



**Proceedings of the 13th
Python in Science Conference**

July 6 - 12 • Austin, Texas

Stéfan van der Walt
James Bergstra

PROCEEDINGS OF THE 13TH PYTHON IN SCIENCE CONFERENCE

Edited by Stéfan van der Walt and James Bergstra.

SciPy 2014
Austin, Texas
July 6 - 12, 2014

Copyright © 2014. The articles in the Proceedings of the Python in Science Conference are copyrighted and owned by their original authors

This is an open-access publication and is distributed under the terms of the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original author and source are credited.

For more information, please see: <http://creativecommons.org/licenses/by/3.0/>

ISSN:2575-9752
<https://doi.org/10.25080/Majora-14bd3278-011>

ORGANIZATION

Conference Chairs

KELSEY JORDAHL, Enthought, Inc.
ANDY R. TERREL, Continuum Analytics

Program

KATY HUFF, University of California, Berkeley
SERGE RAY, Arizona State University

Communications

ANTHONY SCOPATZ, University of Wisconsin-Madison
MATTHEW TURK, Columbia University

Birds of Feathers

KYLE MANDLI, University of Texas at Austin
MATT MCCORMICK, Kitware, Inc.

Proceedings

STÉFAN VAN DER WALT, Stellenbosch University
JAMES BERGSTRA, University of Waterloo

Financial Aid

JOHN WIGGINS, Enthought, Inc.
JEFF DAILY, Pacific Northwest National Laboratory

Tutorials

JAKE VANDERPLAS, University of Washington
KRISTEN THYNG, Texas A&M University

Sprints

CORRAN WEBSTER, Enthought, Inc.
JONATHAN ROCHER, Enthought, Inc.

Technical

MATTHEW TERRY, Nextdoor.com
SHEILA MIGUEZ
JIM IVANOFF, Polar Bear Design

Sponsors

BRETT MURPHY, Enthought, Inc.
JILL COWAN, Enthought, Inc.

Financial

BILL COWAN, Enthought, Inc.
JODI HAVRANEK, Enthought, Inc.

Logistics

LEAH JONES, Enthought, Inc.

Program Committee

ARON AHMADIA
DANIEL ARRIBAS
CHRIS BARKER
DANA BAUER
AZALEE BOSTROEM
MATTHEW BRETT
HOWARD BUTLER
NEAL CAREN
AASHISH CHAUDHARY
ADINA CHUANG HOWE
CHRISTINE CHOIRAT
DAV CLARK
JEAN CONNELLY
MATT DAVIS
MICHAEL DROETTBOOM
JUAN CARLOS DUQUE
DANIEL DYE
CARSON FARMER
DAVID FOLCH
PATRICIA FRANCIS-LYON
SEAN GILLIES
BRIAN GRANGER
PAUL IVANOV
MARK JANIKAS
JACKIE KAZIL
HANS PETER LANGTANGEN
JASON LAURA
DANIEL J. LEWIS
WENWEN LI
CINDEE MADISON
MATTHEW MCCORMICK
MIKE MCKERNS
AUGUST MUENCH
ANA NELSON
FLORIAN RATHGEBER
MATTHEW ROCKLIN
TOM ROBITAILLE
CHARLES R. SCHMIDT
SKIPPER SEABOLD
ASHTON SHORTRIDGE
RICH SIGNELL
WILLIAM SPOTZ
JOHN STACHURSKI
PHILIP STEPHENS
TRACY TEAL
KRISTEN M. THYNG
ERIK TOLLERUD
JAKE VANDERPLAS
SHAI VAINGAST
GAËL VAROQUAUX
NELLE VAROQUAUX
SHAUN WALBRIDGE
XINYUE YE
ALEX ZVOLEFF

Proceedings Reviewers

ARON AHMADIA
DANI ARRIBAS-BEL
CHRIS BARKER

AZALEE BOSTROEM
MATTHEW BRETT
CHRISTINE CHOIRAT
DAVID FOLCH
SATRAJIT GHOSH
BRIAN GRANGER
PIETER HOLTZHAUSEN
HANS PETER LANGTANGEN
TERRY LETSCHE
DANIEL LEWIS
AUGUST MUENCH
JOSEF PERKTOLD
BENJAMIN RAGAN-KELLEY
MATTHEW ROCKLIN
DANIEL SOTO
ERIK TOLLERUD
JAKE VANDERPLAS

Mini Symposium Committee

TREVOR BLANARIK AND NICHOLAS LEDERER, Engineering
DAV CLARK, Computational Social Science and Digital Humanities
MATT HALL, Geophysics
MATTHEW MCCORMICK, Vision, Visualization, and Imaging
THOMAS ROBITAILLE, Astronomy and Astrophysics
TRACY TEAL, Bioinformatics

Diversity Committee

PATRICIA FRANCIS-LYON, University of San Francisco
KATY HUFF, University of California, Berkeley
LEAH SILEN, Numfocus
KRISTEN THYNG, Texas A&M University

CONTENTS

Preface	1
<i>Andy Terrel, Jonathan Rocher, Stéfan van der Walt, James Bergstra</i>	
Scientific Computing with SciPy for Undergraduate Physics Majors	2
<i>G William Baxter</i>	
BCE: Berkeley's Common Scientific Compute Environment for Research and Education	5
<i>Dav Clark, Aaron Culich, Brian Hamlin, Ryan Lovett</i>	
Measuring rainshafts: Bringing Python to bear on remote sensing data	13
<i>Scott Collis, Scott Giangrande, Jonathan Helmus, Di Wu, Ann Fridlind, Marcus van Lier-Walqui, Adam Theisen</i>	
Teaching numerical methods with IPython notebooks and inquiry-based learning	19
<i>David I. Ketcheson</i>	
Project-based introduction to scientific computing for physics majors	25
<i>Jennifer Klay</i>	
Hyperopt-Sklearn: Automatic Hyperparameter Configuration for Scikit-Learn	32
<i>Brent Komer, James Bergstra, Chris Eliasmith</i>	
Python Coding of Geospatial Processing in Web-based Mapping Applications	38
<i>James A. Kuiper, Andrew J. Ayers, Michael E. Holm, Michael J. Nowak</i>	
Scaling Polygon Adjacency Algorithms to Big Data Geospatial Analysis	45
<i>Jason Laura, Sergio J. Rey</i>	
Campaign for IT literacy through FOSS and Spoken Tutorials	51
<i>Kannan M. Moudgalya</i>	
Python for research and teaching economics	59
<i>David R. Pugh</i>	
Validated numerics with Python: the ValidiPy package	65
<i>David P. Sanders, Luis Benet</i>	
Creating a browser-based virtual computer lab for classroom instruction	72
<i>Ramalingam Saravanan</i>	
TracPy: Wrapping the Fortran Lagrangian trajectory model TRACMASS	79
<i>Kristen M. Thyng, Robert D. Hetland</i>	
Frequentism and Bayesianism: A Python-driven Primer	85
<i>Jake VanderPlas</i>	
Blaze: Building A Foundation for Array-Oriented Computing in Python	94
<i>Mark Wiebe, Matthew Rocklin, TJ Alumbaugh, Andy Terrel</i>	
Simulating X-ray Observations with Python	98
<i>John A. ZuHone, Veronica Biffi, Eric J. Hallman, Scott W. Randall, Adam R. Foster, Christian Schmid</i>	

Preface

Andy Terrel^{‡*}, Jonathan Rocher[§], Stéfan van der Walt^{||}, James Bergstra[¶]

SciPy 2014, the thirteenth annual Scientific Computing with Python conference, was held July 6–12th in Austin, Texas. SciPy is a community dedicated to the advancement of scientific computing through open source Python software for mathematics, science, and engineering.

The SciPy conferences have become a prominent forum for Python users from the academic, commercial and government sectors to present and develop their latest tools and innovations. Topics cover a wide array of domains, from cross-language interactions to education and cutting-edge research. These events, by virtue of embracing both in-depth scientific exploration and programming/code, form a unique connection between the academic and developer communities. At SciPy, code, science, and math live on the same screen.

It is an exciting time to be part of this community that has spent the last decade developing sophisticated tools—tools that are now ideally suited to address technical problems arising at the blooming intersection of specialized domains and computation. Many contributors to the community have been hired at university data institutions, Python has become the number one language for undergraduate teaching, and many productive partnerships have been formed with industry.

The conference continues to grow with almost 500 participants from across the globe. More than half of attendees are now from industry, the rest split between government laboratories and the academy. The organizing committee is committed to increasing representation from underrepresented groups. This year, 15% of attendees were women, a significant increase from 3% in 2013. A Diversity Committee was formed to ensure that this trend continues.

Geospatial Computing and Education were central themes this year, with additional minisymposia on the following topics:

- Astronomy and Astrophysics
- Bioinformatics
- Geophysics
- Vision, Visualization, and Imaging
- Computational Social Science and Digital Humanities
- Engineering

Birds of a Feather sessions were organized around select solicited topics, providing an effective platform for discussing issues rele-

vant to the community. New open space activities, sponsor funded social events and tutorials effectively exposed newcomers to the welcoming and inclusive scientific Python community.

We were privileged to have three prominent community members present keynotes. Greg Wilson gave a heart-felt call for action, encouraging the enhancement of tools for education in scientific computing. Lorena Barba focused on the interactions between computation, the system under study and learning, highlighting the importance of tools that facilitate those connections. The final keynote speaker, Python core developer Nick Coghlan, presented his perspective on the distribution of open source tools, emphasizing the need to bridge gaps that exist between various channels of distribution.

These proceedings contain 16 peer-reviewed contributions, based on talks presented at the conference. They provide a peek into the current state of the ever-evolving landscape of Python in Science. We hope you find pleasure in the effort the authors have made to carefully present their work in a clear and accessible fashion.

On behalf of the SciPy2014 organizers,

Andy Terrel & Kelsey Jordahl, conference chairs
Stéfan van der Walt & James Bergstra, proceedings chairs

* Corresponding author: aterrell@tacc.utexas.edu

‡ Continuum Analytics

§ Enthought, Inc.

|| University of California, Berkeley

¶ University of Waterloo

Scientific Computing with SciPy for Undergraduate Physics Majors

G William Baxter^{‡*}



Abstract—The physics community is working to improve the undergraduate curriculum to include computer skills that graduates will need in the workforce. At Penn State Erie, The Behrend College, we have added computational tools to our Junior/Senior physics laboratory, PHYS421w Research Methods. The course emphasizes Python software tools (SciPy) for data analysis rather than traditional programming. The course uses real experiments to motivate the mastery of these tools.

Index Terms—laboratory, computing, software tools, experiment

Introduction

There is a growing debate within the physics community as to what skills a physics graduate with a Bachelor degree must have to be successful in industry or academia [Chonacky2008], [Landau2006]. The computer has become such a basic tool for data acquisition, process control, and analysis that an undergraduate physics degree should include computer skills. However, while some undergraduate physics degrees require one or more courses in programming, often C++ programming, many physics degrees require no computer courses at all [Fuller2006]. To remedy this, computing could be taught by way of a new course [Kaplan2004], [Spencer2005], by adding the material to an existing course [Caballero2014], [Timberlake2008], [Serbanescu2011], [Nakroshis2013] or both [Taylor2006]. Many degree programs are limited by their total number of credits so that adding a computing course would require removing an existing course. Easier is to add the material to an existing course. At Penn State Erie, we added the material to our advanced laboratory, PHYS 421w Research Methods [Hanif2009].

In those majors that include computation, the emphasis is often on either simulation or programming numerical techniques. A strong case can be made that a student would benefit from less traditional programming and more *computing with software tools*, by which we mean analyzing data or solving problems with software which might be created by someone else. At Penn State Erie, we require a traditional introductory course in C++ programming, but we have added an emphasis on computing with software tools to our advanced laboratory, PHYS 421w Research Methods.

* Corresponding author: gwb6@psu.edu

‡ Physics, School of Science, Penn State Erie - The Behrend College

Copyright © 2014 G William Baxter. This is an open-access article distributed under the terms of the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original author and source are credited.

Experiment	Type
Charge to Mass Ratio of the Electron	Exp
Cavendish Measurement of G	Exp
Millikan Oil Drop Measurement of e	Exp
Harmonic Motion and Torsion Oscillators	Exp
Domains and Bubbles in a Magnetic Film	Exp
Two-slit Interference, One Photon at a Time	Exp
Earth's Field Nuclear Magnetic Resonance	Exp
Vibrations of a Vertical Chain Fixed at One End	Exp
Video Microscop of Brownian Motion	Exp
Diffusivity in Liquids	Exp
Percolation	Sim
Scaling Properties of Random Walks	Sim
Critical Temperature of a Two Dimensional Ising Model	Sim

TABLE 1: Partial list of available experiments for PHYS421w.

Research Methods Laboratory

PHYS 421w Research Methods is a 3 credit lab in which each student chooses three experiments from classical and modern physics. See Table 1 for a partial list of experiments. Most are physical experiments, but students are allowed to do a computer simulation for their third experiment. Note that the data acquisition and analysis can be very different for different experiments. Some involve adjusting a parameter and making a measurement, others involve extracting measurements from a time series of measurements, and others involve extracting a measurement from a series of images.

The course's guiding principles are:

- Experiments should be as close as possible to the way physics is really done including submitting LaTeX papers for peer-reviewed grading.
- Emphasis is on software tools for analysis and publication rather than on numerical techniques.
- Software tools presented will be needed in the experiments. However useful they may be, we do not introduce tools which do not apply to any of these experiments.
- All software will be free and open-source.

Students are strongly motivated to learn when they believe the material will be useful in their future careers. Here, the emphasis is on a *realistic* experimental experience. Students have 4 weeks to setup, perform, analyze, and write up the results of each experiment. Papers must adhere to the standards for publication in an

Software Tools Topic	Included Tools	Covered?
Visualization	matplotlib	Always
Modeling and Fitting	scipy.optimize.leastsq	Always
Array Operations	python, numpy	Always
Statistics and Uncertainty	numpy.statistics & special	Always
Special Functions	numpy.special	As Needed
Image Processing	numpy, PIL, scipy.ndimage	As Needed
Frequency Space	numpy.fft	As Needed
Differential Equations	scipy.integrate.odeint	Always
[Monte Carlo Techniques]	python	As Needed

TABLE 2: Software tool categories.

American Physical Society (APS) Physical Review journal. Papers are reviewed by a set of anonymous reviewers who comment on each paper. Authors have the opportunity to rewrite their paper and submit it for a final review. Papers must be written in LaTeX with the APS RevTeX extensions.

The course is taken by Junior and Senior level physics majors and some engineering students seeking a minor in physics. Students entering the course typically have had one or more computer programming classes using C++ taught by the Computer Science Department. On average, their programming skills are poor. They have some familiarity with Microsoft Word and Excel. Excel is a good package for business; but, it has serious limitations for scientific work, despite being used for nearly all introductory science labs at Penn State Erie. These limitations include: poor handling of uncertainties and error bars, limited fit functions, no Fourier transforms, and no image processing capability. They are very poorly prepared for the manipulation and analysis of experimental data.

The course begins with two lectures introducing the Unix/Linux operating system. It continues with 4 lectures on LaTeX and BibTeX. Each lecture is followed by a homework assignment which allows the student to practice the day's topic. Then a series of lectures on Scientific Python tools follow as shown in Table 2. Students receive custom documentation¹ on each topic with many examples showing the commands and the results as well as homework on each topic for practice.

We begin with plotting and *visualization*. Viewing data is the first step to determining what to do with it. Students often have little experience with error bars and histograms and no experience with when or how to use logarithmic scales. This topic also includes reading and writing of data files. We follow this with a discussion of and exercises on *modeling and fitting*. Students are given five noisy data sets. With no additional information on each, they first determine the correct functional form and necessary parameters and initial conditions. Then they must determine the best-fit parameters with uncertainties on all parameters and plot the fitted curve through the data. "Guessing" the functional form is difficult for many students, but they are strongly motivated by the fact that they know they will have to use this skill in their upcoming experiments. Examples of the data sets and fitted curves are shown in figure 1. Notice that there is little discussion of the numerical technique. We are choosing to treat this as a *tool* and save discussions of the details of the numerical technique for a

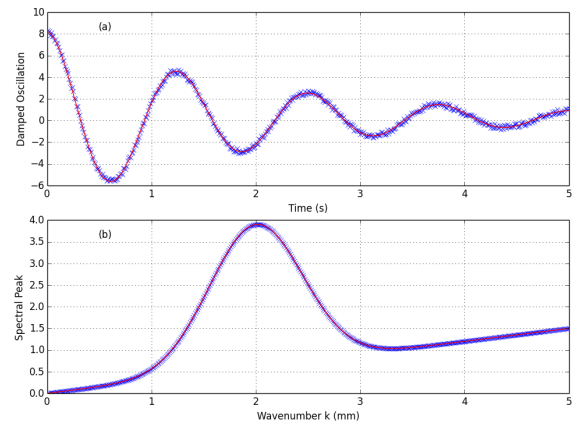


Fig. 1: Examples of two data sets used for fitting practice. Students are given only a simple data file with no additional information. They must decide on the appropriate function and the necessary fit parameters. In (a), $y(t) = 8.0e^{-0.5t} \cos(5.0t) + 0.25$ and in (b) $y(k) = 3.3e^{-2.5(k-2.0)^2} + 0.30k$.

numerical analysis course, an optional course in our major but not a requirement.

Some may be concerned that focusing on software tools rather than numerical methods may lead students to believe that they never need a deeper understanding of the numerical methods upon which these tools depend. I believe this risk is small. Using fitting as an example, we do discuss in a very general way the method of least squares. As they use the tool, students quickly learn that a poor initial guess often leads to nonsensical results and wonder "why?" I believe it likely that, having used a tool, students will be *more motivated* to learn the numerical methods on which it depends.

The *array operations* material is introduced so that students will be able to rescale and manipulate data once it is read from a file. For example, to collect spectra, we use a grating spectrometer, a light sensor and LABview to get a series of data points (12 per degree) from the $m=1$ spectral lines on one side of the center line to approximately the same point on the other side. Using this data, the student must determine the zero angle point, average angles and signals on both sides, and convert to wavelengths. The *statistics* material is used to introduce students to a serious discussion of uncertainty, error, and distributions. We discuss and calculate standard deviations both theoretically and for real data. And we explore non-Gaussian distributions such as the Poisson and binomial distributions which occur in real experiments.

Other topics are introduced as needed depending on which experiments students have chosen. *Image processing* is introduced when any students are doing experiments which take data in the form of images (such as the magnetic film, vibrations of a vertical chain, and video microscopy). The specific goal is the extraction of useful data from images. Specific material includes image formats and conversion to matrix form, region of interest, background subtraction, thresholding, and filtering to find lines or points. *Special functions* is introduced when experiments will produce data that has a functional form of Bessel, Legendre, or other special functions. These occur often in optics, electrostatic, and wave problems. Without knowing how to access these functions in numpy, fitting or modeling the data would not be possible. *Fre-*

1. Materials are available for download from <https://sites.psu.edu/teachingexperimentalphysics/>.

quency space introduces the Fourier transform, FFT, windowing and the power spectrum. It is particularly useful for analyzing experiments which have either a temporal or spatial periodicity. The *differential equations* material is introduced so that it can be used in a Junior/Senior classical mechanics class offered the following semester.

Discussion

We have no formal assessment in place; however, anecdotal evidence is positive. Returning graduates have specifically cited the ability to fit experimental data as valuable in graduate school. Faculty have said they value research students who have learned to plot and fit data in this course, and some students have set up our software tool environments on their own computers. From my perspective as professor, the quality of student figures and analysis in PHYS 421w has dramatically improved. It remains a challenge to convince some students that they need to know more than Microsoft Excel; but, students are more interested in learning software tools when they see their utility and know they will need to use them. Ideally, any course following PHYS 421w should reinforce these skills by also requiring students to use these computer tools; however, at Penn State Erie, it has been difficult to get faculty to include computer skills in upper-level courses; at present only classical mechanics uses any computer tools. This course and its software tools materials remain a work in progress.

Acknowledgements

I gratefully acknowledge the assistance and feedback of the students of PHYS 421w at Penn State Erie, The Behrend College.

REFERENCES

- [Caballero2014] M. Caballero and S. Pollock, *A model for incorporating computation without changing the course: An example from middle-division classical mechanics*, American Journal of Physics 82 (2014) pp231-237.
- [Chonacky2008] N. Chonacky and D. Winch, *Integrating computation into the undergraduate curriculum: A vision and guidelines for future developments*, American Journal of Physics, 76(4&5) (2008) pp327-333.
- [Fuller2006] R. Fuller, *Numerical Computations in US Undergraduate Physics Courses*, Computing in Science and Engineering, September/October 2006, pp16-21.
- [Hanif2009] M. Hanif, P. H. Sneddon, F. M. Al-Ahmadi, and R. Reid, *The perceptions, views and opinions of university students about physics learning during undergraduate laboratory work*, Eur J. Phys, 30, 2009, pp85-96.
- [Kaplan2004] D. Kaplan, *Teaching computation to undergraduate scientists*, SIGSCE, Norfolk, VA, March 3-7, 2004.
- [Landau2006] R. Landau, *Computational Physics: A better model for physics education?*, Computing in Science and Engineering, September/October 2006, pp22-30.
- [Nakroshis2013] P. Nakroshis, *Introductory Computational Physics Using Python*, unpublished course notes, 2013.
- [Serbanescu2011] R. Serbanescu, P. Kushner, and S. Stanley, *Putting computation on a par with experiments and theory in the undergraduate physics curriculum*, American Journal of Physics, 79 (2011), pp919-924.
- [Spencer2005] R. Spencer, *Teaching computational physics as a laboratory sequence*, 73, (2005), pp151-153.
- [Taylor2006] J. Taylor and B. King, *Using Computational Methods to Reinvigorate an Undergraduate Physics Curriculum*, Computing in Science and Engineering, September/October 2006, pp38-43.

- [Timberlake2008] T. Timberlake and J. Hasbun, *Computation in classical mechanics*, American Journal of Physics, 76 (2008), pp334-339.

BCE: Berkeley's Common Scientific Compute Environment for Research and Education

Dav Clark^{‡*}, Aaron Culich[‡], Brian Hamlin[§], Ryan Lovett[‡]

<http://www.youtube.com/watch?v=e7jaZ5SFvFk>



Abstract—There are numerous barriers to the use of scientific computing toolsets. These barriers are becoming more apparent as we increasingly see mixing of different academic backgrounds, and compute ranging from laptops to cloud platforms. Members of the UC Berkeley D-Lab, Statistical Computing Facility (SCF), and Berkeley Research Computing (BRC) support such use-cases, and have developed strategies that reduce the pain points that arise. We begin by describing the variety of concrete training and research use-cases in which our strategy might increase accessibility, productivity, reuse, and reproducibility. We then introduce available tools for the “recipe-based” creation of compute environments, attempting to demystify and provide a framework for thinking about *DevOps* (along with explaining what “DevOps” means!). As a counterpoint to novel DevOps tools, we’ll also examine the success of OSGeo-Live [OSGL] – a project that has managed to obtain and manage developer contributions for a large number of geospatial projects. This is enabled through the use of commonly known skills like shell scripting, and is a model of complexity that can be managed *without* these more recent DevOps tools. Given our evaluation of a variety of technologies and use-cases, we present our current strategy for constructing the Berkeley Common Environment [BCE], along with general recommendations for building environments for your own use-cases.

Index Terms—education, reproducibility, virtualization

Introduction

Most readers of this paper will have dealt with the challenges of sharing or using complex compute stacks – be that in the course of instruction, collaboration, or shipping professional software. Here, we suggest an approach for introducing novices to new software that reduces complexity by providing a *standard reference* end-user environment. We’ll discuss approaches to building and using a common environment from any major OS, including an overview of the tools available to make this easier. This approach can make it easier to provide complete and robust instructions, and make it easier for students to follow demos.

At a university, students often need to reproduce an environment required to run the software for a course. Researchers need to reproduce their collaborator’s workflows, or *anyone’s* workflow in the name of reproducible research. Recently, a new crop of tools-for-managing-tools has emerged under the *DevOps* banner – a contraction of software *development* and systems *operation*

– with a general philosophy that instead of merely documenting systems operation tasks (configuration, deployment, maintenance, etc.), that developers can and should be scripting these tasks as much as possible.

