they enable efficient interactive scientific computing. We encourage interested readers to visit the Web site and participate on the project's mailing lists.

One of us (Fernando Pérez) started IPython as a merger of some personal enhancements to the basic interactive Python shell with two existing open source projects (both now defunct and subsumed into IPython):

- LazyPython, developed by Nathan Gray at Caltech, and
- Interactive Python Prompt (IPP) by Janko Hauser at the University of Kiel's Institute of Marine Research.

After an initial development period as a mostly single-author project, IPython has attracted a growing group of contributors. Today, Ville Vainio and other collaborators maintain the stable official branch, while we're developing a next-generation system.

Since IPython's beginning, we've tried to provide the best possible interactive environment for everyday computing tasks, whether the actual work was scientific or not. With this goal in mind, we've freely mixed new ideas with existing ones from Unix system shells and environments such as Mathematica and IDL.

Features of a Good Interactive Computing Environment

In addition to providing direct access to the underlying language (in our case, Python), we consider a few basic principles to be the minimum requirements for a productive interactive computing system.

Access to all session state. When working interactively, scientists commonly perform hundreds of computations in sequence and often might need to reuse a previous result. The standard Python shell remembers the very last output and stores it into a variable named “_” (a single underscore), but each new result overwrites this variable. IPython stores a session's inputs and outputs into a pair of numbered tables called `In` and `Out`. All outputs are also accessible as `_N`, where `N` is the number of results (you can also save a session's inputs and outputs to a log file). Figure 2 shows the use of previous results in an IPython session. Because keeping a very large set of previous results can potentially lead to memory exhaustion, IPython lets users limit how many results are kept. Users can also manually delete individual references using the standard Python `del` keyword.

A control system. It's important to have a secondary control mechanism that is reasonably orthogonal

to the underlying language being executed (and independent of any variables or keywords in the language). Even programming languages as compact as Python have a syntax that requires parentheses, brackets, and so on, and thus aren't the most convenient for interactive control systems.

IPython offers a set of control commands (or *magic commands*, as inherited from IPP) designed to improve Python's usability in an interactive context. The traditional Unix shell largely inspires the syntax for these magic commands, with white space used as a separator and dashes indicating options. This system is accessible to the user, who can extend it with new commands as desired.

The fragment in Figure 3 shows how to activate IPython's logging system to save the session to a named file, requesting that the output is logged and every entry is time stamped. IPython automatically interprets the `logstart` name as a call to a magic command because no Python variable with that name currently exists. If there were such a variable, typing `%logstart` would disambiguate the names.

Operating system access. Many computing tasks involve working with the underlying operating system (reading data, looking for code to execute, loading other programs, and so on). IPython lets users create their own aliases for common system tasks, navigate the file system with familiar commands such as `cd` and `ls`, and prefix any command with `!` for direct execution by the underlying OS. Although these are fairly simple features, in practice they help maintain a fluid work experience—for example, they let users type standard Python code for programming tasks and perform common OS-level actions with a familiar Unix-like syntax. IPython goes beyond this, letting users call system commands with values computed from Python variables. These features have led some users (especially under Windows, a platform with a very primitive system shell) to use IPython as their default shell for everyday work.

Figure 4 shows how to perform the simple task of normalizing the names of a few files to a different convention.

Dynamic introspection and help. One benefit of working interactively is being able to directly manipulate code and objects as they exist in the runtime environment. Python offers an interactive help system and exposes a wide array of introspective capabilities as a standard module (`inspect.py`) that provides functions for exploring various types of objects in the language.

```
$ ipython
Python 2.4.3 (Apr 27 2006, 14:43:58)
Type "copyright", "credits" or "license" for more
information.

IPython 0.7.3 -- An enhanced Interactive Python.
?          -> Introduction to IPython features.
%magic     -> Information about IPython magic %
functions.
Help       -> Python help system.
object?    -> Details about object. ?object also
works, ?? prints more.
In [1]:2**45+1
Out[1]:35184372088833L
In [2]:import cmath
In [3]:cmath.exp(-1j*cmath.pi)
Out[3]:(-1-1.2246063538223773e-16j)
# The last result is always stored as '_'
In [4]:_ ** 2
Out[4]:(1+2.4492127076447545e-16j)
# And all results are stored as N, where _N is
their number:
In [5]:_3+_4
Out[5]:1.2246063538223773e-16j
```

Figure 2. The use of previous results in an IPython session. In IPython, all outputs are also accessible as `_N`, where `N` is the number of results.

```
In [2]: logstart -o -t ipsession.log
Activating auto-logging. Current session state
plus future input saved.
Filename           : ipsession.log
Mode               : backup
Output logging     : True
Raw input log      : False
Timestamping       : True
State              : active
```

Figure 3. Activating IPython's logging system to save the session to a named file. IPython interprets the `logstart` name as a call to a control command (or *magic command*).