In scientific computing the environment was commonly managed via Makefiles & Unix-y hacks, or alternatively with monolithic software like Matlab. More recently, centralized package management has provided curated tools that work well together. But as more and more essential functionality is built out across a variety of systems and languages, the value – and also the difficulty – of coordinating multiple tools continues to increase. Whether we are producing research results or web services, it is becoming increasingly essential to set up new languages, libraries, databases, and more.

Documentation for complex software environments is stuck between two opposing demands. To make things easier on novice users, documentation must explain details relevant to factors like different operating systems. Alternatively, to save time writing and updating documentation, developers like to abstract over such details. A DevOps approach to “documenting” an application might consist of providing brief descriptions of various install paths, along with *scripts* or “recipes” that automate setup. This can be more *enjoyable* and certainly easily and robustly reproducible for end-users – even if your setup instructions are wrong, they will be reproducibly wrong! As we’ll describe below, many readers will already have tools and skills to do this, in the form of package management and basic shell scripting. In other words, the primary shift that’s required is not one of new tooling, as most developers already have the basic tooling they need. Rather, the needed shift is one of *philosophy*.

We recognize that excellent tools have been developed to allow for configuring *Python* environments, including environments that peacefully co-exist on the same computer (e.g., pip, virtualenv, venv, conda, and buildout). These specialized tools can increase our efficiency and provide ready access to a broader range of options (such as different versions or compile-time settings). But, we may also wish to coordinate the desktop environment, including text editors, version control systems, and so on. As such, these tools from the Python community to manage packages and run-time environments cannot solve all of our problems. But any of them could be used within the broader approach we’ll describe.

More recent configuration management tools are directed at solving this larger problem of configuring nearly any aspect of a compute system, and yet other DevOps tools provide efficient ways of managing environments across compute contexts. Unfor-

* Corresponding author: davclark@berkeley.edu

‡ UC Berkeley

§ OSGeo California Chapter

tunately, the variety and complexity of tools match the variety and complexity of the problem space, and the target space for most of them was *not* scientific computing. Thus, before discussing available tooling, we first lay out a fuller set of concerns relevant to supporting scientific computing.

Issues for Scientific Computing

The users of computational tools (and their collaborators) are often equipped with a suite of informally learned foundational skills (command line usage, knowledge of specific applications, etc.). Newcomers to a field often lack these technical skills, which creates a boundary between those who do and do not (and perhaps cannot) participate in that discipline. However, we are entering an era where these boundaries are becoming barriers to the research and educational mission of our university. Our primary concern at present for the Berkeley Common Environment [BCE] is educational, particularly introductory computational science and statistics. However, where possible, we wish to build an environment that supports the broader set of uses we outline here.

For instruction

We are entering an era where experimental philosophers want to take courses in advanced statistics and sociologists need best-of-breed text analysis. These students are willing to work hard, and might sign up for the university courses meant to provide these skills. But while the group that the course was originally designed for (e.g., statistics or computer science students) have a set of *assumed* skills that are necessary to succeed in the class, these skills aren't taught *anywhere* in the curriculum. In these cases, instructors may spend a large amount of time addressing installation and setup issues – taking time away from higher value instruction. Alternatively, students with divergent backgrounds often drop these classes with the sense that they simply can't obtain these skills. This is not an equitable situation.

It's difficult, however, to write instructions that work for any potential student. As mentioned above, students come to a course with many possible environments (i.e., on their laptop or a server). But if a standardized environment is provided, this task becomes much simpler. Written instructions need fewer special cases, and illustrations can be essentially pixel-identical to what students should be seeing on their screen.

The most accessible instructions will only require skills possessed by the broadest number of people. In particular, many potential students are not yet fluent with notions of package management, scripting, or even the basic idea of command-line interfaces [SWC]. Thus, installing an accessible solution should require only GUI operations. The installed common environment, then, can look and operate in a uniform way. This uniformity can scaffold students' use of more challenging “developer” tools. This “uniformity of the environment in which the user is clicking” cannot be implemented without full control of the graphical environment, and systems that configure only a self-contained set of libraries or computational tools cannot do this. At the other end, it would be unreasonable to reconfigure students' desktop on their laptop. Thus, we wish to set up an isolated, uniform environment in its totality where instructions can provide essentially pixel-identical guides to what the student will see on their own screen.

For scientific collaboration

Across campus, we encounter increasing numbers of researchers who wish to borrow techniques from other researchers. These

researchers often come from different domains with different standards for tools. These would-be collaborators are increasingly moving towards open-source tools – often developed in Python or R – which already dramatically reduces financial barriers to collaboration.

The current situation, however, results in chaos, misery, and the gnashing of teeth. It is common to encounter a researcher with three or more Python distributions installed on their machine, and this user will have no idea how to manage their command-line path, or which packages are installed where. In particularly pathological cases, pip will install packages to an otherwise inactive python distribution. These nascent scientific coders will have at various points had a working system for a particular task, and often arrive at a state in which nothing seems to work. A standard environment can eliminate this confusion, and if needed, isolate environments that serve different projects. Snapshots of working systems can provide even more resilience of the continued functioning of already running projects. And it bears repeating that we don't want to disrupt the already productive environments that these researchers are using!

This issue becomes even more pronounced when researchers attempt to reproduce published results without access to the expert who did the initial research. It is unreasonable to expect any researcher to develop code along with instructions on how to run that code on any potential environment. As with the instructional case above, an easy way to do this is to ensure others have access to the exact environment the original researcher was working on, and again, “pixel-identical” instructions can be provided.

For administration

At UC Berkeley, the D-Lab supports tools for courses and short trainings. Similarly, the Statistical Computing Facility (SCF) supports an instructional lab and “cloud” resources for some courses, and grad student assistants often provide virtual machines for computer science courses (we'll explain virtual machines later). In each and every case, multiple technical challenges are common. These technical glitches can delay or reduce the quality of instruction as compared to an environment that students are already familiar with. It is also a drag on the time of those supporting the course – time that could be better directed at course content!

The more broadly a standard environment is adopted across campus, the more familiar it will be to all students. Using infrastructure for collaborative administration, technical glitches can be tracked or resolved by a community of competent contributors, allowing course instructors to simply use a well-polished end product, while reducing the complexity of instructions for students to set up course-specific software. These environments can also be tuned in ways that would be beyond the scope of what's worth doing for an individual course – for example optimizations to increase the efficiency of numeric computations or network bandwidth for remote desktops.

At this point that our use case starts to sound like the case in which product developers are working together to deploy software on a production server, while maintaining a useful development environment on their own machines, testing servers, and so on. However, going forwards, we will suggest that novel tools for building and managing compute environments be largely the domain of specialized administrator-contributors to a common environment. Technically skilled students, professors and researchers can continue to use the tools they are familiar with, such as the Ubuntu package manager, pip, shell scripts, and so on.

Technical challenges for a common environment

Any common environment needs to provide a base of generally useful software, and it should be clear how it was installed and configured. It should be equally clear how one could set up additional software following the pattern of the “recipe” for the environment, making it easy to share new software with other users of the environment. More generally, we seek to address the following challenges, though we have not definitely solved them! After each problem, we list relevant tools, which will be described in full in a later section.

Complex requirements

The quote at the beginning of this paper illustrates a case in which requirements are not explicitly stated and there is an assumption that all collaborators know how to set up the necessary environment. The number of steps or the time required is unknown, and regularly exceeds the time available. For example, in the context of a 1.5 hour workshop or a class with only handful of participants, if all cannot be set up within a fixed amount of time (typically 20 minutes at most) it will jeopardize successfully completing the workshop or class materials and will discourage participation. All participants must be able to successfully complete the installation with a fixed number of well-known steps across all platforms within a fixed amount of time.

Additional difficulties arises when users are using different versions of the “same” software. For example, Git Bash on Windows lacks a `man` command. We *can't* control the base environment that users will have on their laptop or workstation, nor do we wish to! A useful environment should provide consistency and not depend on or interfere with users' existing setup. Relevant tools discussed below include Linux, virtual machines, and configuration management.

Going beyond the laptop

Laptops are widely used across the research and teaching space and in our experience it is reasonable to assume most individuals will have at least a 64-bit laptop with 4GB of RAM. Such a laptop is sufficient for many tasks, however the algorithms or size of in-memory data may exceed the available memory of this unit-of-compute and the participant may need to migrate to another compute resource such as a powerful workstation with 128GB of RAM (even the most advanced laptops typically max-out at 16GB at the time of this writing). Thus, an environment should not be *restricted* to personal computers. Across systems, a user should be able to replicate the data processing, transformations, and analysis steps they ran on their laptop in this new environment, but with better performance. Relevant tools discussed below include Packer and Docker.

Managing cost / maximizing value

Imagine you have the grant money to buy a large workstation with lots of memory and many processors, but you may only need that resource for a 1 to 2 week period of time. Spending your money on a resource that remains unused 95% of the time is a waste of your grant money! A homogeneous, familiar environment can enable easier usage of the public cloud. A private cloud approach to managing owned resources can also allow more researchers to get value out of those resources. This is a critical enabler to allow us to serve less well-funded researchers. In addition, more recent technologies can avoid exclusively reserving system resources for

Goal	Relevant tools
Make Linux available as a VM (regardless of host OS)	Local VM tool or public cloud (e.g., VirtualBox or Amazon EC2 – choose something supported by Packer)
Apply configurations in a repeatable fashion	Scripting, package managers (e.g., apt, pip), configuration management (e.g., Ansible)
Generate OS image for multiple platforms	Packer
Enable light-weight custom environment (instead of heavy-weight virtualization)	Docker, LXC

TABLE 1: Recommended automation tools for our use-cases.

a single environment. Relevant tools discussed below are Packer, Docker (and LXC), and cloud-based virtual machines.

Existing Tools

As discussed above, the problems outlined above are not unique to scientific computing. Developers and administrators have produced a variety of tools that make it easier to ensure consistent environments across all kinds of infrastructure, ranging from a slice of your personal laptop, to a dynamically provisioned slice of your hybrid public/private cloud. We cannot cover the breadth of tooling available here, and so we will restrict ourselves to focusing on those tools that we've found useful to automate the steps that come before you start *doing science*. We'll also discuss popular tools we've found to add more complexity for our use-cases than they eliminate. Table 1 provides an overview from the perspective of the DevOps engineer (i.e., contributor, maintainer, *you*, etc.).

Linux OS (Operating System)

A foundational tool for our approach is the Linux operating system. It is far easier to standardize on a single OS instead of trying to manage cross-platform support. It is relatively easy to install (or build) scientific code *and* DevOps tools on Linux. Moreover, Linux is not encumbered by licensing constraints, which reduces barriers to collaboration, distribution, and reuse. This choice of a single target OS is a primary reason to use *virtual machines* (described below) because most people don't use Linux as their primary laptop OS.

Virtual machines (VMs)

Virtual machine (VM) software enables running another OS (in BCE, Ubuntu server with XFCE installed) as a *guest* OS inside the *host* OS – often Mac OS or Windows. If a system is not virtualized (for example, the host OS), it is said to be running on “bare metal.” For BCE, we have focused on VirtualBox and VMware (the former of which is free) as they both run on Windows, Mac OS, *and* Linux. Cloud providers like EC2 *only* provide virtual machines (there is no access to “bare metal”), and similar concepts apply across local and cloud virtual systems. A notable distinction is that web tools are often available for cloud services, as opposed to a local GUI tool for systems like VirtualBox. Both kinds of services provide command-line tools that can perform a superset of the tasks possible with graphical interfaces.

For some users, a VM simply will not run locally, generally because they have a very old operating system or computer. Thus, one should assume that any VM solution will not work

for some individuals and provide a fallback solution (particularly for instructional environments) on a remote server. In this case, remote desktop software may be necessary, or in the case of BCE, we are able to enable all essential functionality via a web browser using IPython notebooks. RStudio server would provide a similar approach to sidestepping the need for a full remote desktop session.

One concern is that VMs reserve compute resources exclusively. Some approaches, however, allow for more elastic usage of resources, most notably with LXC-like solutions, discussed in the Docker section below. Another issue that can arise is dealing with mappings between host and guest OS, which vary from system to system – arguing for the utility of an abstraction layer for VM configuration like Vagrant or Packer (discussed below). This includes things like port-mapping, shared files, enabling control of the display for a GUI vs. enabling network routing for remote operation. These settings may also interact with the way the guest OS is configured. Specifically with BCE we noticed that some desktop environments interacted poorly with VirtualBox (for example, LXDE did not handle resize events properly).

Note that if you are already running Linux on “bare metal”, it’s still useful to run a virtualized Linux guest OS. The BCE model relies on a well-known, curated set of dependencies and default configurations. To ensure that it is possible to consistently and reliably manage those elements no matter what flavor, variant, or version of Linux you may be running as the host OS. However, we have intentionally made choices that allow an informed developer set up a partial environment that matches BCE. For example, python requirements are installed with pip using a requirements file. This makes it easy to set up a virtualenv or conda environment with those packages.

The easiest way to use a VM is to use a pre-existing image – a file that contains all relevant data and metadata about an environment (described more fully at [images]). It’s very easy to make modifications to an environment and make a new image by taking a snapshot. Note that while both local and cloud-based VM systems often allow for easy snapshotting, it may be hard to capture exactly how changes happened – especially changes and configuration that was made “by hand.” So, snapshots are not necessarily a good solution for reproducibility. You can also install an OS to a virtual image in essentially the same manner you would install it to bare metal. The primary difference is that you need to use specialized VM software to start this process. For example, you can do this directly in VirtualBox simply by clicking the “New” button, and you’ll be guided through all of the steps. There are more automated ways, however, and we discuss these below.

Configuration management and automated image creation

Creating an image or environment is often called *provisioning*. The way this was done in traditional systems operation was interactively, perhaps using a hybrid of GUI, networked, and command-line tools. The DevOps philosophy encourages that we accomplish as much as possible with scripts (ideally checked into version control!). Most readers of this paper will already be able to create a list of shell commands in a file and execute it as a script. So, if you already know how to execute commands at the Bash prompt to configure Linux, this can do *most* of the system setup for you.

Package managers in particular provide high-level commands to install and configure packages. Currently, we use a combination

of apt, pip, and shell scripts. We also evaluated conda and found that it introduced additional complexity. For example, it is still hard to install a list of pip requirements with conda if some packages are not available for conda. Most package authors currently make their packages available, however, for pip. Standard apt packages were also adequate for things like databases, and ideal for the desktop environment, where we could reap the benefit of the careful work that went into the LTS Ubuntu distribution.

Some steps may even be done manually. As we explored managing the complexity and reducing the number of tools for the BCE development process, one of the steps in the “recipe” was manual installation of Ubuntu from an ISO. It is straightforward to make a binary image from a snapshot immediately after creating a base image, so this initial step could be done once by a careful individual.

Ultimately, however, we decided it was better to automate installation from an ISO, which is enabled by the Debian Installer [UDI], a system that allows a text file to specify answers to the standard configuration prompts at install-time, in addition to providing many more possibilities. You can find the BCE configuration file for the debian-installer in the `provisioning/http` directory. Later, we’ll discuss how we’re coordinating all of the above using Packer.

Ansible and related tools

Ansible is one of a number of recent DevOps tools for configuration management [Ansible]. These tools enable automated management of customizations to the default status and configuration of software. They are purpose-built domain-specific tools that can replace the scripting approach described above. Such systems provide checks and guarantees for applying changes that would be hard to write as shell scripts alone – just as a makefile handles builds more gracefully than a shell script. This approach manages configuration complexity as an environment grows in feature complexity. It may also allow an end-user to manage and reliably apply personal customizations across multiple versions of an environment over time. For BCE development, we felt Ansible added the least complexity amongst comparable tools. It may be used at build-time and also at run-time within the guest OS, *or from any other location with SSH access to the target being configured*. The only requirements for the target are an SSH server and a Python interpreter (Ansible is Python-based). Ansible execution is also more linear than some systems, which is a limitation, but also a simplification.

At this phase, however, the complexity of BCE doesn’t warrant contributors learning even a simple configuration management tool. The maintainer of the Software Carpentry VM, Matt Davis, has reported a similar observation. He has used another tool, Puppet, to provision the Software Carpentry VM, but will likely use shell scripts in the future. And as we will see below from the OSGeo project, it is perhaps easier to coordinate certain kinds of complexity with more commonly known tools like shell scripting.

While the syntax for each tool varies, the general concept is the same – one describes the desired machine state with a tool-specific language. After execution of this recipe – if you did a good job – the machine state is guaranteed to be how you’ve requested it to be. Unfortunately, all DevOps tools call their recipes something different. While the process certainly seems more like baking than, say, coaching a football team, Ansible calls its scripts “playbooks.” Alternate tools with similar functionality are Chef (which, unsurprisingly *does* call its scripts “recipes”),

Salt (also Python-based, and uses “states”), and Puppet (which uses “manifests”). With any of these, a great way to start learning would be to translate an existing configuration shell script into one of these tools.

Packer

Packer is used at build-time and enables creating identical machine images targeting multiple machine image formats [Packer]. For example, we generate a (mostly) uniformly configured BCE machine image in multiple formats including OVF for VirtualBox and AMI for AWS EC2. Packer coordinates many of the tools described above and below based on a JSON configuration file. This file specifies the Ubuntu ISO to install, a Debian Installer configuration file (which gets served over HTTP), and configures the installed OS by copying files and running a shell script. Packer can also readily use Ansible, Puppet, Chef, or Salt (and has a plugin system if you want to use something more exotic). Images can be built for many popular platforms, including a variety of local and cloud-based providers.

Packer made it possible for us to learn a relatively simple tool that executes the entire image-creation process as a single logical operation. Moreover, end users need have no knowledge of Packer. They can use the Amazon web console or the VirtualBox GUI with no concerns for the complexity at build time.

It is worth noting that while indexes are available for a variety of images (e.g, vagrantbox.es, the Docker index, and Amazon’s list of AMIs), we have encountered surprisingly little effort to publish consistent environment that allows one to readily migrate between platforms. This is, however, precisely the goal of BCE, and it’s enabled by Packer.

Vagrant

Vagrant is a run-time component that needs to be installed on the host OS of the end user’s laptop [Vagrant]. Like Packer, it is a wrapper around virtualization software that automates the process of configuring and starting a VM from a special *Vagrant box* image (Vagrant boxes may be created with any of the above tools). It is an alternative to configuring the virtualization software using the GUI interface or the system-specific command line tools provided by systems like VirtualBox or Amazon. Instead, Vagrant looks for a *Vagrantfile* which defines the configuration, and also establishes the directory under which the `vagrant` command will connect to the relevant VM. This directory is, by default, synced to the guest VM, allowing the developer to edit the files with tools on their host OS. From the command-line (under this directory), the user can start, stop, or ssh into the Vagrant-managed VM. It should be noted that (again, like Packer) Vagrant does no work directly, but rather calls out to those other platform-specific command-line tools.

The initial impetus for the BCE project came from a Vagrant-based project called “jiffylab” [j1]. With a single command, this project launches a VM in VirtualBox or on various cloud services. This VM provides isolated shell and IPython notebook through your web browser. But while Vagrant is conceptually very elegant (and cool), we are not currently using it for BCE. In our evaluation, it introduced another piece of software, requiring command-line usage before students were comfortable with it. Should a use-case arise, however, it would be trivial to create a “vagrant box” (a Vagrant-tuned virtual image) with our current approach using Packer. That said, other “data-science” oriented VMs have chosen Vagrant as their method of distribution [DSTb], [DSTk].

Currently, Vagrant is most useful for experienced developers to share environments with each other.

Docker

Docker is a platform to build, distribute, and run images built on top of Linux Containers (LXC) which provides a lightweight style of virtualization called containerization [Docker]. An important distinction of LXC-based containerization is that the guest OS and the host OS both run the same underlying Linux kernel.

At run-time Docker adds to this containerization a collection of tools to manage configuring and starting an instance in much the same way that Vagrant does for a virtualization environment. Images are created using a simple build script called a Dockerfile which usually runs a series of shell script commands which might even invoke a configuration management system such as Ansible.

Another feature of the platform is the management and distribution of the images built by docker, including incremental differences between images. Docker makes it possible (albeit in a rudimentary way) to track changes to the binary image in a manner similar to the way git allows you to track changes to source code. This also includes the ability to efficiently maintain and distribute multiple branches of binary images that may be derived from a common root.

Docker is also more than just a tool. It is a quickly growing community of open source and industry developers with a rapidly evolving ecosystem of tools built on core OS primitives. There is no clear set of best practices, and those that emerge are not likely to fit all the use cases of the academic community without us being involved in mapping the tools to our needs. However, providing better access to hardware with containers is an important and active research topic for performance [HPC].

Currently, Docker requires a Linux environment to host the Docker server. As such, it clearly adds *additional* complexity on top of the requirement to support a virtual machine. We also evaluated Docker as a way to potentially provide around 30 students access to a VM on a reasonably powered server with only 16GB of RAM. However, in our use-cases, we have full control of our Linux compute environment and existing methods of isolating users with permissions was less complex than using Docker, and of course allowed users to efficiently share all available physical RAM. Moreover, the default method of deploying Docker (at the time of evaluation) on personal computers was with Vagrant. This approach would then *also* add the complexity of using Vagrant. However, recent advances with *boot2docker* provide something akin to a VirtualBox-only, Docker-specific replacement for Vagrant that eliminates *some* of this complexity, though one still needs to grapple with the cognitive load of nested virtual environments and tooling.

OSGeo-Live: A Successful Common Environment

The OSGeo-Live VM is an example of a comprehensive geospatial compute environment with a vibrant community process. It provides a successful example of solving the problems of complex requirements described above – or in this case, perhaps more properly called “dependency hell”. Notably, the project uses none of the recent DevOps tools. OSGeo-Live is instead configured using simple and modular combinations of Python, Perl and shell scripts, along with clear install conventions and examples. Documentation is given high priority.

The VM project began around the same time as, and ultimately joined the Open Source Geospatial Foundation (OSGeo),

an international body modeled on the Apache Foundation [2g]. It started as a smaller open project that sought to build an “easy to try and use” software environment for spatial data applications. Initial efforts consisted of shell scripts to install core geospatial packages. These examples provided guides to the projects that were invited and ultimately contributed packages to the project. Many of these later contributors spoke English as a second language, further highlighting the importance of clear, working code examples. OSGeo-Live is not the only attempt at building such an environment, but it is a highly successful one. More than fifty open-source projects now contribute by actively maintaining and improving their own install scripts, examples and documentation.

Tool Sets

OSGeo-Live itself is not a “Linux distribution” per se, rather it relies on an apt-based ecosystem to handle the heavy-lifting of system updates and upgrades. This is a win, as updates are proven reliable over a very large Ubuntu community process, and project participants can concentrate on adding value to its featured components. Given the component architecture used to build the VM, individual software projects can be installed as-needed on a generic apt-enabled base.

A key component of the success of the overall project has been the availability of widely-known and reliable tools. Rather than require `.deb` installation packages for each project, OSGeo-Live chose to use a simple install script format, with ample examples. This choice proved crucial in the earliest stages, as an outside open-source project evaluating participation in the Live ISO could get started with fewer barriers to entry. Participating open-source projects already had install scripts built for Linux, so they could almost immediately adapt and iterate their own install scripts in a straightforward way, with the flexibility to use the tools they were already using, such as shell, Perl, or Python. Scripts may call package managers, and generally have few constraints (apart from conventions like keeping recipes contained to a particular directory). The project also maintains packages that support broader *kinds* of packages, such as web-based applications. In this case, OSGeo-Live provides a standard configuration for Apache, WSGI, and other components, along with a standard layout for projects that rely on this core. As a result, there is very little conflict among packages that share common resources. Some concerns, like port number usage, have to be explicitly managed at a global level. But the overhead of getting 50 projects to adopt a uniform configuration management tool would likely be much greater.

All recipes are currently maintained in a common subversion repository, using standardized asset hierarchies, including installation scripts [6g]. An OSGeo-Live specific report is maintained on the project trac ticketing system [10g]. And while OSGeo-Live primarily targets a live/bootable ISO, the scripts that are used to build that ISO provide a straightforward method for building OSGeo software in other contexts.

Community Awareness

The initial stages of the adoption of new technology include initial awareness and trialability [4g]. OSGeo-Live intentionally incorporates targeted outreach, professional graphic design and “easy to try” structure to build participation from both developers and end-users. An original project design goal was to provide tools to those doing geospatial fieldwork with limited resources around the globe, and who often lack advanced programming

and administration skills. In other words, a community was built around tools that the desired members already had.