IPython offers access to Python's help system, the ability to complete any object's names and attributes with the Tab key, and a system to query an object for internal details, including source code,

```

In [36]: ls
tt0.dat tt1.DAT tt2.dat tt3.DAT
# 'var = !cmd' captures a system command into a
Python variable:
In [37]: files = !ls
==
['tt0.dat', 'tt1.DAT', 'tt2.dat', 'tt3.DAT']
# Rename the files, using uniform case and 3-digit
numbers:
In [38]: for i, name in enumerate(files):
....:     newname = 'time%03d.dat' % i
....:     !mv $name $newname
....:
In [39]: ls
time000.dat time001.dat time002.dat time003.dat

```

Figure 4. Normalizing file names to a different convention. These code fragments show how IPython allows users to combine normal Python syntax with direct system calls (prefixed with the “!” character). In such calls, Python variables can be expanded by prefixing them with “\$.”

```

In [1]: from universe import DeepThought
In [2]: DeepThought. # Hit the Tab key here
        DeepThought._doc_ DeepThought.answer
        DeepThought.question
        DeepThought._module_ DeepThought.name
In [2]: DeepThought??
Type:          classobj
String Form:   universe.DeepThought
Namespace:     Interactive
File:          /tmp/universe.py
Source:
class DeepThought:
    name = "Deep Thought"
    question = None
    def answer(self):
        """Return the Answer to The Ultimate
Question Of Life, the Universe and Everything"""
        return 42

```

Figure 5. Information returned by IPython after querying an object called `DeepThought` from a module called `universe`. When the user hits the Tab key (line 2), IPython lists all attributes defined for `DeepThought`. For the sequence `DeepThought??`, IPython finds as much information about the object as it can, including its source code.

by typing the object’s name and one or two “?” (two for extra details). These features are useful when developing code, exploring a problem, or using an unfamiliar library because direct experimentation with the system can help produce working code that the user can then copy into an editor as part of a larger program.

Figure 5 shows the information returned by IPython after querying an object called `DeepThought` from a module called `universe`. In line 2, we’ve hit the Tab key, so IPython completes a list of all the attributes defined for `DeepThought`. Then, for the sequence `DeepThought??`, IPython tries to find as much information about the object as it can, including its entire source code.

Access to program execution. Although typing code interactively is convenient, large programs are written in text editors for significant computations. IPython’s `%run` magic command lets users run any Python file within the IPython session as if they had typed it interactively. Upon completion, the program results update the interactive session, so the user can further explore any quantity computed by the program, plot it, and so on. The `%run` command has several options to assist in debugging, profiling, and more. It’s probably the most commonly used magic function in a typical workflow: you use a text editor for significant editing while code is executed (using `run`) in the IPython session for debugging and results analysis. Typing `run?` provides full details about the `run` command.

Figure 6 compares IPython to the default Python shell when running a program that contains errors. IPython provides detailed exception tracebacks with information about variable values, and can activate a debugger (indicated by the `ipdb>` prompt), from which a user can perform postmortem analysis of the crashed code from its in-memory state, walk up the call stack, print variables, and so on. This mechanism saves time during development, because the user doesn’t need to reload libraries used by a program for each new test. It also lets the user perform expensive initialization steps only once, keeping them in memory while the user explores other parts of a problem by making changes to code and running it repeatedly.

A Base Layer for Interactive Environments

In addition to these minimal requirements, IPython exposes its major components to the user for modification and customization, making it a

flexible and open platform. Other scientific computing projects have used IPython's features to build custom interactive environments. A user can declare these customizations in a plaintext file—an *IPython profile*—and load them using the `-profile` flag at startup time.

Input syntax processing. Underlying IPython is a running Python interpreter, so ultimately all code executed by IPython must be valid Python code. However, in some situations the user might want to allow other forms of input that aren't necessarily Python. Such uses can range from simple transformations for input convenience to supporting a legacy system with its own syntax within the IPython-based environment.

As a simple example, IPython ships with a physics profile, which preloads physical unit support from the ScientificPython library (<http://sourceup.cru.fr/projects/scientific-py>), and installs a special input filter. This filter recognizes text sequences that appear to be quantities with units and generates the underlying Python code to define an object with units, without the user having to type out the more verbose syntax, as Figure 7 shows.

IPython exposes the input filtering system, which users can customize to define arbitrary input transformations that might suit their problem domains. For example, the Software for Algebra and Geometry Experimentation (Sage)² project uses an input filter to transform numerical quantities into exact integers, rationals, and arbitrary precision floats instead of Python's normal numerical types. (See the "Projects Using IPython" sidebar for a description of this and other examples.)

Error handling. A common need in interactive environments is to process certain errors in a special manner. IPython offers three exception handlers that treat errors uniformly, differing only in the amount of detail they provide. A custom environment might want to handle internal errors, or errors related to certain special objects, differently from other normal Python errors. IPython lets users register exception handlers that will fire when an exception of their registered type is raised. Python's uniform and object-oriented approach to errors greatly facilitates this feature's implementation: because all exceptions are classes, users can register handlers based on a point in the class hierarchy that will handle any exception that inherits from the registered class. The PyRAF interactive environment at the Space Telescope Science Institute has used this capability to handle its own in-

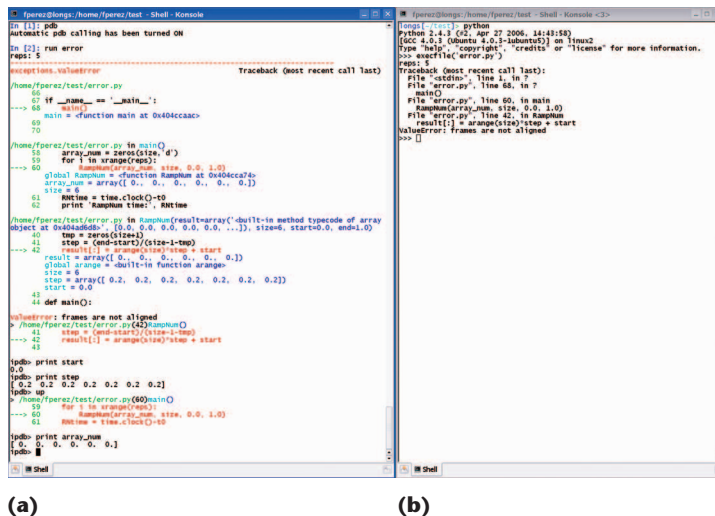


Figure 6. Comparison of IPython to the default Python shell.
(a) IPython provides detailed error information and can automatically activate an interactive debugger to inspect the crashed code's status, print variables, navigate the stack, and so on.
(b) The same error displayed in the default Python shell.

```
In [1]: mass = 3 kg
In [2]: g = 9.8 m/s^2
In [3]: weight=mass*g
In [4]: weight
Out[4]: 29.4 m*kg/s^2
# We can see the actual Python code generated by
IPython:
In [5]: %history # %history is an IPython "magic"
command
1: mass = PhysicalQuantityInteractive(3, 'kg')
2: g = PhysicalQuantityInteractive(9.8, 'm/s**2')
3: weight=mass*g
4: weight
```

Figure 7. Code using IPython's physics profile and input filter. The filter recognizes text sequences that appear to be quantities with units and generates the underlying Python code to define an object with units.

ternal errors separately from errors that are meaningful to the user.

Tab completion. Tab completion is a simple but useful feature in an interactive environment because the system completes not only on Python variables but also on keywords, aliases, magic commands, files, and directories. IPython lets users register new completers to explore certain objects.

PROJECTS USING IPYTHON

Several scientific projects have exploited IPython as a platform rather than as an end-user application. Although the vast majority of IPython users do little customization beyond setting a few personal options, these projects show that there is a real use case for open, customizable interactive environments in scientific computing:

- Sage (<http://modular.math.washington.edu/sage>), a system for mathematical research and teaching with a focus on algebra, geometry, and number theory, uses IPython for its interactive terminal-based interface.
- The Space Telescope Science Institute's PyRAF environment (www.stsci.edu/resources/software_hardware/pyraf) uses IPython for astronomical image analysis. PyRAF provides an IPython-based shell for interactive work with several special-purpose customizations. We made numerous enhancements to IPython based on requests and suggestions from the PyRAF team.
- The National Radio Astronomy Observatory's Common Astronomy Software Applications (CASA, <http://casa.nrao.edu>) uses IPython in its interactive shell.