Several years into the project, with a grant from the Australian government, a professional-level documentation project was initiated for a single-page overview and quick-start instructions for each application. Language internationalization was rendered more efficient, specifically to support local field work. Much later, a “percentage complete” graph for each human language group was added, making translation into a sort of competitive game. This translation has proven very successful. The project has facilitated collaboration across developer communities. For example, we have seen productive application of software developed by the U.S. military to environmental applications [Army].

Steps to Contribute

All build scripts are organized in the open, in source control [6g]. A new contributors FAQ is maintained via wiki [7g] for software projects, and for translation [8g]. At its core, the OSGeo-Live project uses common skills for system administration as opposed to more recent DevOps available, but it very much adopts a DevOps *philosophy*. Contributors pay particular attention to documenting each and every step, and standard approaches are encouraged across the project. Gamification also played a role in spurring useful documentation contributions. The low barrier to entry (allowing contributing projects to use skills they likely already have), combined with guidelines to ensure interoperability have led to OSGeo-Live becoming a standard way to evaluate and install software in the geospatial community.

BCE: The Berkeley Common Environment

The overarching, aspirational goal for the Berkeley Common Environment (BCE) is to make it *easy* to do the “right” thing (or hard to do “wrong” things), where “right” means you’ve managed to use someone else’s code in the manner that was intended. In particular, it allows for targeted instructions that can assume all features of BCE are present. BCE also aims to be stable, reliable, and reduce complexity more than it increases it.

More prosaically, to be useful in the cases described above, BCE provides simple things like a standard GUI text editor, and a command-line editor for when a GUI is not available. BCE pre-configures applications with sensible defaults (e.g., spaces for tab-stops are set up for `nano`). It also enables idiosyncratic features on different VM platforms, for example, enabling simple access to shared folders in VirtualBox and ensuring NFS functions properly on Amazon EC2. The environment is also configured to make minimal demands on underlying resources. For example, the BCE desktop is a solid color to minimize network utilization for remote desktop sessions, and efficient numerics libraries are configured.

BCE provides ready-made images for end-users, and the “recipe” for setting up the image using Packer is maintained on GitHub. Lists of Python packages are maintained in a separate requirements file, and all setup is done via a master Bash script. It is currently common for individuals to *only* distribute scripts, which requires all potential users to install and configure the relevant stack of DevOps tools. There are, however, free services for distributing images for particular tools (e.g., the Docker index), and services like Amazon can host AMIs for pennies a month. (For example, building on a free, existing EBS-backed AMI, one need only save a snapshot, with charges only for *changes* from the base AMI. One GB of extra tools onto a standard EBS-backed

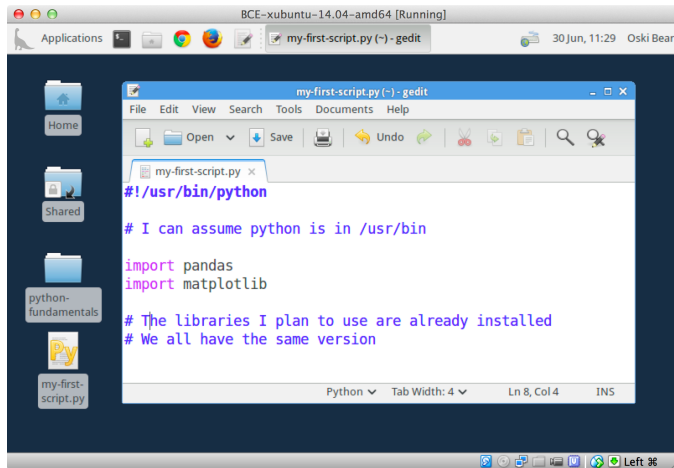


Fig. 1: The Berkeley Common Environment running in VirtualBox on OS X. The interface (and opportunities for confusion) are minimized. For example, all users have the same text editor available, and in particular, it's easy to configure common gotchas like spaces for tabs.

Ubuntu server AMI, currently costs <\$0.1 / GB-month to store.) We strongly recommend distributing a binary along with the recipe for any environment that includes novices in its audience.

Using the BCE

You can see what BCE currently looks like (in a relatively small window) in Figure 1. Throughout various iterations, students have found working on a BCE VM to be confusing and counterproductive to being incredibly useful and efficient – strong evidence that the details matter. It seems critical both to provide a rationale for the use of VMs (i.e., explaining how a standard, “pixel-identical” environment speeds instruction), and also a smooth initial experience. Thus, we’ve worked to make BCE easy for students, researchers, and instructors. Simple instructions are provided on our site for things like opening a terminal (including a description of what the terminal icon looks like). However, for an experienced programmer, the environment should be obvious to navigate.

In our experience, some students will not be able to run the VM while others have difficulty getting regular access to a stable network connection (though fortunately, almost never both!). So, consistency across server and local versions of the environment is critical to effectively support students with either of these difficulties.

If you’re using VirtualBox, we require a 64-bit CPU with support for 64-bit virtualization (note that some 32-bit *operating systems* will support this on some hardware). A reasonable minimum of RAM is 4GB. The full instructions for importing BCE from an OVA image into Virtualbox are available on our project website [BCEVB]. After starting the VM – a process that can be done entirely with the mouse – a user will have all the software installed as part of BCE, including IPython, RStudio, and useful packages.

If you’re using BCE on EC2, even a micro instance is sufficient for basic tasks. Again, complete instructions are provided on the BCE website [BCEAMI]. In brief, you can find our image (AMI) in the public list. You can readily launch in instance, and get instructions on connecting via the EC2 console.

Communicating with the maintainers of the BCE project

All development occurs in the open in our GitHub repository. This repository currently also hosts the project website, with links to all BCE materials. We provide channels for communication on bugs, desired features, and the like via the repository and a mailing list (also linked from the project page), or if a user is comfortable with it, via the GitHub issue tracker. BCE will be clearly versioned for each semester, and versions will not be modified, except for potential bugfix releases.

Contributing to the BCE project

BCE provides a fully scripted (thus, reproducible) workflow that creates the standard VM/image. If the appropriate software is installed, the recipe should run reliably. However, you should generally not need to build the binary VM for BCE for a given semester. If you wish to customize or extend BCE, the best way to do this is by simply writing a shell script that will install requirements properly in the context of BCE (for a complex example, see our `bootstrap-bce.sh` script [boot]). Much as with OSGeo-Live, we have chosen our approach to provisioning to be relatively simple for users to understand. It is our goal for instructors or domain experts to be able to easily extend the recipe for building BCE VMs or images. If not, that’s a bug!

As described above, while we have experimented with Docker, Vagrant, and Ansible for setting up the various BCE images (and evaluated even more tools), the only foundationally useful tool for our current set of problems has been Packer. Packer runs a shell script that uses standard installation mechanisms like `pip` and `apt-get` to complete the setup of our environment. Of central importance, Packer does not require end-users to install or understand any of the current crop of DevOps tools – it operates solely at build time. However, should the need arise, Packer will readily target Vagrant, Docker, and many other targets, and we are not opposed to adopting other tooling.

Conclusion

By merely using recent DevOps tools, *you* arrive at the cutting edge of DevOps for the scientific community. Your collaborators and students likely won’t have needed concepts, so extra care should be taken to make your tooling accessible. Where appropriate, use tools that your collaborators already know – shell, scripting, package management, etc. That said, technologies that allow efficient usage of available hardware, like Docker, stand to provide substantial savings and potential for re-use by researchers with less direct access to capital.

So, let’s be intentional about creating and using environments that are broadly accessible. Let’s follow the DevOps philosophy of being transparent and explicit about our choices and assumptions. That *doesn’t* have to mean “using the latest tools” – a simple text file or even a PDF can provide ample explanation that a human can understand, along with a simple reference script (in shell or Python). In this paper, we’ve made fairly strong recommendations based on what we are actually using (we are eating our own dogfood!). A novice user can access BCE using only a few GUI operations on their laptop, or the Amazon Web Console. As we’ve seen with OSGeo-Live, the simple tools we’ve chosen make it easy for our collaborators (instructors or researchers) to understand. This standard reference allows us to return focus on the interesting bits of developing code and *doing science*.

BCE currently provides a standard reference, built with an easily understood recipe, that eliminates the complexity of describing how to run a large variety of projects across a wide variety of platforms. We can now target our instruction to a single platform. The environment is easy to deploy, and should provide identical results across any base platform – if this is not the case, it's a bug! This environment is already available on VirtualBox and Amazon EC2, and is straightforward to provision for other environments. We welcome loose collaboration in the form of forks that are specialized for other institutions, and eventually, perhaps standardizing across institutions.

REFERENCES

- [BCE] <http://collaboratool.berkeley.edu>
- [OSGL] <http://www.osgeo.org/>
- [BCEVB] <http://collaboratool.berkeley.edu/using-virtualbox.html>
- [BCEAMI] <http://collaboratool.berkeley.edu/using-ec2.html>
- [Ubuntu] <https://help.ubuntu.com/14.04/serverguide/serverguide.pdf>
- [images] http://docs.openstack.org/image-guide/content/ch_introduction.html
- [Ansible] <http://www.ansible.com/about>
- [Packer] <http://www.packer.io/intro>
- [Vagrant] <http://www.vagrantup.com/about.html>
- [Docker] <http://www.docker.com/whatisdocker/>
- [HPC] M. G. Xavier, M. V. Neves, F. D. Rossi, T. C. Ferreto, T. Lange, and C. A. De Rose, "Performance evaluation of container-based virtualization for high performance computing environments," in *the 21st Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, 2013, pp. 233–240.
- [SWC] G Wilson, "Software Carpentry: lessons learned," *F1000Research*, 2014.
- [jl] <http://github.com/ptone/jiffylab>
- [DSTb] <http://datasciencetoolbox.org/>
- [DSTk] <http://www.datasciencetoolkit.org/>
- [UDI] <https://help.ubuntu.com/14.04/installation-guide/i386/apb.html>
- [2g] <http://www.osgeo.org/content/foundation/about.html>
- [4g] E M. Rogers, *Diffusion of Innovations*, 5th ed. New York: Free Press, 2003.
- [6g] <http://svn.osgeo.org/osgeo/livedvd>
- [7g] http://wiki.osgeo.org/wiki/Live_GIS_Add_Project
- [8g] http://wiki.osgeo.org/wiki/Live_GIS_Translate
- [10g] <http://trac.osgeo.org/osgeo/report/10>
- [Army] Army Corps of Engineers, "Army Corps of Engineers Wetlands Regulatory program," presented at the FOSS4G, 2007.
- [boot] <https://github.com/dlab-berkeley/collaboratool/blob/master/provisioning/bootstrap-bce.sh>

Measuring rainshafts: Bringing Python to bear on remote sensing data

Scott Collis^{§*}, Scott Giangrande^{**}, Jonathan Helmus[§], Di Wu^{||}, Ann Fridlind[¶], Marcus van Lier-Walqui[¶], Adam Theisen[‡]

<http://www.youtube.com/watch?v=1D0aTToHrCY>



Abstract—Remote sensing data is complicated, very complicated! It is not only geometrically tricky but also, unlike in-situ methods, indirect as the sensor measures the interaction of the scattering media (eg raindrops) with the probing radiation, not the geophysics. However the problem is made tractable by the large number of algorithms available in the Scientific Python community. While SciPy provides many helpful algorithms for signal processing in this domain, a full software stack from highly specialized file formats from specific sensors to interpretable geospatial analysis requires a common data model for active remote sensing data that can act as a middle layer. This paper motivates this work by asking: How big is a rainshaft? What is the natural morphology of rainfall patterns and how well is this represented in fine scale atmospheric models. Rather than being specific to the domain of meteorology, we will break down how we approach this problem in terms of the tools used from numerous Python packages to read, correct, map and reduce the data into a form better able to answer our science questions. This is a "how" paper, covering the Python-ARM Radar Toolkit (Py-ART) containing signal processing using linear programming methods and mapping using k-d trees. We also cover image analysis using SciPy's ndimage sub-module and graphics using matplotlib.

Index Terms—Remote sensing, radar, meteorology, hydrology

Introduction

RADARs (RADio Detection And Ranging, henceforth radars) specialized to weather applications do not measure the atmosphere, rather, the instrument measures the interaction of the probing radiation with the scattering medium (nominally cloud or precipitation droplets or ice particulate matter). Therefore, in order to extract geophysical insight, such as the relationship between large scale environmental forcing and heterogeneity of surface precipitation patterns, a complex application chain of algorithms needs to be set up.

This paper briefly outlines a framework, using a common data model approach, for assembling such processing chains: the Python-ARM Radar Toolkit, Py-ART [Heistermann2014]. This

paper also provides an example application: using rainfall maps to objectively evaluate the skill of fine scale models in representing precipitation morphology.

The data source: scanning centimeter wavelength radar

Rainfall can occur at many different scales. From small, discrete storm cells at scales of 10's of kilometers to large scale tropical systems such as hurricanes which cover 100's to 1000's of kilometers. Some complex systems can contain many scales and in order to understand this spatial complexity of precipitating cloud systems a sensor is required that can collect spatially diverse data. Radars emit a spatially discrete pulse of radiation with a particular beamwidth and pulse length. A gated receiver detects the backscattered signal and calculates a number of measurements based on the radar spectrum (the power as a function of phase delay which is due to the motion of the scattering medium relative to the antenna). These moments include radar reflectivity factor Z_e , radial velocity of the scattering medium v_r , and spectrum width w . Polarimetric radars transmit pulses with the electric field vector horizontal to the earth's surface as well as vertical to the earth's surface. These radars can give a measure of the anisotropy of the scattering medium with measurements including differential reflectivity Z_{DR} , differential phase difference ϕ_{dp} and correlation coefficient ρ_{HV} . The data is laid out on a time/range grid with each ray (time step) having an associated azimuth and elevation. Data presented in this paper are from 4 ARM [Mather2013] radar systems: One C-Band (5 cm wavelength) and three X-Band (3 cm wavelength) radars as outlined in table 1.

These instruments are arranged as show in figure 1.

The Python ARM Radar Toolkit: Py-ART

Radar data comes in a variety of binary formats but the content is essentially the same: A time-range array for each radar moment along with data describing the pointing and geolocating of the platform. For mobile radar the platform's motion must also be described in the file. Py-ART takes a common data model approach, carefully designing the data containers and mandating that functions and methods accept the container as an argument and return the same data structure. The common data model for radar data in Py-ART is the Radar class which stores data and metadata in Python dictionaries in a particular instance's attributes. Data is

* Corresponding author: scollis@anl.gov

§ Environmental Sciences Division, Argonne National Laboratory.

** Atmospheric Sciences, Brookhaven National Laboratory.

|| NASA Goddard Space Flight Center.

¶ NASA Goddard Institute of Space Sciences.

‡ University of Oklahoma, Cooperative Institute for Mesoscale Meteorological Studies, ARM Climate Research Facility Data Quality Office.

	X-SAPR	C-SAPR
Frequency	9.4 GHz	6.25GHz
Transmitter	Magnetron	Magnetron
Power	200kW	350kW
Gate spacing	50m	120m
Maximum Range	40km	120km
Beam width	1°	1°
Polar. mode	Simul. H/V	Simul. H/V
Manufacturer	Radtec	Adv. Radar Corp.
Native format	Iris Sigmoid	NCAR MDV

TABLE 1: ARM radar systems used in this paper.

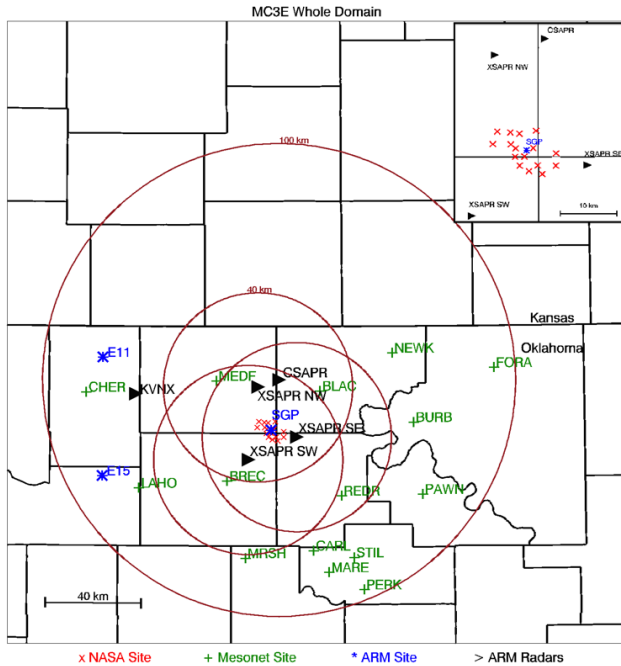


Fig. 1: Arrangement of radars around the ARM Southern Great Plains Facility from [Giangrande2014].

stored in a NumPy array in the 'data' key of the dictionary. For example:

```
print xnw_radar.fields.keys()
['radar_echo_classification',
'corrected_reflectivity',
'differential_phase',
'cross_correlation_ratio',
'normalized_coherent_power',
'spectrum_width',
'total_power', 'reflectivity',
'differential_reflectivity',
'specific_differential_phase',
'veLOCITY',
'corrected_differential_reflectivity']
print xnw_radar.fields['reflectivity'].keys()
['_FillValue', 'coordinates', 'long_name',
'standard_name', 'units', 'data']
print xnw_radar.fields['reflectivity']['long_name']
Reflectivity
print xnw_radar.fields['reflectivity']['data'].shape
(8800, 801)
```

The `xnw_radar` has a variety of fields, including 'reflectivity' with the numerical moment data stored in the 'data' key with 8800 time steps and 801 range gates. Data on instrument pointing is stored in

Format name	Example radar system(s)	Note
CF-Radial	NCAR SPOL, ARM Cloud Radars	Output format
UF	Lots of legacy data	Via RSL
Lassen	BoM CPOL in Darwin, Australia	Via RSL
IRIS Sigmoid	ARM X-SAPR	Native
NCAR MDV	ARM C-SAPR	Native
GAMIC	European radar network	Native
WSR-88D	USA operational network	Native
CHILL	NSF funded deployable S-Band	Native

TABLE 2: Py-ART formats.

`x_nw.azimuth` and `x_nw.elevation` attributes while the center point of each range gate is stored in `x_nw.range`. Again these attributes are dictionaries with data stored in the 'data' key. Functions in Py-ART can append fields or modify data in existing fields (rare).

The vital key is a 'Babelfish' layer which ingests a variety of formats into the common data model. Currently table 2 outlines the formats which are compatible with Py-ART. A number of these formats are available via a Cython wrapper around NASA's Radar Software Library.

There is also active development on supporting NOAA NOXP and NASA D3R radars. Py-ART supports a single output format for radial geometry radar data which is, CF-Radial. CF-Radial is a NetCDF based community format on which the common data model in Py-ART is based on.

Py-ART forms part of an ecosystem of open source radar applications, many of which are outlined in [Heistermann2014]. A key challenge for the radar community is reaching consensus on data transport layers so that an application chain can be built using multiple applications. In terms of the rest of the Scientific python ecosystem, Py-ART brings the data into Python in a very simple way so users can simply and quickly get to doing Science.

Pre-mapping corrections and calculations

Once raw data is collected there is often a number of processing steps that need to be performed. In our case this includes:

- Correcting false Azimuth readings in the Northwest X-Band system.
- Cleaning data of undesirable components such as multiple trips, clutter and non-meteorological returns.
- Processing the raw ϕ_{DP} and extracting the component due to rain water content by using a linear programming technique to fit a profile which mandates positive gradient, see [Giangrande2013].
- Using reflectivity and ϕ_{DP} to retrieve attenuation (in dBZ/km) due to rainwater path.
- Using the techniques outlined in [Ryzhkov2014] to retrieve rainfall rate (in mm/hr) from attenuation.

These are all outlined in the first of the three notebooks which accompany this manuscript: <http://nbviewer.ipython.org/github/scollis/notebooks/tree/master/scipy2014/>. Each process either appends a new field to the Radar instance or returns a field dictionary which can then be added to the instance. Py-ART also comes with visualization methods allowing for the conical (or Plan Position Indicator, PPI) scan to be plotted and geolocated

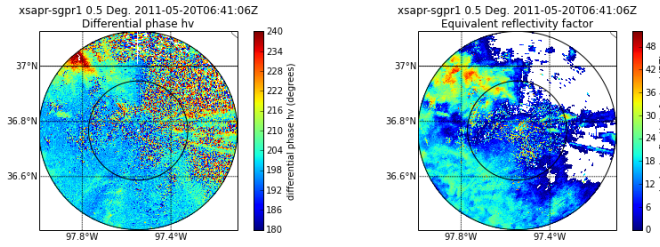


Fig. 2: Raw reflectivity factor and polarimetric phase difference from the lowest (0.5 degree) tilt.

using matplotlib and Basemap. An example plot of raw ϕ_{DP} and reflectivity is shown in figure 2.

The code necessary to create this plot:

```
fields_to_plot = ['differential_phase',
                 'reflectivity']
ranges = [(180, 240), (0, 52)]
display = pyart.graph.RadarMapDisplay(xnw_radar)

nplots = len(fields_to_plot)
plt.figure(figsize=[7 * nplots, 4])
for plot_num in range(nplots):
    field = fields_to_plot[plot_num]
    vmin, vmax = ranges[plot_num]
    plt.subplot(1, nplots, plot_num + 1)
    display.plot_ppi_map(field, 0, vmin=vmin,
                        vmax=vmax, lat_lines=np.arange(20, 60, .2),
                        lon_lines=np.arange(-99, -80, .4),
                        resolution='1')
    display.basemap.drawrivers()
    display.basemap.drawcountries()
    display.plot_range_rings([20, 40])
```

Here, a RadarMapDisplay instance is instantiated by providing a Radar object which is insensitive to the data source. The sample plotting routines can be used to plot data ingested from any of the formats which Py-ART supports.

Mapping to a Cartesian grid

Radars sample in radial coordinates of elevation, azimuth and range. Mathematics for atmospheric phenomena are greatly simplified on Cartesian and Cartesian-like (eg pressure surfaces) grids. Therefore the raw and processed data in the Radar object often need to be mapped onto a regular grid. In the field, this is known as "Objective analysis" (see, for example [Trapp2000]). In this paper we use a technique known as Barnes analysis [Barnes1964] which is an inverse distance weighting, sphere of influence based technique. For each grid point in the Cartesian grid a set of radar gates within a radius of influence are interpolated using the weighting function:

$$W(r) = e^{\frac{-r_{infl}^2}{2.0 * r^2}}$$

where r is the distance from the grid point and r_{infl} is the search radius of influence. A brute force method for performing this mapping would be to calculate the distance from each Cartesian point to each radar gate to find those within the radius of influence, a method which scales as $n * m$ where n is the number of points in the grid and m the number of gates in the radar volume. With a typical grid being 200 by 200 by 37 points and a modern radar having on the order of 8000 time samples and 800 range gates this quickly becomes intractable. A better method is to store the radar gates in a k-d tree or related data structure. This reduces the search to an order $n * \log(m)$ problem. This method is implemented in Py-ART.

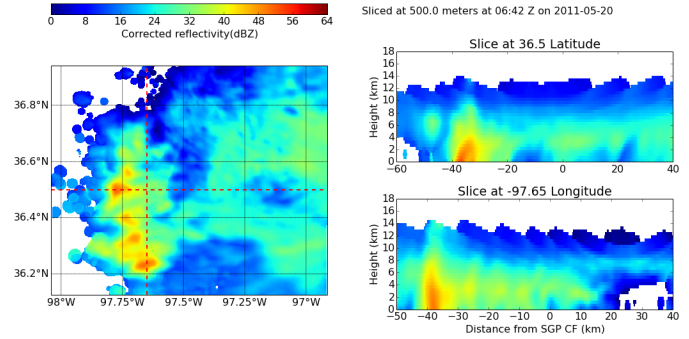


Fig. 3: Single C-Band reflectivity factor field.