- The Ganga system (<http://ganga.web.cern.ch/ganga/>), developed at the European Center for Nuclear Research (CERN) for grid job control for the large hadron collider beauty experiment (LHCb) and Atlas experiments, uses IPython for its command-line interface (CLIP).
- The PyMAD project (<http://ipython.scipy.org/moin/PyMAD>) uses IPython to control a neutron spectrometer at CEA-Grenoble and the Institute Laue Langevin in France.
- The Pymerase project (<http://pymerase.sourceforge.net>) for microarray gene expression databases exposes an IPython shell in its interactive iPymerase mode.

Based on the lessons learned from this usage, we're currently restructuring IPython to allow interactive parallel and distributed computing, to build better user interfaces, and to provide more flexible and powerful components for other projects to build on. We hope that if more projects are developed on top of such a common platform, all users will benefit from the familiarity of having a well-known base layer on top of which their specific projects add custom behavior.

The PyMAD project at the neutron scattering facility of the Institute Laue Langevin in Grenoble, France, uses this feature for interactive control of experimental devices. The IPython console runs on a system that connects to the neutron spectrometer over a network, but users interact with the remote system as if it were local, and Tab completion operates over the network to fetch information about remote objects for display in the user's local console.

Graphical Interface Toolkits and Plotting

Python provides excellent support for GUI toolkits. It ships by default with bindings for Tk, and third-party bindings are available for GTK, WxWidgets, Qt, and Cocoa (under Apple OS X). You can use essentially every major toolkit to write graphical applications from Python. Although few scientists look forward to GUI design, they increasingly have to write small- to medium-sized graphical applications to interface with scientific code, drive instruments, or collect data. Python lets scientists choose the toolkit that best fits their needs.

However, graphical applications are notoriously difficult to test and control from an interactive command line. In the default Python shell, if a user instantiates a Qt application, for example, the command line stops responding as soon as the Qt win-

dow appears. IPython addresses this problem by offering special startup flags that let users choose which toolkit they want to control interactively in a nonblocking manner.

This feature is necessary for one of scientists' most common tasks: interactive data plotting and visualization. Many traditional plotting libraries and programs have Python bindings or process-based interfaces, but most have various limitations for interactive use. The matplotlib project (<http://matplotlib.sourceforge.net>) is a sophisticated plotting library capable of producing publication-quality graphics in a variety of formats, and with full LaTeX support.³ Matplotlib renders its plots to several back ends, the components responsible for generating the actual figure. Some back ends (such as for PostScript, PDF, and Scalable Vector Graphics) are purely disk-based and meant to generate files; others are meant for display in a window. Matplotlib supports all these toolkits, letting users choose which to use via a configuration file setting. (The Scientific Programming department on p. 90 explores matplotlib in more detail.)

IPython and matplotlib developers have collaborated to enable automatic coordination between the two systems. If given the special `-pylab` startup flag, for example, IPython detects the user's matplotlib settings and automatically configures itself to enable nonblocking interactive plotting. This

provides an environment in which users can perform interactive plotting in a manner similar to Matlab or IDL but with complete flexibility in the GUI toolkit used (these programs provide their own GUI support and can't be integrated in the same process with other toolkits).

In the example in Figure 8, plots are generated from an interactive session using matplotlib. We use the special function and numerical integration routines provided by the SciPy package⁴ to verify, at a few points, the standard relation for the first Bessel function

$$J_0(x) = \frac{1}{\pi} \int_0^{\pi} \cos(x \sin \phi) d\phi.$$

The last line shows matplotlib's capabilities for array plotting with a simple 32×32 set of random numbers.

Although matplotlib's main focus is 2D plotting, several packages exist for 3D plotting and visualization in Python. The Visualization Toolkit (VTK) is a mature and sophisticated visualization library written in C++ that ships with Python bindings. Recently, developers have introduced a new set of bindings called Traits-enabled VTK (TVTK),⁵ which provides seamless integration with the NumPy array objects and libraries as well as a higher-level API for application development. Figure 9 shows how to use TVTK interactively from within an IPython session. Because matplotlib has WXPYTHON support, you can use both TVTK and matplotlib concurrently from within IPython.

Interactive Parallel and Distributed Computing

Although interactive computing environments can be extremely productive, they've traditionally had one weakness: they haven't been able to take advantage of parallel computing hardware such as multicore CPUs, clusters, and supercomputers. Thus, although scientists often begin projects using an interactive computing environment, at some point they switch to using languages such as C, C++, and Fortran when performance becomes critical and their projects call for parallelization. In recent years, several vendors have begun offering distributed computing capabilities for the major commercial technical computing systems (see the "Distributed Computing Toolkits for Commercial Systems" sidebar for some examples). These provide various levels of integration between the computational back ends and interactive front ends. An early precursor to these systems, whose model was one of full interactive access to the computational

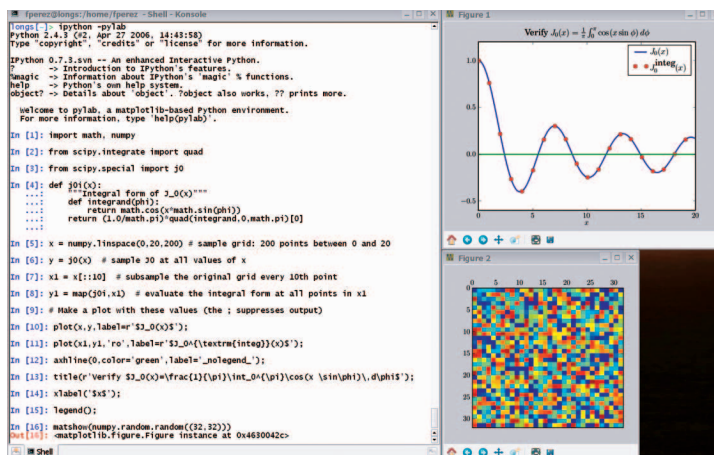


Figure 8. IPython using the `-pylab` flag to enable interactive use of the matplotlib library. Plot windows can open without blocking the interactive terminal, using any of the GUI toolkits supported by matplotlib (Tk, WxWidgets, GTK, or Qt).

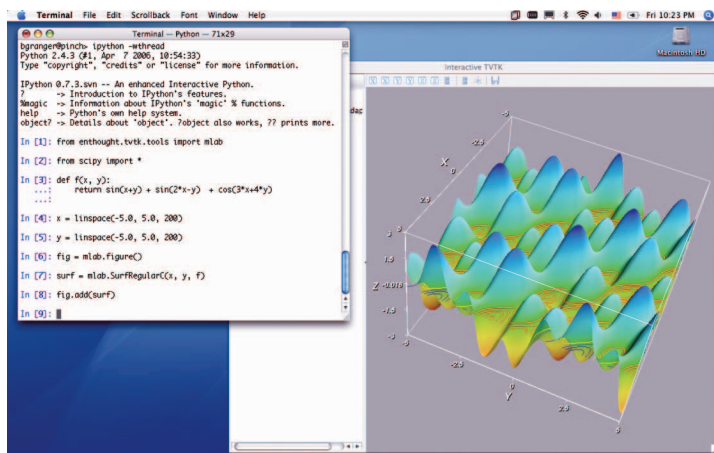


Figure 9. An IPython session showing a 3D plot done with TVTK. The GUI toolkit used is WXPYTHON, so IPython is started with the `-wthread` flag.

nodes, is ParGAP (www.ccs.neu.edu/home/gene/pargap.html), a parallel-enabled version of the open source package Groups, Algorithms, and Programming (GAP) for computational group theory.

In the Python world, several projects also exist that seek to add support for distributed computing. The Python-community-maintained wiki keeps a list of such efforts (<http://wiki.python.org/moin/ParallelProcessing>). Of particular interest to scientific users, Python has been used in parallel computing contexts both with the message-passing interface (MPI, <http://sourceforge.net/projects/pypmpi>; <http://mpi4py.scipy.org>)^{6,7} and the Bulk Synchronous Parallel⁸ models.