In addition a variable radius of influence algorithm is implemented which analyzes the radar volume coverage pattern and deduces an optimized r_{infl} at each grid point. Unlike many other objective analysis codes Py-ART implementation can operate on multiple Radar objects simultaneously, treating the radar gates as a cloud of points. This allows the merging of multiple radar data sets. The method is simple to invoke, for example the code snippet:

```
mesh_mapped_x = pyart.map.grid_from_radars(
    (xnw_radar, xsw_radar, xse_radar),
    grid_shape=(35, 401, 401),
    grid_limits=((0, 17000), (-50000, 40000),
                (-60000, 40000)),
    grid_origin=(36.57861, -97.363611),
    fields=['corrected_reflectivity', 'rain_rate_A',
           'reflectivity'])
```

will map the gates in the three Radar objects (in this case the three ARM X-Band systems in figure 1) to a grid that is (z,y,x) = (35, 401, 401) points with a domain of 0 to 17 km in altitude, -50 to 40 km in meridional extend and -60 to 40 km in zonal extent. The method returns a Grid object which follows a similar layout to a Radar object: fields are stored in the fields attribute, geolocation data in the axes attribute with the numerical data found in the 'data' key of the dictionaries.

Again, as with the Radar object Py-ART has a menu of available routines to visualize data contained in Grid objects as well as an input output layer that can inject CF-compliant netCDF grids and write Grid object out to a CF-complaint file for future analysis and distribution.

For example figure 3 shows a slice through mapped reflectivity from the ARM C-SAPR at 500 m and cross sections at 36.5 N degrees latitude and -97.65 E longitude.

In the vertical cross sections clear artifacts can be seen due to the poor sampling. Figure 4 shows the same scene but using a grid created from three X-Band radars in a network. In both figures the radar data are mapped onto a grid with 225 m spacing.

It is clear that more fine scale detail is resolved due to the rain systems being closer to any given radar in the X-Band network grid. In addition, due to the higher density of high elevation beams (essentially a "web" of radar beams sampling the convective anvil) sampling artifacts are greatly reduced and finer details aloft are able to be studied.

Mesh mapping only works for "specific" measurements, ie not integrated measurements like ϕ_{DP} or directionally dependent moments like v_r . One measurement that can be mapped is our retrieved rain rate.

Figures 5 and 6 show mappings for rain rate using just the C-Band measurement and X-Band network respectively. Again the

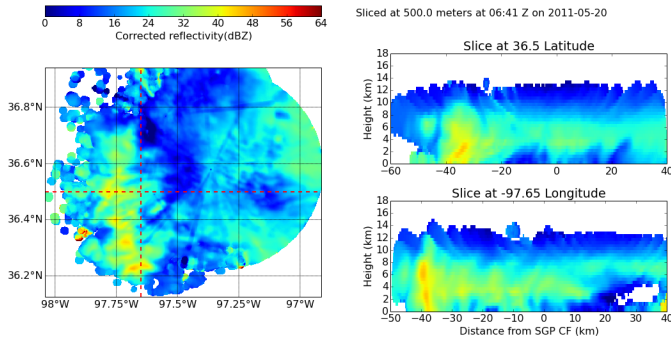


Fig. 4: Reflectivity factor mapped from a network of X-Band radars.

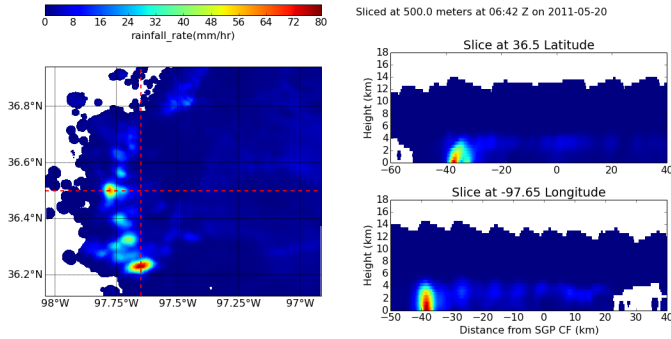


Fig. 5: Single C-Band rainfall field.

mesh map of the X-Band retrieval shows very fine detail resolving (in a volumetric dataset) fall streak patterns. The maxima near 4 km (just below the freezing level) is due to melting particles. The rainfall retrieval has a cut off at the sounding determined freezing level but the "bright band" can extend some depth below this. Future work will entail using polarimetric measurements to determine where there is only pure liquid returns and conditionally apply the rainfall retrieval to those positions.

Spatial distribution of rainfall: a objective test of fine scale models

Previous sections have detailed the correction, retrieval from and mapping to a Cartesian grid of radar data. The last section showed enhanced detail can be retrieved by using a network of radars. The question remains: how can the detail in rain fields be objectively compared? Can parameters derived from radar data be compared to those calculated from forecast models? The meshes generated

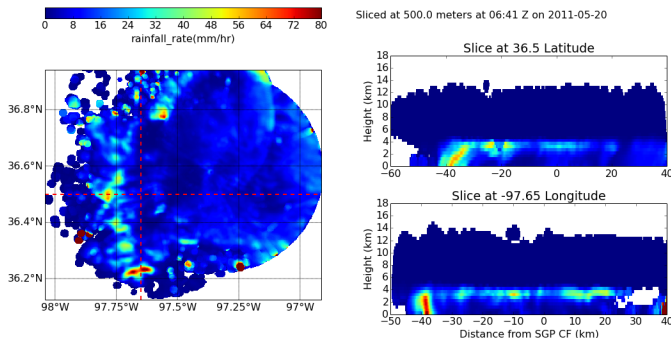


Fig. 6: Rainfall from a network of X-Band systems.

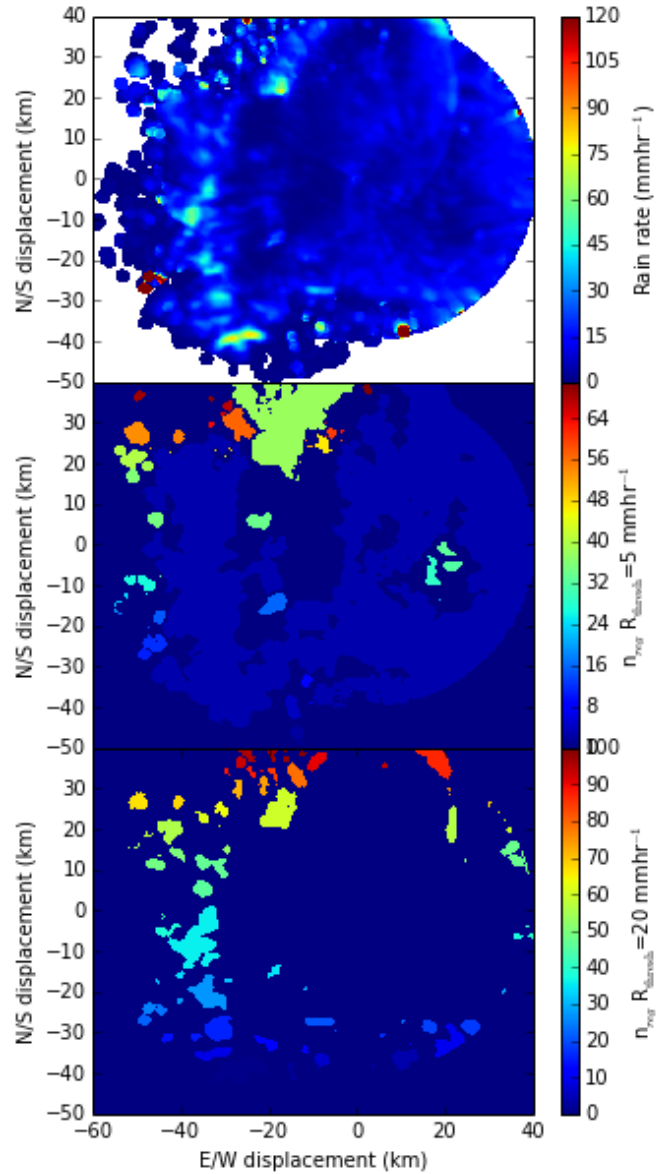


Fig. 7: An example of figure segmentation using scipy.ndimage.label.

using the mapping techniques previously discussed can be treated like image data for which a number of packages exist for analysis.

Measuring rainshafts using SciPy's ndimage subpackage

A simple technique for documenting the features present in an image is to partition it into segments which are above a certain threshold and calculate the number of segments, their accumulated area and the mean rainfall across the segment. The ndimage subpackage in SciPy is perfect for accomplishing this. Figure 7 shows the use of scipy.ndimage.label to segment regions above 5 and 20mm/h.

The code is very simple, for a given rain rate it creates a "black and white" image with whites above the threshold point and the black below, then scipy.ndimage.label segments the regions into a list of regions from which metrics can be calculated:

```
def area_anal(pixel_area, rr_x, rain_rates):
    A_rainrate = np.zeros(rr_x.shape)
    N_rainrate = np.zeros(rr_x.shape)
    Rm_rainrate = np.zeros(rr_x.shape)
```

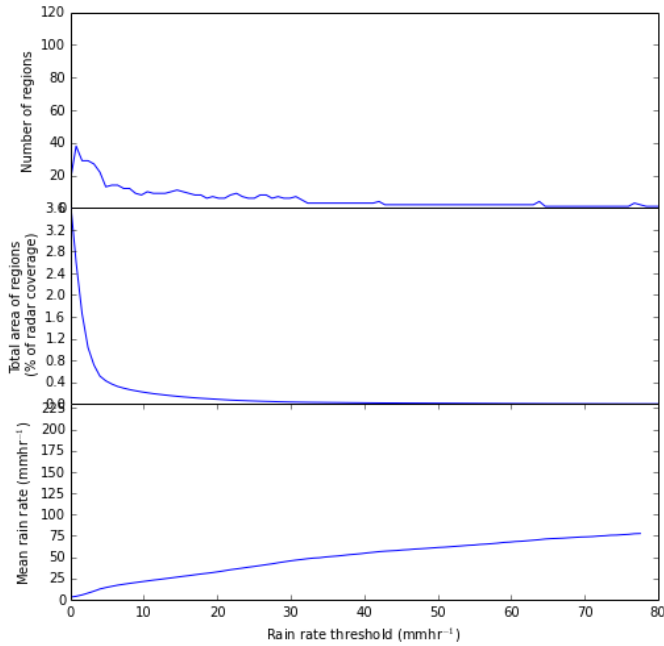



Fig. 8: Number of regions, region covered and mean rain rate as a function of rain rate threshold for a rainmap produced by a single C-Band system.

```

for i in range(len(rr_x)):
    b_fld = np.zeros(rain_rates.shape)
    b_fld[rain_rates > rr_x[i]] = 1.0
    regions, N_rainrate[i] = ndimage.label(b_fld)
    try:
        A_rainrate[i] = (len(np.where(
            regions > 0.5)[0]) *
            pixel_area)
        Rm_rainrate[i] = rain_rates[
            np.where(regions > 0.5)].mean()
    except IndexError:
        A_rainrate[i] = 0.0
        Rm_rainrate[i] = 0.0
return N_rainrate, A_rainrate, Rm_rainrate

```

This produces plots for the X-Band mesh as seen in 9 and single C-Band systems in 8.

The results presented in this paper show that the rainfall field for this case is under-resolved when observed by a single C-Band system. While we have not established that a network of X-Band systems fully resolve the spatial complexity of the rainfall field it clearly shows more detail, especially at higher altitudes.

Future work will focus on establishing limits to spatial complexity and understanding how large scale forcing (instability, moisture etc) influence complexity. In addition we will be applying this technique to fine scale model data as an "observational target" for the model to achieve. That is the methods outlined in this paper can be used as a simple optimization metric which can be used when adjusting the parameters in a model.

Conclusions

This paper has covered the pipeline for proceeding from raw radar measurements through quality control and geophysical retrieval to mapping and finally to the extraction of geophysical insight. The simple conclusion is that, with careful processing, a network of X-Band radars can resolve finer details than a single C-Band radar. More importantly, finer details exist. The paper also presents

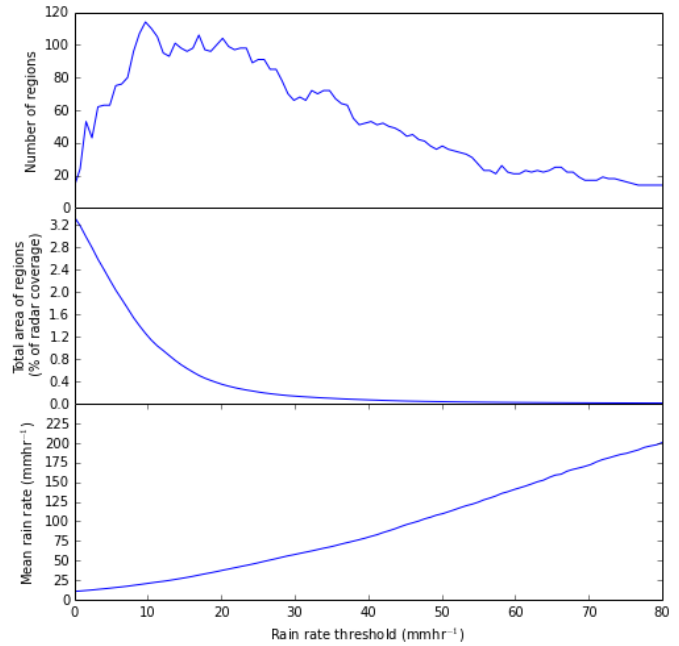


Fig. 9: Number of regions, region covered and mean rain rate as a function of rain rate threshold for a rainmap produced by a network of X-Band systems.

a very simple, image processing based technique to take the "morphological finger print" of rainfall maps. This technique can be used on both remotely sensed and numerically modeled data providing a objective basis for model assessment.

Acknowledgements

Dr. Giangrande's work is supported by the Climate Science for a Sustainable Energy Future (CSSEF) project of the Earth System Modeling (ESM) program in the DOE Office of Science. Argonne National Laboratory's work was supported by the U.S. Department of Energy, Office of Science, Office of Biological and Environmental Research (OBER), under Contract DE-AC02-06CH11357. The work has also been supported by the OBER of the DOE as part of the ARM Program. Adam Theisen's work was supported by Battelle – Pacific Northwest National Laboratory, contract number 206248, and his home institution, CIMMS, is supported by NOAA/Office of Oceanic and Atmospheric Research under NOAA-University of Oklahoma Cooperative Agreement #NA11OAR4320072, U.S. Department of Commerce. The authors wish to thank Dr. Alexander Ryzhkov for support on implementation of specific attenuation-based rainfall methods. We would also like to thank the reviewers of this paper, James Bergstra and Terry Letsche.

REFERENCES

- [Heistermann2014] Heistermann, M., S. Collis, M. J. Dixon, S. E. Giangrande, J. J. Helmus, B. Kelley, J. Koistinen, D. B. Michelson, M. Peura, T. Pfaff and D. B. Wolff, 2014: The Promise of Open Source Software for the Weather Radar Community. *Bull. Amer. Meteor. Soc.*, **In Press**.
- [Mather2013] Mather, J. H., and J. W. Voyles, 2012: The Arm Climate Research Facility: A Review of Structure and Capabilities. *Bull. Amer. Meteor. Soc.*, **94**, 377–392, doi:10.1175/BAMS-D-11-00218.1.

- [Giangrande2014] Giangrande, S. E., S. Collis, A. K. Theisen, and A. Tokay, 2014: Precipitation Estimation from the ARM Distributed Radar Network During the MC3E Campaign. *J. Appl. Meteor. Climatol.*, doi:10.1175/JAMC-D-13-0321.1. <http://journals.ametsoc.org/doi/abs/10.1175/JAMC-D-13-0321.1>
- [Giangrande2013] Giangrande, S. E., R. McGraw, and L. Lei, 2013: An Application of Linear Programming to Polarimetric Radar Differential Phase Processing. *Journal of Atmospheric and Oceanic Technology*, **30**, 1716–1729, doi:10.1175/JTECH-D-12-00147.1.
- [Ryzhkov2014] Ryzhkov, A. V., M. Diederich, P. Zhang, C. Simmer, 2014: Potential utilization of specific attenuation for rainfall estimation, mitigation of partial beam blockage, and radar networking. Submitted, *J. Atmos. Oceanic Technol.*, **in press**.
- [Trapp2000] Trapp, R. J., and C. A. Doswell, 2000: Radar Data Objective Analysis. *Journal of Atmospheric and Oceanic Technology*, **17**, 105–120, doi:10.1175/1520-0426(2000)017<0105:RDOA>2.0.CO;2.
- [Barnes1964] Barnes, S. L., 1964: A Technique for Maximizing Details in Numerical Weather Map Analysis. *Journal of Applied Meteorology*, **3**, 396–409, doi:10.1175/1520-0450(1964)003<0396:ATFMDI>2.0.CO;2.

Teaching numerical methods with IPython notebooks and inquiry-based learning

David I. Ketcheson^{‡*}

<http://www.youtube.com/watch?v=OaP6LiZuaFM>

Abstract—A course in numerical methods should teach both the mathematical theory of numerical analysis and the craft of implementing numerical algorithms. The IPython notebook provides a single medium in which mathematics, explanations, executable code, and visualizations can be combined, and with which the student can interact in order to learn both the theory and the craft of numerical methods. The use of notebooks also lends itself naturally to inquiry-based learning methods. I discuss the motivation and practice of teaching a course based on the use of IPython notebooks and inquiry-based learning, including some specific practical aspects. The discussion is based on my experience teaching a Masters-level course in numerical analysis at King Abdullah University of Science and Technology (KAUST), but is intended to be useful for those who teach at other levels or in industry.

Index Terms—IPython, IPython notebook, teaching, numerical methods, inquiry-based learning

Teaching numerical methods

Numerical analysis is the study of computational algorithms for solving mathematical models. It is used especially to refer to numerical methods for approximating the solution of continuous problems, such as those involving differential or algebraic equations. Solving such problems correctly and efficiently with available computational resources requires both a solid theoretical foundation and the ability to write and evaluate substantial computer programs.

Any course in numerical methods should enable students to:

- 1) **Understand** numerical algorithms and related mathematical concepts like complexity, stability, and convergence
- 2) **Select** an appropriate method for a given problem
- 3) **Implement** the selected numerical algorithm
- 4) **Test** and debug the numerical implementation

In other words, students should develop all the skills necessary to go from a mathematical model to reliably-computed solutions. These skills will allow them to select and use existing numerical software responsibly and efficiently, and to create or extend such software when necessary. Usually, only the first of the objectives above is actually mentioned in the course syllabus, and in some courses it is the only one taught. But the other three objectives

are likely to be of just as much value to students in their careers. The last two skills are practical, and teaching them properly is in some ways akin to teaching a craft. Crafts are not generally taught through lectures and textbooks; rather, one learns a craft by *doing*.

Over the past few years, I have shifted the emphasis of my own numerical courses in favor of addressing all four of the objectives above. In doing so, I have drawn on ideas from inquiry-based learning and used both Sage worksheets and IPython notebooks as an instructional medium. I've found this approach to be very rewarding, and students have told me (often a year or more after completing the course) that the hands-on mode of learning was particularly helpful to them.

The notebooks used in my course for the past two years are available online:

- 2013 course: <https://github.com/ketch/finite-difference-course>
- 2013 course: <https://github.com/ketch/AMCS252>

Please note that these materials are not nearly as polished as a typical course textbook, and some of them are not self-contained (they may rely strongly on my unpublished course notes). Nevertheless, I've made them publicly available in case others find them useful. For more context, you may find it helpful to examine the [course syllabus](#). You can also examine the [notebooks for my short course on hyperbolic PDEs](#), which are more self-contained.

Inquiry-based learning

The best way to learn is to do; the worst way to teach is to talk. --P. R. Halmos [[Hal75](#)]

Many great teachers of mathematics (most famously, R.L. Moore) have argued against lecture-style courses, in favor of an approach in which the students take more responsibility and there is more in-class interaction. The many related approaches that fit this description have come to be called *inquiry-based learning* (IBL). In an inquiry-based mathematics course, students are expected to find the proofs for themselves -- with limited assistance from the instructor. For a very recent review of what IBL is and the evidence for its effectiveness, see [[Ern14a](#)], [[Ern14b](#)] and references therein. If an active, inquiry-based approach is appropriate for the teaching of theoretical mathematics, then it seems even more appropriate for teaching the practical craft of computational mathematics.

A related notion is that of the *flipped classroom*. It refers to a teaching approach in which students read and listen to recorded lectures outside of class. Class time is then used not for lectures

* Corresponding author: david.ketcheson@kaust.edu.sa
[‡] King Abdullah University of Science and Technology

but for more active, inquiry-based learning through things like discussions, exercises, and quizzes.

The value of practice in computational mathematics

Too often, implementation, testing, and debugging are viewed by computational mathematicians as mundane tasks that anyone should be able to pick up without instruction. In most courses, some programming is required in order to complete the homework assignments. But usually no class time is spent on programming, so students learn it on their own -- often poorly and with much difficulty, due to the lack of instruction. This evident disdain and lack of training seem to mutually reinforce one another. I believe that implementation, testing, and debugging are essential skills for anyone who uses or develops numerical methods, and they should be also taught in our courses.

In some situations, a lack of practical skills has the same effect as a lack of mathematical understanding. Students who cannot meaningfully test their code are like students who cannot read proofs: they have no way to know if the claimed results are correct or not. Students who cannot debug their code will never know whether the solution blows up due to an instability or due to an error in the code.

In many cases, it seems fair to say that the skills required to implement state-of-the-art numerical algorithms consists of equal parts of mathematical sophistication and software engineering. In some areas, the development of correct, modular, portable implementations of proposed algorithms is as significant a challenge as the development of the algorithms themselves. Furthermore, there are signs that numerical analysts need to move beyond traditional flop-counting complexity analysis and incorporate more intricate knowledge of modern computer hardware in order to design efficient algorithms for that hardware. As algorithms become increasingly adapted to hardware, the need for implementation skills will only increase.

Perhaps the most important reason for teaching implementation, testing, and debugging is that these skills can and should be used to reinforce the theory. The student who learns about numerical instability by reading in a textbook will forget it after the exam. The student who discovers numerical instability by implementing an apparently correct (but actually unstable) algorithm by himself and subsequently learns how to implement a stable algorithm will remember and understand it much better. Similarly, implementing an explicit solver for a stiff problem and then seeing the speedup obtained with an appropriate implicit solver makes a lasting impression.

It should be noted that many universities have courses (often called "laboratory" courses) that do focus on the implementation or application of numerical algorithms, generally using MATLAB, Mathematica, or Maple. Such courses may end up being those of most lasting usefulness to many students. The tools and techniques discussed in this article could very aptly be applied therein. Unfortunately, these courses are sometimes for less credit than a normal university course, with an attendant reduction in the amount of material that can be covered.

Hopefully the reader is convinced that there is some value in using the classroom to teach students more than just the theory of numerical methods. In the rest of this paper, I advocate the use of inquiry-based learning and IPython notebooks in full-credit university courses on numerical analysis or numerical methods. As we will see, the use of IPython notebooks and the teaching of the craft of numerical methods in general lends itself naturally

to inquiry-based learning. While most of the paper is devoted to the advantages of this approach, there are some significant disadvantages, which I describe in the *Drawbacks* section near the end.

Teaching with the IPython notebook

Python and IPython

The teacher of numerical methods has several choices of programming language. These can broadly be categorized as

- specialized high-level interpreted languages (MATLAB, Mathematica, Maple)
- general-purpose compiled languages (C, C++, Fortran).

High-level languages, especially MATLAB, are used widely in numerical courses and have several advantages. Namely, the syntax is very similar to the mathematical formulas themselves, the learning curve is short, and debugging is relatively simple. The main drawback is that such languages do not provide the necessary performance to solve large research or industrial problems. This may be a handicap for students if they never gain experience with compiled languages.

Python strikes a middle ground between these options. It is a high-level language with intuitive syntax and high-level libraries for everything needed in a course on numerical methods. At the same time, it is a general-purpose language. Although (like MATLAB) it can be relatively slow [VdP14], Python makes it relatively easy to develop fast code by using tools such as [Cython](#) or [f2py](#). For the kinds of exercises used in most courses, pure Python code is sufficiently fast. In recent years, with the advent of tools like [numpy](#) and [matplotlib](#), Python has increasingly been adopted as a language of instruction for numerical courses.

[IPython](#) [Per07] is a tool for using Python interactively. One of its most useful components is the [IPython notebook](#): a document format containing text, code, images, and more, that can be written, viewed, and executed in a web browser.

The IPython notebook as a textbook medium

Many print and electronic textbooks for numerical methods include code, either printed on the page or available online (or both). Some of my favorite examples are [Tre00] and [LeV07]. Such books have become more common, as the importance of exposing students to the craft of numerical methods -- and the value of experimentation in learning the theory -- has become more recognized. The IPython notebook can be viewed as the next step in this evolution. As demonstrated in Figure 1, it combines in a single document

- Mathematics (using LaTeX)
- Text (using Markdown)
- Code (in Python or other languages)
- Figures and animations

Mathematica, Maple, and (more recently) [Sage](#) have document formats with similar capabilities. The Sage worksheet is very similar to the IPython notebook (indeed, the two projects have strongly influenced each other), so most of what I will say about the IPython notebook applies also to the Sage worksheet.