DISTRIBUTED COMPUTING TOOLKITS FOR COMMERCIAL SYSTEMS

Some vendors offer distributed computing capabilities for the major commercial technical computing systems:

- Matlab Distributed Computing Toolbox, www.mathworks.com/products/distribtb.
- FastDL, www.txcorp.com/products/FastDL.
- Mathematica Parallel Computing Toolkit, <http://documents.wolfram.com/applications/parallel>.
- Mathematica Personal Grid Edition, www.wolfram.com/products/personalgrid.
- Grid Mathematica, www.wolfram.com/products/gridmathematica.
- HPC-Grid, www.maplesoft.com/products/toolboxes/HPCgrid.
- Star-P, www.interactivesupercomputing.com.

Other projects seek to support distributed computing using Python (see <http://wiki.python.org/moin/ParallelProcessing>).

Building on Python's and IPython's strengths as an interactive computing system, we've begun a significant effort to add interactive parallel and distributed capabilities to IPython. More specifically, our goal is to enable users to develop, test, debug, execute, and monitor parallel and distributed applications interactively using IPython. To make this possible, we've refactored IPython to support these new features. We've deliberately built a system whose basic components make no specific assumptions about communications models, data distribution, or network protocols. The redesigned IPython consists of

- the IPython *core*, which exposes IPython's core functionality, abstracted as a Python library rather than as a terminal-based application;
- the IPython *engine*, which exposes the IPython core's functionality to other processes (either local to the same machine or remote) over a standard network connection; and
- the IPython *controller*, which is a process that exposes a clean asynchronous interface for working with a set of IPython engines.

With these basic components, specific models of distributed and parallel computing can be implemented as user-visible systems. Currently, we support two models out of the box: a load-balancing and fault-tolerant task-farming interface for coarse-grained parallelism, and a lower-level interface that

gives users direct interactive access to a set of running engines. This second interface is useful for both medium- and fine-grained parallelism that uses MPI for communications between engines. Most importantly, advanced users and developers can use these components to build customized interactive parallel/distributed applications in Python. End users work with the system interactively by connecting to a controller using a Web browser, an IPython- or Python-based front end, or a traditional GUI.

Specific constraints that are relevant in scientific computing guided this design:

- It should support many different styles of parallelism, such as message passing using MPI, task farming, and shared memory.
- It should run on everything from multicore laptops to supercomputers.
- It should integrate well with existing parallel code and libraries written using C, C++, or Fortran, and MPI for communications.
- All network communications, events, and error handling should be fully asynchronous and nonblocking.
- It should support all of IPython's existing features in parallel contexts.

The architectural requirements for running IPython in a distributed manner are similar to those required for decoupling a user front end from a computational back end. Therefore, this restructuring effort also lets IPython offer new types of user interfaces for remote and distributed work, such as a Web browser-based IPython GUI and collaborative interfaces that enable multiple remote users to simultaneously access and share running computational resources and data.

The first public release of these new components was in late 2006. While it should still be considered under heavy development and subject to changes, we've already been contacted by several projects that have begun using it as a tool in production codes. Details about this work are available on the IPython Web site.

Acknowledgments

IPython wouldn't be where it is today if it weren't for its user community's contributions. Over the years, users have sent bug reports, ideas, and often major portions of new code. Some of the more prolific contributors have become codevelopers. As a Free

Software project, it is only because of such a community that it continues to improve. We thank Ville Vainio for maintaining the stable branch of the project, and Benjamin Ragan-Kelley for his continued work as a key developer of IPython's distributed and parallel computing infrastructure.

This research was partially supported by US Department of Energy grant DE-FG02-03ER25583 and DOE/Oak Ridge National Laboratory grant 4000038129 (F. Pérez) and by Tech-X Corporation (B. Granger). We thank Enthought for the hosting and infrastructure support it has provided to IPython over the years.