The notebook has some important advantages over Mathematica and Maple documents:

- It can be viewed, edited, and executed using only **free** software;

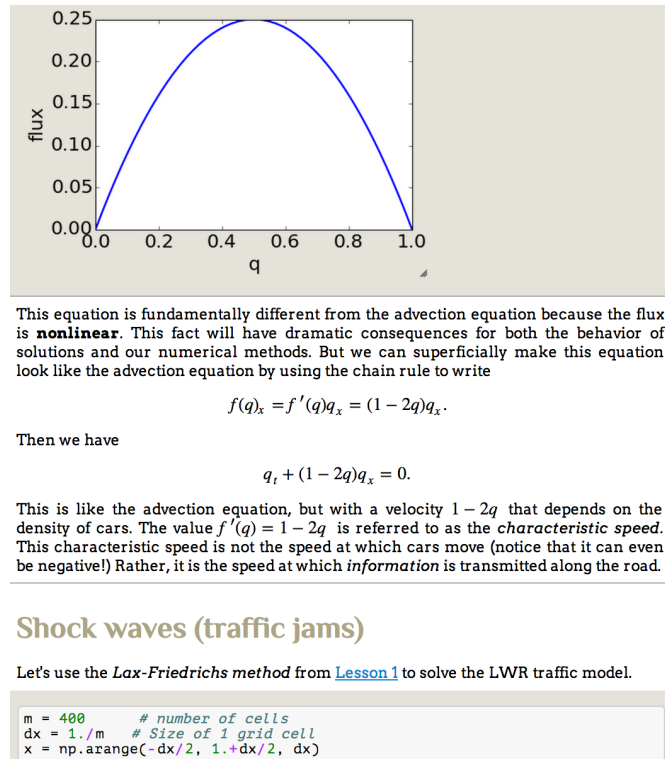


Fig. 1: An excerpt from *Notebook 2 of HyperPython*, showing the use of text, mathematics, code, and a code-generated plot in the IPython notebook.

- It allows the use of multiple programming languages;
- It can be collaboratively edited by multiple users at the same time (currently only on SageMathCloud);
- It is open source, so users can modify and extend it.

The second point above was especially important when I decided to switch from using Sage worksheets to IPython notebooks. Because both are stored as text, I was able to write [a simple script to convert them](#). If I had been using a proprietary binary format, I would have lost a lot of time re-writing my materials in a new format.

Perhaps the most important advantage of the notebook is the community in which it has developed -- a community in which openness and collaboration are the norm. Because of this, those who develop teaching and research materials with IPython notebooks often make them freely available under permissive licenses; see for example Lorena Barba's AeroPython course [Bar14] or [this huge list of books, tutorials, and lessons](#). Due to this culture, the volume and quality of available materials for teaching with the notebook is quickly surpassing what is available in proprietary formats. It should be mentioned that the notebook is also being used as a medium for publishing research, both as open notebook science and full articles.

Mechanics of an interactive, notebook-based course

I have successfully used IPython notebooks as a medium of instruction in both

- semester-length university courses; and
- short 1-3 day tutorials

I will focus on the mechanics of teaching a university course, but much of what I will say applies also to short tutorials. The

notebook is especially advantageous in the context of a tutorial because one does not usually have the luxury of ensuring that students have a textbook. The notebooks for the course can comprise a complete, self-contained curriculum.

Typically I have used a partially-flipped approach, in which half of the class sessions are traditional lectures and the other half are *lab sessions* in which the students spend most of the time programming and discussing their programs. Others have used IPython notebooks with a fully-flipped approach; see for example [Bar13].

Getting students started with the notebook

One historical disadvantage of using Python for a course was the difficulty of ensuring that all students had properly installed the required packages. Indeed, when I began teaching with Python 5 years ago, this was still a major hassle even for a course with twenty students. If just a few percent of the students have installation problems, it can create an overwhelming amount of work for the instructor.

This situation has improved dramatically and is no longer a significant issue. I have successfully used two strategies: local installation and cloud platforms.

Local installation

It can be useful for students to have a local installation of all the software on their own computer or a laboratory machine. The simplest way to achieve this is to install either [Anaconda](#) or [Canopy](#). Both are free and include Python, IPython, and all of the other Python packages likely to be used in any scientific course. Both can easily be installed on Linux, Mac, and Windows systems.

Cloud platforms

In order to avoid potential installation issues altogether, or as a secondary option, notebooks can be run using only cloud services. Two free services exist for running IPython notebooks:

- [Sage Math Cloud](#)
- [Wakari](#)

Both services are relatively new and are developing rapidly. Both include all relevant Python packages by default. I have used both of them successfully, though I have more experience with Sage Math Cloud (SMC). Each SMC project is a complete sandboxed Unix environment, so it is possible for the user to install additional software if necessary. On SMC, it is even possible for multiple users to collaboratively edit notebooks at the same time.

Teaching Python

Since students of numerical methods do not usually have much prior programming experience, and what they have is usually in another language, it is important to give students a solid foundation in Python at the beginning of the course. In the graduate courses I teach, I find that most students have previously programmed in MATLAB and are easily able to adapt to the similar syntax of Numpy. However, some aspects of Python syntax are much less intuitive. Fortunately, a number of excellent Python tutorials geared toward scientific users are available. I find that a 1-2 hour laboratory session at the beginning of the course is sufficient to acquaint students with the necessary basics; further details can be introduced as needed later in the course. Students should be strongly encouraged to work together in developing their programming skills. For examples of such an introduction, see [this notebook](#) or [this one](#).

Lab sessions

At the beginning of each lab session, the students open a new notebook that contains some explanations and exercises. Generally they have already been introduced to the algorithm in question, and the notebook simply provides a short review. Early in the course, most of the code is provided to the students already; the exercises consist mainly of extending or modifying the provided code. As the course progresses and students develop their programming skills, they are eventually asked to implement some algorithms or subroutines from scratch (or by starting from codes they have written previously). Furthermore, the specificity of the instructions is gradually decreased as students develop the ability to fill in the intermediate steps.

It is essential that students arrive to the lab session already prepared, through completing assigned readings or recordings. This doesn't mean that they already know everything contained in the notebook for that day's session; on the contrary, class time should be an opportunity for guided discovery. I have found it very useful to administer a quiz at the beginning of class to provide extra motivation. Quizzes can also be administered just before students begin a programming exercise, in order to check that they have a good plan for completing it, or just after, to see how successful they were.

The main advantage of having students program in class (rather than at home on their own) is that they can talk to the instructor and to other students as they go. Most students are extremely reluctant to do this at first, and it is helpful to require them to explain to one another what their code does (or is intended

to do). This can be accomplished by having them program in pairs (alternating, with one programming while the other makes comments and suggestions). Another option is to have them compare and discuss their code after completing an exercise.

When assisting students during the lab sessions, it is important not to give too much help. When the code fails, don't immediately explain what is wrong or how to fix it. Ask questions. Help them learn to effectively read a traceback and diagnose their code. Let them struggle a bit to figure out why the solution blows up. Even if they seem to grasp things immediately, it's worthwhile to discuss their code and help them develop good programming style.

Typically, in an 80-minute class session the students spend 50-60 minutes working (thinking and programming) and 20-30 minutes listening to explanations, proposing ideas, discussing their solutions, and taking quizzes. During the working time, the instructor should assess and help students one-on-one as needed.

Designing effective notebooks

Prescribing how to structure the notebooks themselves is like stipulating the style of a textbook or lecture notes. Each instructor will have his or her own preferences. So I will share some principles I have found to be effective.

Make sure that they type code from the start

This goes without saying, but it's especially important early in the course. It's possible to write notebooks where all the code involved is already completely provided. That's fine if students only need to understand the output of the code, but not if they need to understand the code itself (which they generally do). The plain truth is that nobody reads code provided to them unless they have to, and when they do they understand only a fraction of it. Typing code, like writing equations, dramatically increases the degree to which we internalize it. At the very beginning of the course, it may be helpful to have students work in an IPython session and type code from a notebook into the IPython prompt.

Help students to discover concepts on their own

This is the central principle of inquiry-based learning. Students are more motivated, gain more understanding, and retain knowledge better when they discover things through their own effort and after mentally engaging on a deep level. In a numerical methods course, the traditional approach is to lecture about instability or inaccuracy, perhaps showing an example of a method that behaves poorly. In the flipped approach, you can instead allow the students to implement and experiment in class with naive algorithms that seem reasonable but may be inaccurate or unstable. Have them discuss what they observe and what might be responsible for it. Ask them how they think the method might be improved.

Teaching is tricky because you want the students to come up to date on topics which have taken perhaps decades to develop. But they gain the knowledge quickly without the discipline of having struggled with issues. By letting them struggle and discover you simulate the same circumstances which produced the knowledge in the first place.

Tailor the difficulty to the students' level

Students will lose interest or become frustrated if they are not challenged or they find the first exercise insurmountable. It can be difficult to accommodate the varying levels of experience and skill presented by students in a course. For students who struggle

with programming, peer interaction in class is extremely helpful. For students who advance quickly, the instructor can provide additional, optional, more challenging questions. For instance, in my [HyperPython short course](#), some notebooks contain challenging "extra credit" questions that only the more advanced students attempt.

Gradually build up complexity

In mathematics, one learns to reason about highly abstract objects by building up intuition with one layer of abstraction at a time. Numerical algorithms should be developed and understood in the same way, with the building blocks first coded and then encapsulated as subroutines for later use. Let's consider the multigrid algorithm as an example. Multigrid is a method for solving systems of linear equations that arise in modeling things like the distribution of heat in a solid. The basic building block of multigrid is some way of smoothing the solution; the key idea is to apply that smoother successively on computational grids with different levels of resolution.

I have students code things in the following sequence:

- 1) Jacobi's method (a smoother that doesn't quite work)
- 2) Under-relaxed Jacobi (a smoother that does work for high frequencies)
- 3) A two-grid method (applying the smoother on two different grids in succession)
- 4) The V-cycle (applying the smoother on a sequence of grid)
- 5) Full multigrid (performing a sequence of V-cycles with successively finer grids)

In each step, the code from the previous step becomes a subroutine. In addition to being an aid to learning, this approach teaches students how to design programs well. The multigrid notebook from my course can be found (with some exercises completed) [here](#).

Use animations liberally

Solutions of time-dependent problems are naturally depicted as animations. Printed texts must restrict themselves to waterfall plots or snapshots, but electronic media can show solutions in the natural way. Students learn more -- and have more fun -- when they can visualize the results of their work in this way. I have used Jake Vanderplas' JSAnimation package [VdP13] to easily create such animations. The latest release of IPython (version 2.1.0) natively includes interactive widgets that can be used to animate simulation results.

Time-dependent solutions are not the only things you can animate. For iterative solvers, how does the solution change after each algorithmic iteration? What effect does a given parameter have on the results? Such questions can be answered most effectively through the use of animation. One simple example of teaching a concept with such an animation, shown in Figure 2, can be found in [this notebook on aliasing](#).

Drawbacks

The approach proposed here differs dramatically from a traditional course in numerical methods. I have tried to highlight the advantages of this approach, but of course there are also some potential disadvantages.

Material covered

The most substantial drawback I have found relates to the course coverage. Programming even simple algorithms takes a lot of time, especially for students. Therefore, the amount of material that can be covered in a semester-length course on numerical methods is substantially less under the interactive or flipped model. This is true for inquiry-based learning techniques in general, but even more so for courses that involve programming. I believe that it is better to show less material and have it fully absorbed and loved than to quickly dispense knowledge that falls on deaf ears.

Scalability

While some people do advocate IBL even for larger classes, I have found that this approach works best if there are no more than twenty students in the course. With more students, it can be difficult to fit everyone in a computer lab and nearly impossible for the instructor to have meaningful interaction with individual students.

Nonlinear notebook execution

Code cells in the notebook can be executed (and re-executed) in any order, any number of times. This can lead to different results than just executing all the cells in order, which can be confusing to students. I haven't found this to be a major problem, but students should be aware of it.

Opening notebooks

Perhaps the biggest inconvenience of the notebook is that opening one is not as simple as clicking on the file. Instead, one must open a terminal, go to the appropriate directory, and launch the ipython notebook. This is fine for users who are used to UNIX, but is non-intuitive for some students. With IPython 2.0, one can also launch the notebook from any higher-level directory and then navigate to a notebook file within the browser.

It's worth noting that on SMC one can simply click on a notebook file to open it.

Lengthy programs and code reuse

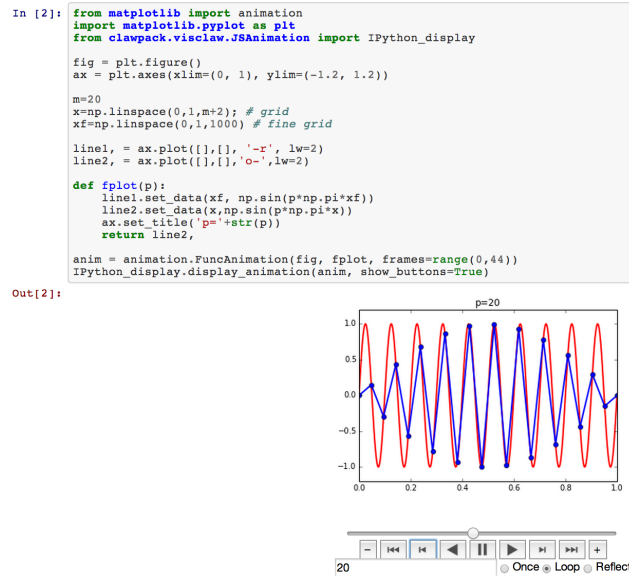
Programming in the browser means you don't have all the niceties of your favorite text editor. This is no big deal for small bits of code, but can impede development for larger programs. I also worry that using the notebook too much may keep students from learning to use a good text editor. Finally, running long programs from the browser is problematic since you can't detach the process.

Usually, Python programs for a numerical methods course can be broken up into fairly short functions that each fit on a single screen and run in a reasonable amount of time.

Placing code in notebooks also makes it harder to reuse code, since functions from one notebook cannot be used in another with copying. Furthermore, for the reasons already given, the notebook is poorly suited to development of library code. Exclusive use of the notebook for coding may thus encourage poor software development practices. This can be partially countered by teaching students to place reusable functions in files and then importing them in the notebook.

Interactive plotting

In my teaching notebooks, I use Python's most popular plotting package, Matplotlib [Hun07]. It's an extremely useful package, whose interface is immediately familiar to MATLAB users, but



Try to answer the questions below with pencil and paper; then check them by modifying the code above.

1. For a given number of grid points m , which modes p will be aliased to the $p = 0$ mode?
2. What is the highest frequency mode that can be represented on a given grid?

Fig. 2: A short notebook on grid aliasing, including code, animation, and exercises.

it has a major drawback when used in the IPython notebook. Specifically, plots that appear inline in the notebook are not interactive -- for instance, they cannot be zoomed or panned. There are a number of efforts to bring interactive plots to the notebook (such as Bokeh and Plotly) and I expect this weakness will soon be an area of strength for the IPython ecosystem. I plan to incorporate one of these tools for plotting in the next course that I teach.

More resources

Many people are advocating and using the IPython notebook as a teaching tool, for many subjects. For instance, see:

- [Teaching with the IPython Notebook](#) by Matt Davis
- [How IPython Notebook and Github have changed the way I teach Python](#) by Eric Matthes
- [Using the IPython Notebook as a Teaching Tool](#) by Greg Wilson
- [Teaching with ipython notebooks -- a progress report](#) by C. Titus Brown

To find course actual course materials (in many subjects!), the best place to start is this curated list: [A gallery of interesting IPython Notebooks](#).

Acknowledgments

I am grateful to Lorena Barba for helpful discussions (both online and offline) of some of the ideas presented here. I thank Nathaniel Collier, David Folch, and Pieter Holtzhausen for their comments that significantly improved this paper. This work was supported by the King Abdullah University of Science and Technology (KAUST).

REFERENCES

- [LeV07] R. J. LeVeque. *Finite Difference Methods for Ordinary and Partial Differential Equations*, Society for Industrial and Applied Mathematics, 2007.
- [Tre00] L. N. Trefethen. *Spectral Methods in MATLAB*, Society for Industrial and Applied Mathematics, 2000.
- [Bar14] L. A. Barba, O. Mesnard. *AeroPython*, 10.6084/m9.figshare.1004727. Code repository, Set of 11 lessons in classical Aerodynamics on IPython Notebooks. April 2014.
- [Bar13] L. A. Barba. *CFD Python: 12 steps to Navier-Stokes*, <http://lorenabarba.com/blog/cfd-python-12-steps-to-navier-stokes/>, 2013.
- [Hal75] P. R. Halmos, E. E. Moise, and G. Piranian. *The problem of learning how to teach*, The American Mathematical Monthly, 82(5):466--476, 1975.
- [Ern14a] D. Ernst. *What the heck is IBL?*, Math Ed Matters blog, <http://maamathedmatters.blogspot.com/2013/05/what-heck-is-ibl.html>, May 2014.
- [Ern14b] D. Ernst. *What's So Good about IBL Anyway?*, Math Ed Matters blog, <http://maamathedmatters.blogspot.com/2014/01/whats-so-good-about-ibl-anyway.html>, May 2014.
- [VdP14] J. VanderPlas. *Why Python is Slow: Looking Under the Hood*, Pythonic Perambulations blog, <http://jakevdp.github.io/blog/2014/05/09/why-python-is-slow/>, May 2014.
- [VdP13] J. VanderPlas. *JSAnimation*, <https://github.com/jakevdp/JSAnimation>, 2013.
- [Per07] F. Pérez, B. E. Granger. *IPython: A System for Interactive Scientific Computing*, Computing in Science and Engineering, 9(3):21-29, 2007. <http://ipython.org/>
- [Hun07] J. D. Hunter. *Matplotlib: A 2D graphics environment*, Computing in Science and Engineering, 9(3):90-95, 2007. <http://matplotlib.org/>

Project-based introduction to scientific computing for physics majors

Jennifer Klay^{‡*}

<https://www.youtube.com/watch?v=eJhmMf6bHDU>

Abstract—This paper presents an overview of a project-based course in computing for physics majors using Python and the IPython Notebook that was developed at Cal Poly San Luis Obispo. The course materials are made freely available on GitHub as a project under the Computing4Physics [C4P] organization.

Index Terms—physics, scientific computing, undergraduate education

Introduction

Computational tools and skills are as critical to the training of physics majors as calculus and math, yet they receive much less emphasis in the undergraduate curriculum. One-off courses that introduce programming and basic numerical problem-solving techniques with commercial software packages for topics that appear in the traditional physics curriculum are insufficient to prepare students for the computing demands of modern technical careers. Yet tight budgets and rigid degree requirements constrain the ability to expand computational course offerings for physics majors.

This paper presents an overview of a recently revamped course at California Polytechnic State University San Luis Obispo (Cal Poly) that uses Python and associated scientific computing libraries to introduce the fundamentals of open-source tools, version control systems, programming, numerical problem solving and algorithmic thinking to undergraduate physics majors. The spirit of the course is similar to the bootcamps organized by Software Carpentry [SWC] for researchers in science but is offered as a ten-week for-credit course. In addition to having a traditional in-class component, students learn the basics of Python by completing tutorials on Codecademy's Python track [Codecademy] and practice their algorithmic thinking by tackling Project Euler problems [PE]. This approach of incorporating online training may provide a different way of thinking about the role of MOOCs in higher education. The early part of the course focuses on skill-building, while the second half is devoted to application of these skills to an independent research-level computational physics project. Examples of recent student projects and their results will be presented.

* Corresponding author: jklay@calpoly.edu

‡ California Polytechnic State University San Luis Obispo

Copyright © 2014 Jennifer Klay. This is an open-access article distributed under the terms of the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original author and source are credited.

Background

California Polytechnic State University San Luis Obispo (Cal Poly) is one of the 23 campuses of the California State University system. The university has a "learn by doing" emphasis for the educational experience of its predominantly undergraduate population of approximately 19,000 students, encapsulated in its motto *discere faciendo*. Part of the university's mission is to provide students the opportunity to get directly involved in research at the frontiers of knowledge through interaction with faculty. The university is also committed to enhancing opportunities for under-represented groups and is committed to fostering a diverse student body.

The College of Engineering enrolls the largest fraction of Cal Poly undergraduates (~28%). Due to the large number of engineering undergraduates at Cal Poly, the distribution of male (~54%) and female (~46%) students is opposite that of the national average.

The Department of Physics, in the College of Science & Mathematics, offers Bachelor of Science and Arts degrees in Physics, and minors in astronomy and geology, with approximately 150 students enrolled. There are roughly 30 tenure-track faculty, for a current student-to-faculty ratio of 1:5. In addition, there are typically 5-10 full-time lecturers and fifteen part-time and retired faculty teaching courses in physics and geology. A typical introductory physics course for scientists and engineers has 48 students, in contrast to typical class sizes of over a hundred at large public universities. The curriculum for physics majors includes a Senior Project which is often the continuation of paid summer internships undertaken with faculty members in the department who have funding to support student assistants. Some internal funding is made available to support these activities.

Cal Poly has one of the largest (in terms of degrees granted) and most successful undergraduate physics programs in the United States. Only about 5% of all physics programs in the United States regularly award more than 15 degrees per year, and most of those are at Ph.D. granting institutions. In 2013-2014, 28 B.S. and 1 B.A. degrees were awarded. The Cal Poly Physics Department is uniquely successful among four-year colleges. As a result, Cal Poly was one of 21 departments deemed to be "thriving" and profiled in 2002 by the SPIN-UP study (Strategic Programs for Innovation in Undergraduate Physics) sponsored by the American Association of Physics Teachers, the American Physical Society, and the American Institute of Physics [SPIN-UP]. The external reviewers from SPIN-UP made special mention of the strong

faculty-student interactions and of the success of the physics lounge (known as "h-bar") at making students feel welcome and at home in an intense academic environment. Cal Poly hosted the SPIN-UP Western Regional Workshop in June 2010 where faculty teams from 15 western colleges and universities came to learn how to strengthen their undergraduate physics programs.

Computational physics at Cal Poly

The physics department has a strong record of preparing students for advanced degrees in physics, often at top tier research institutions. Between 2005 and 2009, at least 20% of Cal Poly physics graduates entered Ph.D. programs in physics and related disciplines with another 10% seeking advanced degrees in engineering, mathematics, law, and business.

The Cal Poly physics program provides a strong base in theoretical physics with the standard traditional sequence of courses while providing excellent experimental training of students in the laboratory, with a full year of upper division modern physics experiments and several additional specialty lab courses offered as advanced physics electives. Unfortunately, the department has not yet developed as cohesive and comprehensive of a program in computational physics. There has been one course "Physics on the Computer" on computational methods required for physics majors since 1996. The current catalog description of the course is

Introduction to using computers for solving problems in physics: differential equations, matrix manipulations, simulations and numerical techniques, nonlinear dynamics. 4 lectures.

Students are encouraged to take the course in the Spring of their sophomore year, after completing their introductory physics and math courses. The original pre-requisites for the course were General Physics III: Electricity and Magnetism and Linear Analysis I (MATH), although in 1998 concurrent enrollment for Linear Analysis was allowed and in 2001 the phrase "and computer literacy" was added to the pre-requisites, although it was dropped when enforceable pre-requisites were introduced in 2011.

Despite the desire for students to come to this course with some "computer literacy", no traditional computer science courses have been required for physics majors (although they can be counted as free technical electives in the degree requirements). Each instructor selects the tools and methods used to implement the course. Early on, many numerical topics were covered using Excel because students typically had access and experience with it. Interactive computer algebra systems such as Maple and interactive computing environments such as MATLAB were also employed, but no open-source standard high level programming languages were used. Between 2007 and 2012 MATLAB was the preferred framework, although some use of Excel for introductory tasks was also included.

Although simple data analysis and graphing tasks are taught in upper division laboratories, there is no concerted effort to include computational or numerical techniques in upper division theory courses. Instructors choose to include this material at their own discretion. There is also currently no upper division computational physics elective in the catalog.

When I joined the faculty of Cal Poly in 2007 I quickly obtained external funding from the National Science Foundation to involve Cal Poly physics undergraduates in research at the CERN Large Hadron Collider with the ALICE experiment. My background in particle and nuclear physics has been very software

intensive, owing to the enormous and complex datasets generated in heavy nucleus collisions. I have served as software coordinator for one of the ALICE detector sub-systems and I am the architect and lead developer of the offline analysis framework for the Neutron Induced Fission Fragment Tracking Experiment (NIFFTE). Most of my scientific software is written in C/C++, although I have experience with Pascal, Fortran, Java and shell scripting. I found it extremely challenging to engage students in my research because of the steep learning curve for these software tools and languages.

In 2012 I became interested in learning Python and decided to offer an independent study course called "Python 4 Physicists" so students could learn it with me. Over 30 eager students signed up for the course. We followed Allen Downey's "Think Python" book [Downey2002] for six weeks, largely on our own, but met weekly for one hour to discuss issues and techniques. For the second half of the course, the students were placed in groups of 3 and assigned one of two projects, either a cellular automaton model of traffic flow or a 3-D particle tracking algorithm for particle collision data reconstruction. All code and projects were version controlled with git and uploaded to GitHub. Examples can be found on GitHub [Traffic], [3DTracker]. At the end of the quarter the groups presented their projects to the class.

Not all groups were able to successfully complete the projects but this is likely due to competing priorities consuming their available coding time given that this was only a 1-unit elective course. Nevertheless, they were excited to work on a research-level problem and to be able to use their newly acquired programming skills to do so. Most of them gained basic programming proficiency and some students reported that the course helped them secure summer internships. It became clear to me that Python is an effective and accessible language for teaching physics majors how to program. When my opportunity to teach "Physics on the Computer" came in 2013-14, I decided to make it a project-based Python programming course that would teach best practices for scientific software development, including version control and creation of publication quality graphics, while giving a broad survey of major topics in computational physics.

Course Organization

The complete set of materials used for this course are available on GitHub under the Computing4Physics [C4P] organization and can be viewed with the IPython Notebook Viewer [nbviewer]. The learning objectives for the course are a subset of those developed and adopted by the Cal Poly physics department in 2013 for students completing a degree in physics:

- Use basic coding concepts such as loops, control statements, variable types, arrays, array operations, and boolean logic. (LO1)
- Write, run and debug programs in a high level language. (LO2)
- Carry out basic operations (e.g. cd, ls, dir, mkdir, ssh) at the command line. (LO3)
- Maintain a version controlled repository of your files and programs. (LO4)
- Create publication/presentation quality graphics, equations. (LO5)
- Visualize symbolic analytic expressions - plot functions and evaluate their behavior for varying parameters. (LO6)

- Use numerical algorithms (e.g. ODE solvers, FFT, Monte Carlo) and be able to identify their limitations. (LO7)
- Code numerical algorithms from scratch and compare with existing implementations. (LO8)
- Read from and write to local or remote files. (LO9)
- Analyze data using curve fitting and optimization. (LO10)
- Create appropriate visualizations of data, e.g. multidimensional plots, animations, etc. (LO11)

The course schedule and learning objective map are summarized in Table 1. Class time was divided into two 2-hour meetings on Tuesdays and Thursdays each week for ten weeks. For the first two weeks the students followed the Python track at Codecademy [Codecademy] to learn basic syntax and coding concepts such as loops, control statements, variable types, arrays, array operations, and boolean logic. In class, they were instructed about the command line, ssh, the UNIX shell and version control. Much of the material for the early topics came from existing examples, such as Software Carpentry [SWC] and Jake Vanderplas's Astronomy 599 course online [Vanderplas599]. These topics were demonstrated and discussed as instructor-led activities in which they entered commands in their own terminals while following along with me.

The IPython Notebook was introduced in the second week and their first programming exercise outside of Codecademy was to pair-program a solution to Project Euler [PE] Problem 1. They created their own GitHub repository for the course and were guided through the workflow at the start and end of class for the first several weeks to help them get acclimated. We built on their foundations by taking the Battleship game program they wrote in Codecademy and combining it with ipythonblocks [ipythonblocks] to make it more visual. We revisited the Battleship code again in week 4 when we learned about error handling and a subset of the students used ipythonblocks as the basis for their final project on the Schelling Model of segregation. The introduction, reinforcement and advanced application of programming techniques was employed to help students build lasting competency with fundamental coding concepts.

For each class session, the students were provided a "tour" of a specific topic for which they were instructed to read and code along in their own IPython Notebook. They were advised not to copy/paste code, but to type their own code cells, thinking about the commands as they went to develop a better understanding of the material. After finishing a tour they worked on accompanying exercises. I was available in class for consultations and questions but there was very little lecturing beyond the first week. Class time was activity-based rather than lecture-based. Along with the homework exercises, they completed a Project Euler problem each week to practice efficient basic programming and problem solving.

A single midterm exam was administered in the fifth week to motivate the students to stay on top of their skill-building and to assess their learning at the midway point. The questions on the midterm were designed to be straightforward and completable within the two-hour class time.

Assessment of learning

Figuring out how to efficiently grade students' assignments is a non-trivial task. Grading can be made more efficient by automatic output checking but that doesn't leave room for quality assessment and feedback. To deal with the logistics of grading, a set of UNIX shell scripts was created to automate the bookkeeping and communication of grades. Individual assignments were assessed

Week	Topics	Learning Objectives
1	Programming Bootcamp	LO1, LO2, LO3, LO4
2	Programming Bootcamp	LO1-4, LO11
3	Intro to NumPy/SciPy, Data I/O	LO1-4, LO9, LO11
4	Graphics, Animation and Error handling	LO1-4, LO5, LO6, LO11
5	Midterm Exam, Projects and Program Design	LO1-4, LO5, LO6, LO9
6	Interpolation and Differentiation	LO1-4, LO5, LO6, LO7, LO8, LO11
7	Numerical Integration, Ordinary Differential Equations (ODEs)	LO1-4, LO5, LO6, LO7, LO8, LO11
8	Random Numbers and Monte-Carlo Methods	LO1-4, LO5, LO6, LO7, LO8, LO11
9	Linear Regression and Optimization	LO1-11
10	Symbolic Analysis, Project Hack-a-thon!	LO1-4, LO5, LO6, LO11
Final	Project Demos	LO1-11

TABLE 1: Course schedule of topics and learning objectives

Points	Description
5	Goes above and beyond. Extra neat, concise, well-commented code, and explores concepts in depth.
4	Complete and correct. Includes an analysis of the problem, the program, verification of at least one test case, and answers to questions, including plots.
3	Contains a few minor errors.
2	Only partially complete or has major errors.
1	Far from complete.
0	No attempt.

TABLE 2: Grading rubric for assigned exercises.

personally by me while a grader was employed to evaluate the Project Euler questions. The basic grading rubric uses a 5-point scale for each assigned question, outlined in Table 2. Comments and numerical scores were recorded for each student and communicated to them through a script-generated email. Students' final grades in the course were determined by weighting the various course elements accordingly: Project Euler (10%), Exercises (30%), Midterm (20%), Project (30%), Demo (10%).

Projects

Following the midterm exam one class period was set aside for presenting three project possibilities and assigning them. Two of the projects came from Stanford's NIFTY assignment database [Nifty] - "Schelling's Model of Segregation" by Frank McCown [McCown2014] and "Estimating Avogadro's Number from Brownian Motion" by Kevin Wayne [Wayne2013]. The Schelling Model project required students to use IPython widgets and ipythonblocks to create a grid of colored blocks that move according to a set of rules governing their interactions. Several recent physics publications on the statistical properties

of Schelling Model simulations and their application to physical systems [Vinkovic2006], [Gauvin2009], [DallAsta2008] were used to define research questions for the students to answer using their programs. For estimating Avogadro's number, the students coded a particle identification and tracking algorithm that they could apply to the frames of a movie showing Brownian motion of particles suspended in fluid. The initial test data came from the Nifty archive, but at the end of the quarter the students collected their own data using a microscope in the biology department to image milkfat globules suspended in water. The challenges of adapting their code to the peculiarities of a different dataset were part of the learning experience. They used code from a tour and exercise they did early in the quarter, based on the MultiMedia programming lesson on Software Carpentry, which had them filter and count stars in a Hubble image.

The third project was to simulate galaxy mergers by solving the restricted N-body problem. The project description was developed for this course and was based on a 1972 paper by Toomre and Toomre [Toomre1972]. They used SciPy's *odeint* to solve the differential equations describing the motion of a set of massless point particles (stars) orbiting a main galaxy core as a disrupting galaxy core passed in a parabolic trajectory. The students were not instructed on solving differential equations until week 7, so they were advised to begin setting up the initial conditions and visualization code until they had the knowledge and experience to apply *odeint*.

The projects I selected for the course are ones that I have not personally coded myself but for which I could easily outline a clear algorithmic path to a complete solution. Each one could form a basis for answering real research questions. There are several reasons for this approach. First, I find it much more interesting to learn something new through the students' work. I would likely be bored otherwise. Second, having the students work on a novel project is similar to how I work with students in research mentoring. My interactions with them are much more like a real research environment. By not already having one specific solution I am able to let them choose their own methods and algorithms, providing guidance and suggestions rather than answers to every problem or roadblock they encounter. This gives them the chance to experience the culture of research before they engage in it outside of the classroom. Finally, these projects could easily be extended into senior projects or research internship opportunities, giving the students the motivation to keep working on their projects after the course is over. As a consequence of these choices, the project assessment was built less on "correctness" than on their formulation of the solution, documentation of the results, and their attempt to answer the assigned "research question". The rubric was set up so that they could earn most of the credit for developing an organized, complete project with documentation, even if their results turned out to be incorrect.

When this course was piloted in 2013, project demonstrations were not included, as they had been for the 2012 independent study course. I was disappointed in the effort showed by the majority of students in the 2013 class, many of whom ultimately gave up on the projects and turned in sub-standard work, even though they were given additional time to complete them. For 2014, the scheduled final exam time was used for 5-7 minute project demonstrations by each individual student. Since the class was divided into three groups, each working on a common project, individual students were assigned a personalized research question to answer with their project code and present during their demo.

The students were advised that they needed to present *something*, even if their code didn't function as expected. Only one student out of 42 did not make a presentation. (That student ultimately failed the course for turning in less than 50% of assignments and not completing the project.) The rest were impressive, even when unpolished.

It was clear from the demos that the students were highly invested in their work and were motivated to make a good impression. The project demos were assessed using a peer evaluation oral presentation rubric that scored the demos on organization, media (graphics, animations, etc. appropriate for the project), delivery, and content. Presenters were also asked to evaluate their own presentations. Grades were assigned using the average score from all peer evaluation sheets. The success of the project demos strongly suggest that they are an essential part of the learning experience for students. This is supported in the literature. See for example, Joughin and Collom [Joughin2003].

Project Examples

The most impressive example from 2014 came from a student who coded the Galaxy Merger project [Parry2014]. Figure 1 shows a still shot from an animated video he created of the direct passage of an equal mass disruptor after the interaction has begun. He also uploaded Youtube videos of his assigned research question (direct passage of an equal mass disruptor) from two perspectives, the second of which he coded to follow his own curiosity - it was not part of the assignment. The main galaxy perspective can be viewed here: <http://www.youtube.com/watch?v=vavfpLwmT0o> and the interaction from the perspective of the disrupting galaxy can be viewed here: <http://www.youtube.com/watch?v=iy7WvV5LUZg>

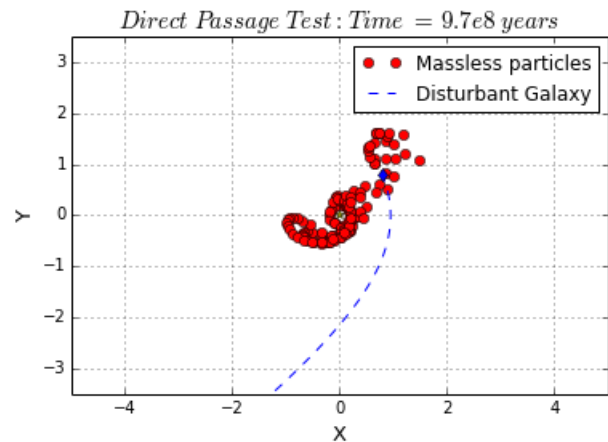


Fig. 1: Direct passage of an equal mass disruptor galaxy shortly after the disrupting galaxy passes the minimum distance of approach. [Parry2014]

There were also two other good Youtube video examples of the galaxy merger project, although the solutions exhibited pathologies that this one did not.

The best examples from the Schelling Model either did an excellent analysis of their research question [Nelson2014] or created the most complete and useful interactive model [Parker2014].

Highlights from 2013

Although no project demos were required in 2013, students who submitted excellent projects were invited to collaborate together

on a group presentation of their work at the 2013 annual meeting of the Far West Section of the American Physical Society held at Sonoma State University Nov. 1-2, 2013 [Sonoma2013]. Two talks were collaborations among four students each, one talk was a pair collaboration, and one was given as a single author talk.

The single author talk came from the best project submitted in 2013, an implementation of a 3-D particle tracking code [VanAtta2013] for use with ionization chamber data from particle collision experiments. Figure 2 shows an example of the output from his tracker with the voxels associated with different trajectories color coded. The notebook was complete and thorough, addressing all the questions and including references. Although the code could be better organized to improve readability, the results were impressive and the algorithm was subsequently adapted into the NIFFTE reconstruction framework for use in real experiments.

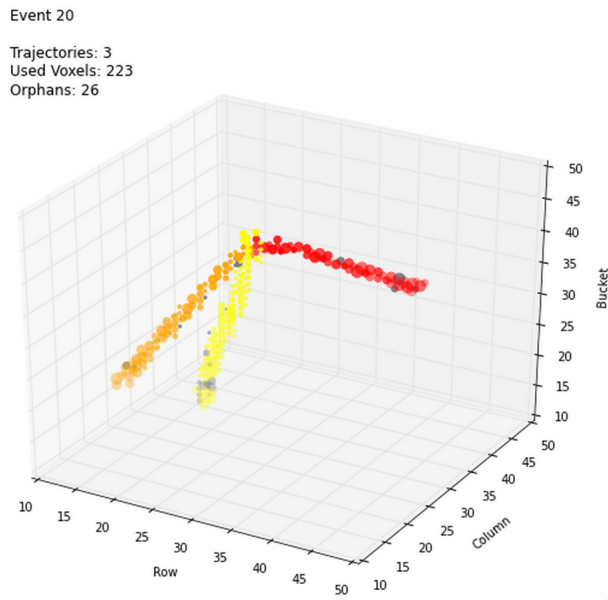


Fig. 2: Matplotlib 3d plot of particle trajectories reconstructed from ionization trails left by charged particles in a gaseous drift detector. [VanAtta2013]

One of the students from the pair collaboration turned his project from 2013 into a Cal Poly senior project recently submitted [Rexrode2014]. He extended his initial work and created an open library of code for modeling the geometry of nuclear collisions with the Monte Carlo Glauber model. The project writeup and the code can be found on GitHub under the [MCGlauber] organization.

Pre- and Post- Assessment

In order to assess the course's success at achieving the learning objectives, both a pre-learner survey and course evaluations were administered anonymously. The pre-learner survey, adapted from a similar Software Carpentry example, was given on the first day of class with 100% participation, while the course evaluation was given in the last week. Some in class time was made available for the evaluations but students were also able to complete it on their own time. Course evaluations are conducted through the Cal Poly "SAIL" (Student Assessment of Instruction and Learning) online system. SAIL participation was 82%. Some questions were common to both the pre and post assessment, for comparison.

Learning Objective	Completely or mostly	Neutral or partially	Not met
LO1	33/36	3/36	0/36
LO2	31/36	5/36	0/36
LO3	33/36	2/36	0/36
LO4	31/36	5/36	0/36
LO5	32/36	4/36	0/36
LO6	31/35	4/35	0/35
LO7	25/35	10/35	0/35
LO8	27/35	7/35	1/35
LO9	30/35	5/35	0/35
LO10	26/35	9/35	0/35
LO11	30/35	5/35	0/35

TABLE 3: Student evaluation of how well the course met the learning objectives.

Language	Pre-	Post-
Fortran	0/42	1/34
C	5/42	7/34
C++	6/42	5/34
Perl	0/42	0/34
MATLAB	5/42	1/34
Python	3/42	31/34
R	1/42	1/34
Java	7/42	5/34
Others (list)	7/42	1/34
Labview		
None	20/42	2/34

TABLE 4: With which programming languages could you write a program from scratch that reads a column of numbers from a text file and calculates mean and standard deviation of that data? (Check all that apply)

The first question on the post-assessment course evaluation asked the students to rate how well the course met each of the learning objectives. The statistics from this student-based assessment are included in Table 3.

Students were also asked to rate the relevance of the learning objectives for subsequent coursework at Cal Poly and for their career goals beyond college. In both cases, a majority of students rated the course as either "Extremely useful, essential to my success" (21/34 and 20/34) or "Useful but not essential" (12/34 and 11/34) and all but one student out of 34 expected to use what they learned beyond the course itself. Almost all students indicated that they spent at least 5-6 hours per week outside of class doing work for the course, with half (17/34) indicating they spent more than 10 hours per week outside of class.

The four questions that were common to both the pre- and post- evaluations and their corresponding responses are included in Tables 4, 5, 6, and 7.

It is worth noting that the 7/42 students who indicated they could complete the programming task with Labview at the beginning of the course probably came directly from the introductory electronics course for physics majors, which uses Labview heavily.

Of the free response comments in the post-evaluation, the most common was that more lecturing by the instructor would have enhanced their learning and/or helped them to better understand some of the coding concepts. In future offerings, I might add

Answer	Pre-	Post-
I could not complete this task.	19/42	3/34
I could complete the task with documentation or search engine help.	22/42	13/34
I could complete the task with little or no documentation or search engine help.	1/42	18/34

TABLE 5: In the following scenario, please select the answer that best applies to you. A tab-delimited file has two columns showing the date and the highest temperature on that day. Write a program to produce a graph showing the average highest temperature for each month.

Answer	Pre-	Post-
I could not complete this task.	42/42	2/34
I could complete the task with documentation or search engine help.	0/42	17/34
I could complete the task with little or no documentation or search engine help.	0/42	15/34

TABLE 6: In the following scenario, please select the answer that best applies to you. Given the URL for a project's version control repository, check out a working copy of that project, add a file called notes.txt, and commit the change.

a brief mini-lecture to the beginning of each class meeting to introduce and discuss concepts but I will keep the focus on student-centered active learning.

Conclusion

This paper presented an example of a project-based course in scientific computing for undergraduate physics majors using the Python programming language and the IPython Notebook. The complete course materials are available on GitHub through the Computing4Physics [C4P] organization. They are released under a modified MIT license that grants permission to anyone the right to use, copy, modify, merge, publish, distribute, etc. any of the content. The goal of this project is to make computational tools for training physics majors in best practices freely available. Contributions and collaboration are welcome.

The Python programming language and the IPython Notebook are effective open-source tools for teaching basic software skills. Project-based learning gives students a sense of ownership of their work, the chance to communicate their ideas in oral live software demonstrations and a starting point for engaging in physics research.

REFERENCES

- [C4P] All course materials can be obtained directly from the Computing4Physics organization on GitHub at <https://github.com/Computing4Physics/C4P>
- [SWC] "Software Carpentry: Teaching lab skills for scientific computing", <http://software-carpentry.org/>, accessed 2 July 2014.

Answer	Pre-	Post-
I could not create this list.	35/42	3/34
I would create this list using "Find in Files" and "copy and paste"	2/42	0/34
I would create this list using basic command line programs.	4/42	2/34
I would create this list using a pipeline of command line programs.	1/42	2/34
I would create this list using some Python code and the ! escape.	N/A	19/34
I would create this list with code using the Python 'os' and 'sys' libraries.	N/A	8/34

TABLE 7: How would you solve this problem? A directory contains 1000 text files. Create a list of all files that contain the word "Drosophila" and save the result to a file called results.txt. **Note:** the last two options on this question were included in the post-survey only.

- [Codecademy] "Codecademy: Learn to code interactively, for free.", <http://www.codecademy.com/>, accessed 2 July 2014.
- [PE] "ProjectEuler.net: A website dedicated to the puzzling world of mathematics and programming", <https://projecteuler.net/>, accessed 2 July 2014.
- [SPIN-UP] "American Association of Physics Teacher: Strategic Programs for Innovations in Undergraduate Physics", <http://www.aapt.org/Programs/projects/spinup/>, accessed 2 July 2014.
- [Downey2002] Allen B. Downey, Jeffrey Elkner, and Chris Meyers, "Think Python: How to Think Like a Computer Scientist", Green Tea Press, 2002, ISBN 0971677506, <http://www.greenteapress.com/thinkpython/thinkpython.html>
- [Traffic] D.Townsend, J. Fernandes, R. Mullen, and A. Parker, GitHub repository for the cellular automaton model of traffic flow created for the Spring 2012 PHYS 200/400 course at Cal Poly, <https://github.com/townsenddw/discrete-graphic-traffic>, accessed 2 July 2014.
- [3DTracker] R.Cribbs, K. Boucher, R. Campbell, K. Flatland, and B. Norris, GitHub repository for the 3-D pattern recognition tracker created for the Spring 2012 PHYS 200/400 course at Cal Poly, <https://github.com/Rolzroyz/3Dtracker>, accessed 2 July 2014.
- [nbviewer] "nbviewer: A simple way to share IPython Notebooks", <http://nbviewer.ipython.org>, accessed 2 July 2014.
- [Vanderplas599] Jake Vanderplas, "Astronomy 599: Introduction to Scientific Computing in Python", https://github.com/jakevdp/2013_fall_ASTR599/, accessed 2 July 2014.
- [ipythonblocks] "ipythonblocks: code + color", <http://ipythonblocks.org/>, accessed 2 July 2014.
- [Nifty] "Nifty Assignments: The Nifty Assignments session at the annual SIGCSE meeting is all about gathering and distributing great assignment ideas and their materials.", <http://nifty.stanford.edu/>, accessed 2 July 2014.
- [McCown2014] Frank McCown, "Schelling's Model of Segregation", <http://nifty.stanford.edu/2014/mccown-schelling-model-segregation/>, accessed 2 July 2014.
- [Wayne2013] Kevin Wayne, "Estimating Avogadro's Number", <http://nifty.stanford.edu/2013/wayne-avogadro.html>, accessed 2 July 2014.
- [Vinkovic2006] D.Vinkovic and A.Kirman, Proc.Nat.Acad.Sci., vol. 103 no. 51, 19261-19265 (2006). <http://www.pnas.org/content/103/51/19261.full>
- [Gauvin2009] L.Gauvin, J.Vannimenus, J.-P.Nadal, Eur.Phys.J. B, Vol. 70:2 (2009). <http://link.springer.com/article/10.1140/2Fepjb%2Fe2009-00234-0>

- [DallAsta2008] L.Dall'Asta, C.Castellano, M.Marsili, J.Stat.Mech. L07002 (2008). <http://iopscience.iop.org/1742-5468/2008/07/L07002/>
- [Toomre1972] A.Toomre and J.Toomre, Astrophysical Journal, 178:623-666 (1972). <http://adsabs.harvard.edu/abs/1972ApJ...178..623T>
- [Joughin2003] G.Joughin and G.Collom, "Oral Assessment. The Higher Education Academy", (2003) http://www.heacademy.ac.uk/resources/detail/resource_database/id433_oral_assessment, retrieved 2 July 2014.
- [Parry2014] B.W. Parry, "Galaxy Mergers: The Direct Passage Case", <http://nbviewer.ipython.org/github/bwparry202/PHYS202-S14/blob/master/GalaxyMergers/GalaxyMergersFinal.ipynb>, accessed 2 July 2014.
- [Nelson2014] P.C. Nelson, "Schelling Model", <http://nbviewer.ipython.org/github/pcnelson202/PHYS202-S14/blob/master/IPython/SchellingModel.ipynb>, accessed 2 July 2014.
- [Parker2014] J.Parker, "Schelling Model", <http://nbviewer.ipython.org/github/jparke08/PHYS202-S14/blob/master/SchellingModel.ipynb>, accessed 2 July 2014.
- [Sonoma2013] "2013 Annual Meeting of the American Physical Society, California-Nevada Section", <http://epo.sonoma.edu/aps/index.html>, accessed 2 July 2014.
- [VanAtta2013] John Van Atta, "3-D Trajectory Generation in Hexagonal Geometry", <http://nbviewer.ipython.org/github/jvanatta/PHYS202-S13/blob/master/project/3dtracks.ipynb>, accessed 2 July 2014.
- [Rexrode2014] Chad Rexrode, "Monte-Carlo Glauber Model Simulations of Nuclear Collisions", <http://nbviewer.ipython.org/github/crexrode/PHYS202-S13/blob/master/SeniorProject/MCGlauber.ipynb>, accessed 2 July 2014.
- [MCGlauber] "MCGlauber: An Open-source IPython-based Monte Carlo Glauber Model of Nuclear Collisions", <https://github.com/MCGlauber>, accessed 2 July 2014.