References

1. T.-Y.B. Yang, G. Furnish, and P.F. Dubois, "Steering Object-Oriented Scientific Computations," *Proc. Technology of Object-Oriented Languages and Systems (TOOLS)*, IEEE CS Press, 1998, pp. 112–119.
2. W. Stein and D. Joyner, "SAGE: System for Algebra and Geometry Experimentation," *Comm. Computer Algebra*, vol. 39, 2005, pp. 61–64.
3. P. Barrett, J. Hunter, and P. Greenfield, "Matplotlib: A Portable Python Plotting Package," *Astronomical Data Analysis Software & Systems*, vol. 14, 2004.
4. E. Jones, T. Oliphant, and P. Peterson, "SciPy: Open Source Scientific Tools for Python," 2001; www.scipy.org.
5. P. Ramachandran, "TVTK: A Pythonic VTK," *Proc. EuroPython Conf.*, EuroPython, 2005; <http://svn.enthought.com/enthought/attachment/wiki/TVTK/tvtk-paper-epc2005.pdf>.
6. D.M. Beazley and P.S. Lomdahl, "Extensible Message Passing Application Development and Debugging with Python," *Proc. 11th Int'l Parallel Processing Symp.*, IEEE CS Press, 1997, pp. 650–655.
7. P. Miller, "Parallel, Distributed Scripting with Python," *Third Linux Clusters Inst. Int'l Conf. Linux Clusters: The HPC Revolution*, Lawrence Livermore Nat'l Laboratory, 2002; www.llnl.gov/tid/lof/documents/pdf/240425.pdf.
8. K. Hinszen, "High-Level Parallel Software Development with Python and BSP," *Parallel Processing Letters*, vol. 13, s2003, pp. 473–484.

Fernando Pérez is a research associate in the Department of Applied Mathematics at the University of Colorado at Boulder. His research interests include new algorithms for solving PDEs in multiple dimensions with a focus on problems in atomic and molecular structure, the use of high-level languages for scientific computing, and new approaches to distributed and parallel problems. Pérez has a PhD in physics from the University of Colorado. Contact him at Fernando.Perez@colorado.edu.

Brian E. Granger is a research scientist at Tech-X. He has a background in scattering and many-body theory in the context of atomic, molecular, and optical physics. His research interests include interactive parallel and distributed computing, remote visualization, and Web-based interfaces in scientific computing. Granger has a PhD in theoretical physics from the University of Colorado. Contact him at bgranger@txcorp.com.

AMERICAN INSTITUTE OF PHYSICS

The American Institute of Physics is a not-for-profit membership corporation chartered in New York State in 1931 for the purpose of promoting the advancement and diffusion of the knowledge of physics and its application to human welfare. Leading societies in the fields of physics, astronomy, and related sciences are its members.

In order to achieve its purpose, AIP serves physics and related fields of science and technology by serving its member societies, individual scientists, educators, students, R&D leaders, and the general public with programs, services, and publications—information that matters.

The Institute publishes its own scientific journals as well as those of its member societies; provides abstracting and indexing services; provides online database services; disseminates reliable information on physics to the public; collects and analyzes statistics on the profession and on physics education; encourages and assists in the documentation and study of the history and philosophy of physics; cooperates with other organizations on educational projects at all levels; and collects and analyzes information on federal programs and budgets.

The Institute represents approximately 134,000 scientists through its member societies. In addition, approximately 6,000 students in more than 700 colleges and universities are members of the Institute's Society of Physics Students, which includes the honor society Sigma Pi Sigma. Industry is represented through the membership of 38 Corporate Associates.

Governing Board: * *Mildred S. Dresselhaus* (chair), David Aspnes, Anthony Atchley, Martin Blume, *Marc H. Brodsky* (ex officio), Slade Cargill, *Charles W. Carter Jr.*, Hilda A. Cerdeira, Marvin L. Cohen, *Timothy A. Cohn*, Lawrence A. Crum, Bruce H. Curran, *Morton M. Denn*, Robert E. Dickinson, Michael D. Duncan, *Judy R. Franz*, Brian J. Fraser, *John A. Graham*, *Toufic Hakim*, Joseph H. Hamilton, Ken Heller, James N. Hollenhorst, Judy C. Holoviak, John J. Hopfield, Anthony M. Johnson, Angela R. Keyser, Louis J. Lanzerotti, Harvey Leff, *Rudolf Ludeke*, Robert W. Milkey, John A. Orcutt, Richard W. Peterson, S. Narasinga Rao, *Elizabeth A. Rogan*, Bahaa A.E. Saleh, *Charles E. Schmid*, Joseph Serene, *James B. Smathers*, *Benjamin B. Snavely* (ex officio), A.F. Spilhaus Jr, and Hervey (Peter) Stockman.

*Board members listed in italics are members of the Executive Committee.

Management Committee: Marc H. Brodsky, Executive Director and CEO; Richard Baccante, Treasurer and CFO; Theresa C. Braun, Vice President, Human Resources; James H. Stith, Vice President, Physics Resources; Darlene A. Walters, Senior Vice President, Publishing; and Benjamin B. Snavely, Secretary.

www.aip.org