Hyperopt-Sklearn: Automatic Hyperparameter Configuration for Scikit-Learn

Brent Komer^{‡*}, James Bergstra[‡], Chris Eliasmith[‡]



Abstract—Hyperopt-sklearn is a new software project that provides automatic algorithm configuration of the Scikit-learn machine learning library. Following Auto-Weka, we take the view that the choice of classifier and even the choice of preprocessing module can be taken together to represent a *single large hyperparameter optimization problem*. We use Hyperopt to define a search space that encompasses many standard components (e.g. SVM, RF, KNN, PCA, TFIDF) and common patterns of composing them together. We demonstrate, using search algorithms in Hyperopt and standard benchmarking data sets (MNIST, 20-Newsgroups, Convex Shapes), that searching this space is practical and effective. In particular, we improve on best-known scores for the model space for both MNIST and Convex Shapes.

Index Terms—bayesian optimization, model selection, hyperparameter optimization, scikit-learn

Introduction

The size of data sets and the speed of computers have increased to the point where it is often easier to fit complex functions to data using statistical estimation techniques than it is to design them by hand. The fitting of such functions (training machine learning algorithms) remains a relatively arcane art, typically mastered in the course of a graduate degree and years of experience. Recently however, techniques for automatic algorithm configuration based on Regression Trees [Hut11], Gaussian Processes [Moc78], [Sno12], and density-estimation techniques [Ber11] have emerged as viable alternatives to hand-tuning by domain specialists.

Hyperparameter optimization of machine learning systems was first applied to neural networks, where the number of parameters can be overwhelming. For example, [Ber11] tuned Deep Belief Networks (DBNs) with up to 32 hyperparameters, and [Ber13a] showed that similar methods could search a 238-dimensional configuration space describing multi-layer convolutional networks (convnets) for image classification.

Relative to DBNs and convnets, algorithms such as Support Vector Machines (SVMs) and Random Forests (RFs) have a small-enough number of hyperparameters that manual tuning and grid or random search provides satisfactory results. Taking a step back though, there is often no particular reason to use either an SVM or an RF when they are both computationally viable. A model-agnostic practitioner may simply prefer to go with the one that

provides greater accuracy. In this light, *the choice of classifier can be seen as hyperparameter* alongside the C -value in the SVM and the max-tree-depth of the RF. Indeed the choice and configuration of *preprocessing* components may likewise be seen as part of the model selection / hyperparameter optimization problem.

The Auto-Weka project [Tho13] was the first to show that an entire library of machine learning approaches (Weka [Hal09]) can be searched within the scope of a single run of hyperparameter tuning. However, Weka is a GPL-licensed Java library, and was not written with scalability in mind, so we feel there is a need for alternatives to Auto-Weka. Scikit-learn [Ped11] is another library of machine learning algorithms. Is written in Python (with many modules in C for greater speed), and is BSD-licensed. Scikit-learn is widely used in the scientific Python community and supports many machine learning application areas.

With this paper we introduce Hyperopt-Sklearn: a project that brings the benefits of automatic algorithm configuration to users of Python and scikit-learn. Hyperopt-Sklearn uses Hyperopt [Ber13b] to describe a search space over possible configurations of Scikit-Learn components, including preprocessing and classification modules. Section 2 describes our configuration space of 6 classifiers and 5 preprocessing modules that encompasses a strong set of classification systems for dense and sparse feature classification (of images and text). Section 3 presents experimental evidence that search over this space is viable, meaningful, and effective. Section 4 presents a discussion of the results, and directions for future work.

Background: Hyperopt for Optimization

The Hyperopt library [Ber13b] offers optimization algorithms for search spaces that arise in algorithm configuration. These spaces are characterized by a variety of types of variables (continuous, ordinal, categorical), different sensitivity profiles (e.g. uniform vs. log scaling), and conditional structure (when there is a choice between two classifiers, the parameters of one classifier are irrelevant when the other classifier is chosen). To use Hyperopt, a user must define/choose three things:

- 1) a search domain,
- 2) an objective function,
- 3) an optimization algorithm.

The search domain is specified via random variables, whose distributions should be chosen so that the most promising combinations have high prior probability. The search domain can include Python operators and functions that combine random

* Corresponding author: bjkomer@uwaterloo.ca

‡ Centre for Theoretical Neuroscience, University of Waterloo

variables into more convenient data structures for the objective function. The objective function maps a joint sampling of these random variables to a scalar-valued score that the optimization algorithm will try to *minimize*. Having chosen a search domain, an objective function, and an optimization algorithm, Hyperopt’s `fmin` function carries out the optimization, and stores results of the search to a database (e.g. either a simple Python list or a MongoDB instance). The `fmin` call carries out the simple analysis of finding the best-performing configuration, and returns that to the caller. The `fmin` call can use multiple workers when using the MongoDB backend, to implement parallel model selection on a compute cluster.

Scikit-Learn Model Selection as a Search Problem

Model selection is the process of estimating which machine learning model performs best from among a possibly infinite set of possibilities. As an optimization problem, the search domain is the set of valid assignments to the configuration parameters (hyperparameters) of the machine learning model, and the objective function is typically cross-validation, the negative degree of success on held-out examples. Practitioners usually address this optimization by hand, by grid search, or by random search. In this paper we discuss solving it with the Hyperopt optimization library. The basic approach is to set up a search space with random variable hyperparameters, use scikit-learn to implement the objective function that performs model training and model validation, and use Hyperopt to optimize the hyperparameters.

Scikit-learn includes many algorithms for classification (classifiers), as well as many algorithms for preprocessing data into the vectors expected by classification algorithms. Classifiers include for example, K-Neighbors, SVM, and RF algorithms. Preprocessing algorithms include things like component-wise Z-scaling (Normalizer) and Principle Components Analysis (PCA). A full classification algorithm typically includes a series of preprocessing steps followed by a classifier. For this reason, scikit-learn provides a *pipeline* data structure to represent and use a sequence of preprocessing steps and a classifier as if they were just one component (typically with an API similar to the classifier). Although hyperopt-sklearn does not formally use scikit-learn’s pipeline object, it provides related functionality. Hyperopt-sklearn provides a parameterization of a *search space* over pipelines, that is, of sequences of preprocessing steps and classifiers.

The configuration space we provide includes six preprocessing algorithms and seven classification algorithms. The full search space is illustrated in Figure 1. The preprocessing algorithms were (by class name, followed by n. hyperparameters + n. unused hyperparameters): `PCA(2)`, `StandardScaler(2)`, `MinMaxScaler(1)`, `Normalizer(1)`, `None`, and `TF-IDF(0+9)`. The first four preprocessing algorithms were for dense features. PCA performed whitening or non-whitening principle components analysis. The `StandardScaler`, `MinMaxScaler`, and `Normalizer` did various feature-wise affine transforms to map numeric input features onto values near 0 and with roughly unit variance. The `TF-IDF` preprocessing module performed feature extraction from text data. The classification algorithms were (by class name (used + unused hyperparameters)): `SVC(23)`, `KNN(4+5)`, `RandomForest(8)`, `ExtraTrees(8)`, `SGD(8+4)`, and `MultinomialNB(2)`. The `SVC` module is a fork of `LibSVM`, and our wrapper has 23 hyperparameters because we

treated each possible kernel as a different classifier, with its own set of hyperparameters: `Linear(4)`, `RBF(5)`, `Polynomial(7)`, and `Sigmoid(6)`. In total, our parameterization has 65 hyperparameters: 6 for preprocessing and 53 for classification. The search space includes 15 boolean variables, 14 categorical, 17 discrete, and 19 real-valued variables.

Although the total number of hyperparameters is large, the number of *active* hyperparameters describing any one model is much smaller: a model consisting of PCA and a `RandomForest` for example, would have only 12 active hyperparameters (1 for the choice of preprocessing, 2 internal to PCA, 1 for the choice of classifier and 8 internal to the RF). Hyperopt description language allows us to differentiate between *conditional* hyperparameters (which must always be assigned) and *non-conditional* hyperparameters (which may remain unassigned when they would be unused). We make use of this mechanism extensively so that Hyperopt’s search algorithms do not waste time learning by trial and error that e.g. RF hyperparameters have no effect on SVM performance. Even internally within classifiers, there are instances of conditional parameters: `KNN` has conditional parameters depending on the distance metric, and `LinearSVC` has 3 binary parameters (`loss`, `penalty`, and `dual`) that admit only 4 valid joint assignments. We also included a blacklist of (preprocessing, classifier) pairs that did not work together, e.g. PCA and `MinMaxScaler` were incompatible with `MultinomialNB`, TF-IDF could only be used for text data, and the tree-based classifiers were not compatible with the sparse features produced by the TF-IDF preprocessor. Allowing for a 10-way discretization of real-valued hyperparameters, and taking these conditional hyperparameters into account, a grid search of our search space would still require an infeasible number of evaluations (on the order of 10^{12}).

Finally, the search space becomes an optimization problem when we also define a scalar-valued search *objective*. Hyperopt-sklearn uses scikit-learn’s `score` method on *validation data* to define the search criterion. For classifiers, this is the so-called "Zero-One Loss": the number of correct label predictions among data that has been withheld from the data set used for training (and also from the data used for testing *after* the model selection search process).

Example Usage

Following Scikit-learn’s convention, hyperopt-sklearn provides an `Estimator` class with a `fit` method and a `predict` method. The `fit` method of this class performs hyperparameter optimization, and after it has completed, the `predict` method applies the best model to test data. Each evaluation during optimization performs training on a large fraction of the training set, estimates test set accuracy on a validation set, and returns that validation set score to the optimizer. At the end of search, the best configuration is retrained on the whole data set to produce the classifier that handles subsequent `predict` calls.

One of the important goals of hyperopt-sklearn is that it is easy to learn and to use. To facilitate this, the syntax for fitting a classifier to data and making predictions is very similar to scikit-learn. Here is the simplest example of using this software.

```
from hpsklearn import HyperoptEstimator
# Load data ({train,test}_{data,label})
# Create the estimator object
estim = HyperoptEstimator()
# Search the space of classifiers and preprocessing
# steps and their respective hyperparameters in
```

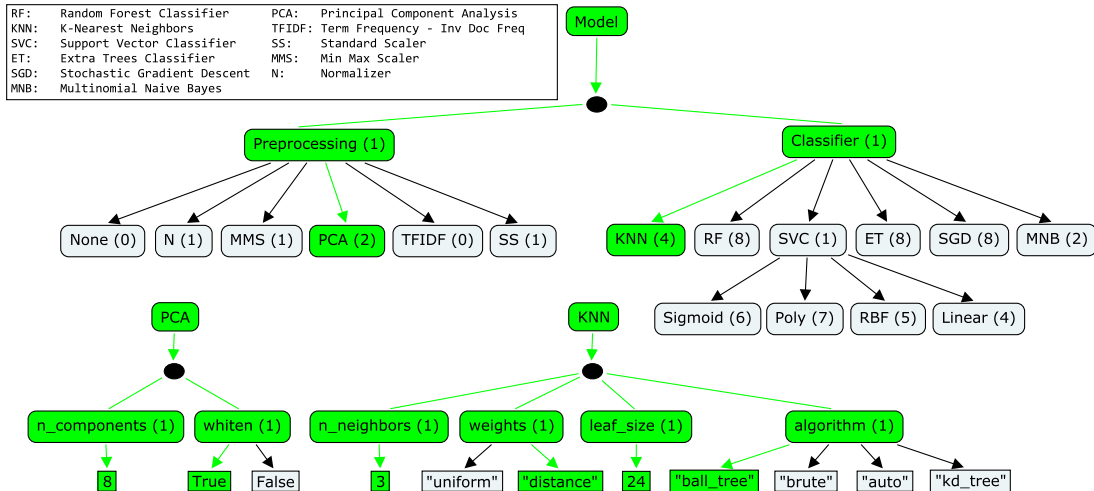


Fig. 1: Hyperopt-sklearn's full search space ("Any Classifier") consists of a (preprocessing, classifier) pair. There are 6 possible preprocessing modules and 6 possible classifiers. Choosing a model within this configuration space means choosing paths in an ancestral sampling process. The highlighted green edges and nodes represent a (PCA, K-Nearest Neighbor) model. The number of active hyperparameters in a model is the sum of parenthetical numbers in the selected boxes. For the PCA+KNN combination, 7 hyperparameters are activated.

```
# scikit-learn to fit a model to the data
estim.fit(train_data, train_label)
# Make a prediction using the optimized model
prediction = estim.predict(unknown_data)
# Report the accuracy of the classifier
# on a given set of data
score = estim.score(test_data, test_label)
# Return instances of the classifier and
# preprocessing steps
model = estim.best_model()
```

The `HyperoptEstimator` object contains the information of what space to search as well as how to search it. It can be configured to use a variety of hyperparameter search algorithms and also supports using a combination of algorithms. Any algorithm that supports the same interface as the algorithms in hyperopt can be used here. This is also where you, the user, can specify the maximum number of function evaluations you would like to be run as well as a timeout (in seconds) for each run.

```
from hpsklearn import HyperoptEstimator
from hyperopt import tpe
estim = HyperoptEstimator(algo=tpe.suggest,
                          max_evals=150,
                          trial_timeout=60)
```

Each search algorithm can bring its own bias to the search space, and it may not be clear that one particular strategy is the best in all cases. Sometimes it can be helpful to use a mixture of search algorithms.

```
from hpsklearn import HyperoptEstimator
from hyperopt import anneal, rand, tpe, mix
# define an algorithm that searches randomly 5% of
# the time, uses TPE 75% of the time, and uses
# annealing 20% of the time
mix_algo = partial(mix.suggest, p_suggest=[
    (0.05, rand.suggest),
    (0.75, tpe.suggest),
    (0.20, anneal.suggest)])
estim = HyperoptEstimator(algo=mix_algo,
                          max_evals=150,
                          trial_timeout=60)
```

Searching effectively over the entire space of classifiers available in scikit-learn can use a lot of time and computational resources. Sometimes you might have a particular subspace of models that they are more interested in. With hyperopt-sklearn it is possible to

specify a more narrow search space to allow it to be explored in greater depth.

```
from hpsklearn import HyperoptEstimator, svc
# limit the search to only models a SVC
estim = HyperoptEstimator(classifier=svc('my_svc'))
```

Combinations of different spaces can also be used.

```
from hpsklearn import HyperoptEstimator, svc, knn, \
from hyperopt import hp
# restrict the space to contain only random forest,
# k-nearest neighbors, and SVC models.
clf = hp.choice('my_name',
               [random_forest('my_name.random_forest'),
                svc('my_name.svc'),
                knn('my_name.knn')])
estim = HyperoptEstimator(classifier=clf)
```

The support vector machine provided by scikit-learn has a number of different kernels that can be used (linear, rbf, poly, sigmoid). Changing the kernel can have a large effect on the performance of the model, and each kernel has its own unique hyperparameters. To account for this, hyperopt-sklearn treats each kernel choice as a unique model in the search space. If you already know which kernel works best for your data, or you are just interested in exploring models with a particular kernel, you may specify it directly rather than going through the `svc`.

```
from hpsklearn import HyperoptEstimator, svc_rbf
estim = HyperoptEstimator(
    classifier=svc_rbf('my_svc'))
```

It is also possible to specify which kernels you are interested in by passing a list to the `svc`.

```
from hpsklearn import HyperoptEstimator, svc
estim = HyperoptEstimator(
    classifier=svc('my_svc',
                  kernels=['linear',
                           'sigmoid']))
```

In a similar manner to classifiers, the space of preprocessing modules can be fine tuned. Multiple successive stages of preprocessing can be specified by putting them in a list. An empty list means that no preprocessing will be done on the data.

```
from hpsklearn import HyperoptEstimator, pca
estim = HyperoptEstimator(
    preprocessing=[pca('my_pca')])
```

Combinations of different spaces can be used here as well.

```
from hpsklearn import HyperoptEstimator, tfidf, pca
from hyperopt import hp
preproc = hp.choice('my_name',
[[pca('my_name.pca')],
[pca('my_name.pca'), normalizer('my_name.norm')],
[standard_scaler('my_name.std_scaler')],
[]])
estim = HyperoptEstimator( preprocessing=preproc )
```

Some types of preprocessing will only work on specific types of data. For example, the `TfidfVectorizer` that scikit-learn provides is designed to work with text data and would not be appropriate for other types of data. To address this, `hyperopt-sklearn` comes with a few pre-defined spaces of classifiers and preprocessing tailored to specific data types.

```
from hpsklearn import HyperoptEstimator, \
    any_sparse_classifier, \
    any_text_preprocessing
from hyperopt import tpe
estim = HyperoptEstimator(
    algo=tpe.suggest,
    classifier=any_sparse_classifier('my_clf')
    preprocessing=any_text_preprocessing('my_pp')
    max_evals=200,
    trial_timeout=60 )
```

So far in all of these examples, every hyperparameter available to the model is being searched over. It is also possible for you to specify the values of specific hyperparameters, and those parameters will remain constant during the search. This could be useful, for example, if you knew you wanted to use whitened PCA data and a degree-3 polynomial kernel SVM.

```
from hpsklearn import HyperoptEstimator, pca, svc_poly
estim = HyperoptEstimator(
    preprocessing=[pca('my_pca', whiten=True)],
    classifier=svc_poly('my_poly', degree=3))
```

It is also possible to specify ranges of individual parameters. This is done using the standard `hyperopt` syntax. These will override the defaults defined within `hyperopt-sklearn`.

```
from hpsklearn import HyperoptEstimator, pca, sgd
from hyperopt import hp
import numpy as np
sgd_loss = hp.pchoice('loss',
[[ (0.50, 'hinge'),
(0.25, 'log'),
(0.25, 'huber')]])
sgd_penalty = hp.choice('penalty',
['l2', 'elasticnet'])
sgd_alpha = hp.loguniform('alpha',
low=np.log(1e-5),
high=np.log(1) )
estim = HyperoptEstimator(
    classifier=sgd('my_sgd',
    loss=sgd_loss,
    penalty=sgd_penalty,
    alpha=sgd_alpha) )
```

All of the components available to the user can be found in the `components.py` file. A complete working example of using `hyperopt-sklearn` to find a model for the 20 newsgroups data set is shown below.

```
from hpsklearn import HyperoptEstimator, tfidf, \
    any_sparse_classifier
from sklearn.datasets import fetch_20newsgroups
from hyperopt import tpe
import numpy as np
# Download data and split training and test sets
train = fetch_20newsgroups(subset='train')
test = fetch_20newsgroups(subset='test')
X_train = train.data
```

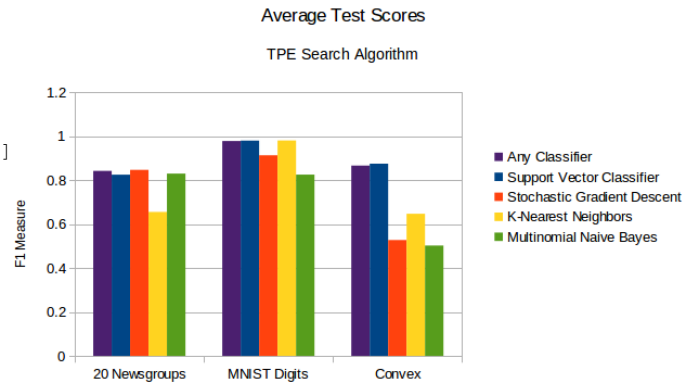


Fig. 2: For each data set, searching the full configuration space (“Any Classifier”) delivered performance approximately on par with a search that was restricted to the best classifier type. (Best viewed in color.)

```
y_train = train.target
X_test = test.data
y_test = test.target
estim = HyperoptEstimator(
    classifier=any_sparse_classifier('clf'),
    preprocessing=[tfidf('tfidf')],
    algo=tpe.suggest,
    trial_timeout=180)
estim.fit(X_train, y_train)
print(estim.score(X_test, y_test))
print(estim.best_model())
```

Experiments

We conducted experiments on three data sets to establish that `hyperopt-sklearn` can find accurate models on a range of data sets in a reasonable amount of time. Results were collected on three data sets: MNIST, 20-Newsgroups, and Convex Shapes. MNIST is a well-known data set of 70K 28x28 greyscale images of hand-drawn digits [Lec98]. 20-Newsgroups is a 20-way classification data set of 20K newsgroup messages ([Mit96], we did not remove the headers for our experiments). Convex Shapes is a binary classification task of distinguishing pictures of convex white-colored regions in small (32x32) black-and-white images [Lar07].

Figure 2 shows that there was no penalty for searching broadly. We performed optimization runs of up to 300 function evaluations searching the entire space, and compared the quality of solution with specialized searches of specific classifier types (including best known classifiers).

Figure 3 shows that search could find different, good models. This figure was constructed by running `hyperopt-sklearn` with different initial conditions (number of evaluations, choice of optimization algorithm, and random number seed) and keeping track of what final model was chosen after each run. Although support vector machines were always among the best, the parameters of best SVMs looked very different across data sets. For example, on the image data sets (MNIST and Convex) the SVMs chosen never had a sigmoid or linear kernel, while on 20 newsgroups the linear and sigmoid kernel were often best.

Discussion and Future Work

Table 1 lists the test set scores of the best models found by cross-validation, as well as some points of reference from previous work.

MNIST		20 Newsgroups		Convex Shapes	
Approach	Accuracy	Approach	F-Score	Approach	Accuracy
Committee of convnets	99.8%	CFC	0.928	hyperopt-sklearn	88.7%
hyperopt-sklearn	98.7%	hyperopt-sklearn	0.856	hp-dbnet	84.6%
libSVM grid search	98.6%	SVMTorch	0.848	dbn-3	81.4%
Boosted trees	98.5%	LibSVM	0.843		

TABLE 1: Hyperopt-sklearn scores relative to selections from literature on the three data sets used in our experiments. On MNIST, hyperopt-sklearn is one of the best-scoring methods that does not use image-specific domain knowledge (these scores and others may be found at <http://yann.lecun.com/exdb/mnist/>). On 20 Newsgroups, hyperopt-sklearn is competitive with similar approaches from the literature (scores taken from [Gua09]). In the 20 Newsgroups data set, the score reported for hyperopt-sklearn is the weighted-average F1 score provided by sklearn. The other approaches shown here use the macro-average F1 score. On Convex Shapes, hyperopt-sklearn outperforms previous automatic algorithm configuration approaches [Egg13] and manual tuning [Lar07].

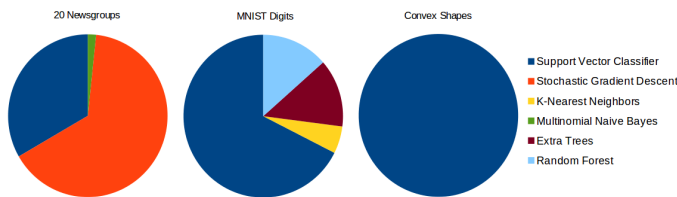


Fig. 3: Looking at the best models from all optimization runs performed on the full search space (using different initial conditions, and different optimization algorithms) we see that different data sets are handled best by different classifiers. SVC was the only classifier ever chosen as the best model for Convex Shapes, and was often found to be best on MNIST and 20 Newsgroups, however the best SVC parameters were very different across data sets.

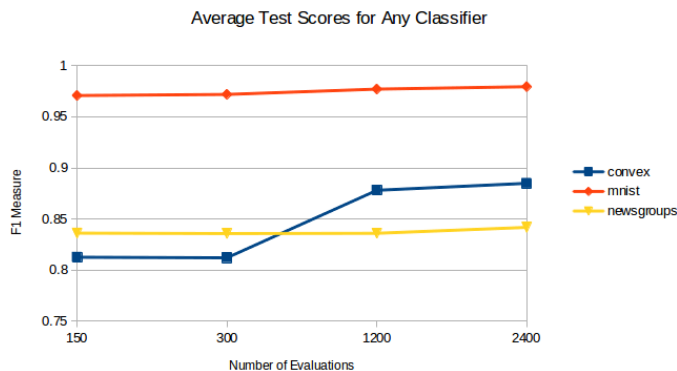


Fig. 4: Using Hyperopt’s Anneal search algorithm, increasing the number of function evaluations from 150 to 2400 lead to a modest improvement in accuracy on 20 Newsgroups and MNIST, and a more dramatic improvement on Convex Shapes. We capped evaluations to 5 minutes each so 300 evaluations took between 12 and 24 hours of wall time.

Hyperopt-sklearn’s scores are relatively good on each data set, indicating that with hyperopt-sklearn’s parameterization, Hyperopt’s optimization algorithms are competitive with human experts.

The model with the best performance on the MNIST Digits data set uses deep artificial neural networks. Small receptive fields of convolutional winner-take-all neurons build up the large network. Each neural column becomes an expert on inputs pre-processed in different ways, and the average prediction of 35 deep neural columns to come up with a single final prediction [Cir12]. This model is much more advanced than those available in scikit-

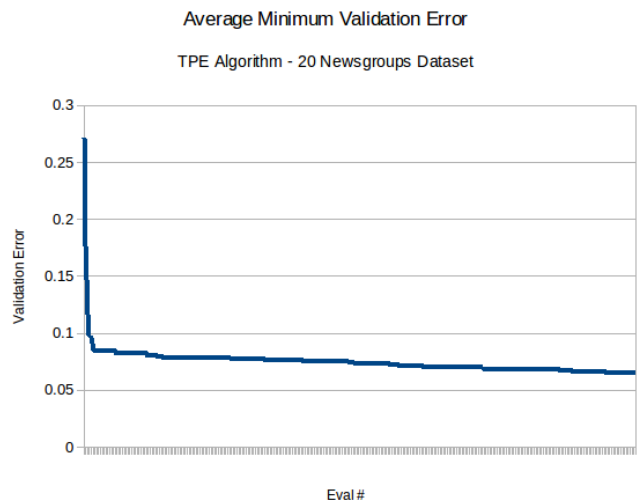


Fig. 5: Right: TPE makes gradual progress on 20 Newsgroups over 300 iterations and gives no indication of convergence.

learn. The previously best known model in the scikit-learn search space is a radial-basis SVM on centered data that scores 98.6%, and hyperopt-sklearn matches that performance [MNIST].

The CFC model that performed quite well on the 20 newsgroups document classification data set is a Class-Feature-Centroid classifier. Centroid approaches are typically inferior to an SVM, due to the centroids found during training being far from the optimal location. The CFC method reported here uses a centroid built from the inter-class term index and the inner-class term index. It uses a novel combination of these indices along with a denormalized cosine measure to calculate the similarity score between the centroid and a text vector [Gua09]. This style of model is not currently implemented in hyperopt-sklearn, and our experiments suggest that existing hyperopt-sklearn components cannot be assembled to match its level of performance. Perhaps when it is implemented, Hyperopt may find a set of parameters that provides even greater classification accuracy.

On the Convex Shapes data set, our Hyperopt-sklearn experiments revealed a more accurate model than was previously believed to exist in any search space, let alone a search space of such standard components. This result underscores the difficulty and importance of hyperparameter search.

Hyperopt-sklearn provides many opportunities for future work: more classifiers and preprocessing modules could be included in the search space, and there are more ways to combine even the existing components. Other types of data require different preprocessing, and other prediction problems exist beyond classification. In expanding the search space, care must be taken to ensure that the benefits of new models outweigh the greater difficulty of searching a larger space. There are some parameters that scikit-learn exposes that are more implementation details than actual hyperparameters that affect the fit (such as `algorithm` and `leaf_size` in the KNN model). Care should be taken to identify these parameters in each model and they may need to be treated differently during exploration.

It is possible for a user to add their own classifier to the search space as long as it fits the scikit-learn interface. This currently requires some understanding of how hyperopt-sklearn’s code is structured and it would be nice to improve the support for this so minimal effort is required by the user. We also plan to allow the user to specify alternate scoring methods besides just accuracy and F-measure, as there can be cases where these are not best suited to the particular problem.

We have shown here that Hyperopt’s random search, annealing search, and TPE algorithms make Hyperopt-sklearn viable, but the slow convergence in e.g. Figure 4 and 5 suggests that other optimization algorithms might be more call-efficient. The development of Bayesian optimization algorithms is an active research area, and we look forward to looking at how other search algorithms interact with hyperopt-sklearn’s search spaces. Hyperparameter optimization opens up a new art of matching the parameterization of search spaces to the strengths of search algorithms.

Computational wall time spent on search is of great practical importance, and hyperopt-sklearn currently spends a significant amount of time evaluating points that are un-promising. Techniques for recognizing bad performers early could speed up search enormously [Swe14], [Dom14]. Relatedly, hyperopt-sklearn currently lacks support for K-fold cross-validation. In that setting, it will be crucial to follow SMAC in the use of racing algorithms to skip un-necessary folds.

Conclusions

We have introduced Hyperopt-sklearn, a Python package for automatic algorithm configuration of standard machine learning algorithms provided by Scikit-Learn. Hyperopt-sklearn provides a unified view of 6 possible preprocessing modules and 6 possible classifiers, yet with the help of Hyperopt’s optimization functions it is able to both rival and surpass human experts in algorithm configuration. We hope that it provides practitioners with a useful tool for the development of machine learning systems, and automatic machine learning researchers with benchmarks for future work in algorithm configuration.

Acknowledgements

This research was supported by the NSERC Banting Fellowship program, the NSERC Engage Program and by D-Wave Systems. Thanks also to Hristijan Bogoevski for early drafts of a hyperopt-to-scikit-learn bridge.

REFERENCES

- [Ber11] J. Bergstra, R. Bardenet, Y. Bengio, and B. Kegl. *Algorithms for hyper-parameter optimization*, NIPS, 24:2546–2554, 2011.
- [Ber13a] J. Bergstra, D. Yamins, and D. D. Cox. *Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures*. In Proc. ICML, 2013a.
- [Ber13b] J. Bergstra, D. Yamins, and D. D. Cox. *Hyperopt: A Python library for optimizing the hyperparameters of machine learning algorithms*, SciPy’13, 2013b.
- [Cir12] D. Ciresan, U. Meier, and J. Schmidhuber. *Multi-column Deep Neural Networks for Image Classification*, IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 3642-3649. 2012.
- [Dom14] T. Domhan, T. Springenberg, F. Hutter. *Extrapolating Learning Curves of Deep Neural Networks*, ICML AutoML Workshop, 2014.
- [Egg13] K. Eggensperger, M. Feurer, F. Hutter, J. Bergstra, J. Snoek, H. Hoos, and K. Leyton-Brown. *Towards an empirical foundation for assessing bayesian optimization of hyperparameters*, NIPS workshop on Bayesian Optimization in Theory and Practice, 2013.
- [Gua09] H. Guan, J. Zhou, and M. Guo. *A class-feature-centroid classifier for text categorization*, Proceedings of the 18th international conference on World wide web, 201-210. ACM, 2009.
- [Hal09] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. *The weka data mining software: an update*, ACM SIGKDD explorations newsletter, 11(1):10-18, 2009.
- [Hut11] F. Hutter, H. Hoos, and K. Leyton-Brown. *Sequential model-based optimization for general algorithm configuration*, LION-5, 2011. Extended version as UBC Tech report TR-2010-10.
- [Lar07] H. Larochelle, D. Erhan, A. Courville, J. Bergstra, and Y. Bengio. *An empirical evaluation of deep architectures on problems with many factors of variation*, ICML, 473-480, 2007.
- [Lec98] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. *Gradient-based learning applied to document recognition*, Proceedings of the IEEE, 86(11):2278-2324, November 1998.
- [Mit96] T. Mitchell. *20 newsgroups data set*, <http://qwone.com/jason/20Newsgroups/>, 1996.
- [Moc78] J. Mockus, V. Tiesis, and A. Zilinskas. *The application of Bayesian methods for seeking the extremum*, L.C.W. Dixon and G.P. Szego, editors, Towards Global Optimization, volume 2, pages 117–129. North Holland, New York, 1978.
- [MNIST] The MNIST Database of handwritten digits: <http://yann.lecun.com/exdb/mnist/>
- [Ped11] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. *Scikit-learn: Machine Learning in Python*, Journal of Machine Learning Research, 12:2825–2830, 2011.
- [Sno12] J. Snoek, H. Larochelle, and R. P. Adams. *Practical Bayesian optimization of machine learning algorithms*, Neural Information Processing Systems, 2012.
- [Swe14] K. Swersky, J. Snoek, R.P. Adams. *Freeze-Thaw Bayesian Optimization*, arXiv:1406.3896, 2014.
- [Tho13] C. Thornton, F. Hutter, H. H. Hoos, and K. Leyton-Brown. *Auto-WEKA: Automated selection and hyper-parameter optimization of classification algorithms*, KDD 847-855, 2013.

Python Coding of Geospatial Processing in Web-based Mapping Applications

James A. Kuiper^{‡*}, Andrew J. Ayers[‡], Michael E. Holm[‡], Michael J. Nowak[‡]



Abstract—Python has powerful capabilities for coding elements of Web-based mapping applications. This paper highlights examples of analytical geospatial processing services that we have implemented for several Open Source-based development projects, including the Eastern Interconnection States' Planning Council (EISPC) Energy Zones Mapping Tool (<http://eispctools.anl.gov>), the Solar Energy Environmental Mapper (<http://solarmapper.anl.gov>), and the Ecological Risk Calculator (<http://bogi.evs.anl.gov/erc/portal>). We used common Open Source tools such as GeoServer, PostGIS, GeoExt, and OpenLayers for the basic Web-based portal, then added custom analytical tools to support more advanced functionality. The analytical processes were implemented as Web Processing Services (WPSs) running on PyWPS, a Python implementation of the Open Geospatial Consortium (OGC) WPS. For report tools, areas drawn by the user in the map interface are submitted to a service that utilizes the spatial extensions of PostGIS to generate buffers for use in querying and analyzing the underlying data. Python code then post-processes the results and outputs JavaScript Object Notation (JSON)-formatted data for rendering. We made use of PyWPS's integration with the Geographic Resources Analysis Support System (GRASS) to implement flexible, user-adjustable suitability models for several renewable energy generation technologies. In this paper, we provide details about the processing methods we used within these project examples.

Index Terms—GIS, web-based mapping, PyWPS, PostGIS, GRASS, spatial modeling

BACKGROUND AND INTRODUCTION

Web-based mapping applications are effective in providing simple and accessible interfaces for geospatial information, and often include large spatial databases and advanced analytical capabilities. Perhaps the most familiar is Google Maps [Ggl] which provides access to terabytes of maps, aerial imagery, street address data, and point-to-point routing capabilities. Descriptions are included herein of several Web-based applications that focus on energy and environmental data and how their back-end geoprocessing services were built with Python.

The Eastern Interconnection States' Planning Council (EISPC) Energy Zones Mapping Tool (EZMT) [Ezmt] was developed primarily to facilitate identification of potential energy zones or areas of high resource concentration for nine different low- or no-carbon energy resources, spanning more than 30 grid-scale energy generation technologies. The geographic scope is the Eastern Interconnection (EI), the electrical grid that serves the eastern

United States and parts of Canada. The EZMT includes more than 250 map layers, a flexible suitability modeling capability with more than 35 pre-configured models and 65 input modeling layers, and 19 reports that can be run for user-specified areas within the EI. More background about the project is available from [Arg13].

Solar Energy Environmental Mapper (Solar Mapper) [Sol] provides interactive mapping data on utility-scale solar energy resources and related siting factors in the six southwestern states studied in the Solar Energy Development Programmatic Environmental Impact Statement [DOI12]. The application was first launched in December 2010, and a version that has been reengineered with open-source components is scheduled for launch in June 2014. Solar Mapper supports identification and screening-level analyses of potential conflicts between development and environmental resources, and is designed primarily for use by regulating agencies, project planners, and public stakeholders. More details about Solar Mapper can be found in [Sol13].

The Ecological Risk Calculator (ERC) [Erc] estimates risk in individual watersheds in the western United States to federally listed threatened and endangered species, and their designated critical habitats from energy-related surface and groundwater withdrawals. The approach takes into account several biogeographical characteristics of watersheds including occupancy, distribution, and imperilment of species, and their sensitivity to impacts from water withdrawals, as well as geophysical characteristics of watersheds known to include designated critical habitats for species of concern. The ERC is intended to help project planners identify potential levels of conflicts related to listed species (and thus the associated regulatory requirements), and is intended to be used as a preliminary screening tool.

Each of these Web-based mapping applications includes both vector (geographic data stored using coordinates) and raster (geographic data stored as a matrix of equally sized cells) spatial data stored in a relational database. For each application, Python was used to add one or more custom geoprocessing, modeling, or reporting services. The following section provides background on the software environment used, followed by specific examples of code with a discussion about the unique details in each.

One of the distinctive elements of geographic data management and processing is the need for coordinate reference systems and coordinate transformations (projections), which are needed to represent areas on the earth's oblate spheroid shape as planar maps and to manage data in Cartesian coordinate systems. These references appear in the code examples as "3857," the European Petroleum Survey Group (EPSG) Spatial Reference ID (SRID) reference for WGS84 Web Mercator (Auxiliary Sphere)

* Corresponding author: jkuiper@anl.gov

‡ Argonne National Laboratory

and "102003," the USA Contiguous Albers Equal Area Conic projection commonly used for multi-state and national maps of the United States. These standardized EPSG definitions are now maintained by the International Association of Oil & Gas Producers (OGP) Surveying & Positioning Committee [OGP].

The Web Mercator projection has poor properties for many elements of mapping and navigation [NGA] but is used for most current Web-based mapping applications because of the wide availability of high-quality base maps in the Web Mercator projection from providers such as Google Maps. In the Solar Mapper project, we compared area computations in the southwestern United States using Web Mercator against the Albers Equal Area projection and found very large discrepancies in the results (Table 1).

The distortion inherent in world-scale Mercator projections is easily seen by the horizontal expansion of features, which increases dramatically in the higher northern and southern latitudes. In each of our projects, we chose to store local geographic data in Web Mercator to match the base maps and increase performance. However, for geographic processing such as generating buffers and computing lengths and areas, we first convert coordinates to the Albers Equal Area projection to take advantage of the improved properties of that projection.

SOFTWARE ENVIRONMENT

Each of these systems was built with a multi-tier architecture composed of a Javascript/HTML (hypertext markup language) interface built on Bootstrap [Btsrp], OpenLayers [OpLyr], and ExtJS [Sen]; a Web application tier built on Ruby on Rails [RoR]; a mapping tier implemented with GeoServer [Gsrvr]; a persistence tier implemented with PostGIS [PGIS]; and an analysis tier built on Python, PyWPS [PyWPS], GRASS [GRASS], and the spatial analysis functionality of PostGIS. These systems are deployed on Ubuntu [Ub] virtual machines running in a private VMware [VM] cloud. The Python-orchestrated analysis tier is the focus of this paper.

Many of the examples show geospatial operations using PostGIS. PostGIS extends the functionality of PostgreSQL for raster and vector spatial data with a robust library of functions. We found it to be well documented, reliable, and the "footprint analysis" tools we describe in the examples run significantly faster than similar tools we had previously developed with a popular commercial GIS framework.

EXAMPLES

One of the primary capabilities of each of our Web applications was using an area selected or drawn by the user for analysis (a "footprint"); collecting vector and raster data inside, intersecting, or near the footprint; and compiling it in a report. The first example shows the steps followed through the whole process, including the user interface, and later examples concentrate on refinements of the Python-coded steps.

Full Process for Footprint Analysis of Power Plant Locations Stored as Point Features

This example is from the EZMT and illustrates part of its Power Plant report. The user draws an area of interest over the map (Figure 1) and specifies other report parameters (Figure 2). The "Launch Report" button submits a request to the Web application

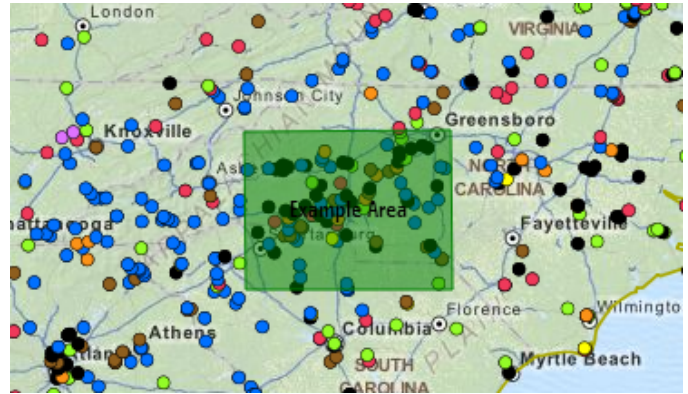


Fig. 1: EZMT Interface View of User-Specified Analysis Area and Power Plant Points

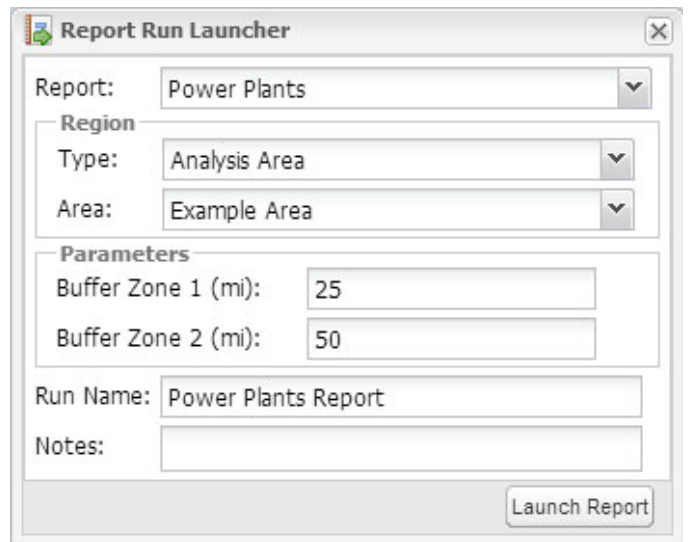


Fig. 2: EZMT Interface View of the Report Run Launcher

server to schedule, launch, track, and manage the report's execution.

The Web application initiates the report run by making a WPS request to the service, which is implemented in PyWPS. The request is an XML (extensible markup language) document describing the WPS "Execute" operation and is submitted via a hypertext transfer protocol (HTTP) POST. PyWPS receives this POST request, performs some basic validation and preprocessing, and routes the request to the custom `WPSProcess` implementation for that request. PyWPS then prepares the HTTP response and returns it to the application server. The code below illustrates the major steps used to generate the data for the report.

We use the `psycopg2` library to interact with the database, including leveraging the geographic information system (GIS) capabilities of PostGIS.

```
# Import PostgreSQL library for database queries
import psycopg2
```

The user-specified footprint corresponding to Figure 1 is hard-coded in this example with Web Mercator coordinates specified in meters and using the Well-Known Text (WKT) format.

```
# Footprint specified in WKT with web Mercator
# coordinates
fp_webmerc = "POLYGON((-9152998.67 4312042.45,
-8866818.44 4319380.41,-8866818.44 4099241.77,
```

Projection	Area (square miles)		
	Large Horizontal Area	Large Vertical Area	Smaller Square Area
Albers Equal Area	7,498.7	10,847.3	35.8
Web Mercator	13,410.0	18,271.4	63.0
Difference	5,911.3	7,424.1	27.2
Percent Difference	44%	41%	43%

TABLE 1: Comparison of Area Computations between the Web Mercator Projection and the Albers Equal Area Projection in the Southwestern United States

```
-9143214.73 4101687.75,-9152998.67 4312042.45))"
# Input GIS data
layer="power_plant_platts_existing"

A database connection is then established, and a cursor is created.

# Make database connection and cursor
conn = psycopg2.connect(host=pg_host,
    database=pg_database, user=pg_user,
    password=pg_password)
cur = self.conn().cursor()

Structured Query Language (SQL) is used to (1) convert the
Web Mercator footprint to the Albers Equal Area projection, (2)
generate a buffer around the Albers version of the footprint, and
(3) convert that buffer back to Web Mercator. In these sections,
ST_GeomFromText converts WKT to binary geometry, and
ST_AsText converts binary geometry back to WKT. Because
WKT does not store projection information, it is given as a
parameter in ST_GeomFromText.

# Convert web Mercator footprint to Albers projection
# (equal area)
sql = "SELECT ST_AsText(ST_Transform("+
    "ST_GeomFromText('"+fp_webmerc+
    "', 3857), 102003))"
cur.execute(sql)
fp_albers = cur.fetchone()[0]

# Generate Albers projection buffer around footprint
sql = "SELECT ST_AsText(ST_Buffer("+
    "ST_GeomFromText('"+fp_albers+
    "', 102003), "+str(buffer_dist_m)+"))"
cur.execute(sql)
buffer_albers = cur.fetchone()[0]

# Convert buffer to web Mercator projection
# (rpt for second buffer)
sql = "SELECT ST_AsText(ST_Transform("+
    "ST_GeomFromText('"+
    buffer1_albers+"', 102003), 3857))"
cur.execute(sql)
buffer1_webmerc = cur.fetchone()[0]

The previous steps are handled similarly for every report in an
initialization method. The final SQL statement in this example
retrieves data for the report content itself. The ST_Intersects
method queries the geometries in the power plant layer and returns
the records intersecting (overlapping) the footprint. These records
are summarized [count(*), sum(opcap), and GROUP BY
energy_resource] to provide content for the initial graph and
table in the report. This SQL statement is repeated for the two
buffer distances around the footprint.

# Return records falling within footprint and the
# two buffer distances # (Repeat for two footprints)
sql = "SELECT energy_resource,count(*),sum(opcap) "+
    "FROM "+layer+" WHERE ST_Intersects("+
    layer+".geom, ST_GeomFromText('"+fp_webmerc+
    "', 3857)) GROUP BY energy_resource "+
    "ORDER BY energy_resource"
cur.execute(sql)
```

```
l = []
for row in cur:
    # Collect results in list...

Once the data have been retrieved, the code compiles it into a
Python dictionary which is rendered and returned as a JSON
document (excerpt below). This document is retained by the
application for eventual rendering into its final form, HTML with
the graphs built with ExtJS. Figure 3 shows a portion of the report.

# Combine data and return results as JSON.
import json

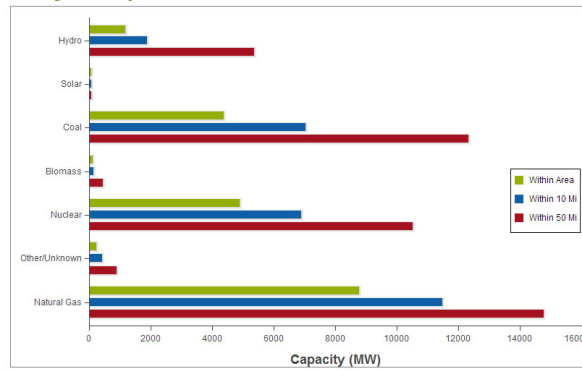
"existing_summary": {
    "header": [
        "EISPC Energy Resource Type",
        ...
    ],
    "data": {
        "Natural Gas": [11,8716.6,14,11408.5,20,14705.5],
        "Other/Unk": [36,186.135,39,365.335,48,838.185],
        "Nuclear": [2,4851.8,4,6843.3,6,10461.9],
        "Biomass": [7,77.3,11,97.3,17,397.08],
        "Coal": [5,4333.1,10,6971.8,24,12253.2],
        "Solar": [7,26.95,7,26.95,9,30.15],
        "Hydro": [36,1127.875,54,1829.675,82,5308.875]
    },
    "metadata": {
        "shortname": "power_plant_platts_existing",
        "feature_type": "point"
    }
}
```

Footprint Analysis of Transmission Lines Stored as Line Features

Another EISPC report uses a user-specified footprint to analyze electrical transmission line information; however, rather than only listing features inside the footprint as in the previous example, (1) in contrast to points, line features can cross the footprint boundary; and (2) we want to report the total length of the portion within the footprint rather than only listing the matching records. Note that ST_Intersects is used to collect the lines overlapping the footprint, whereas ST_Intersection is used to calculate lengths of only the portion of the lines within the footprint. In addition, the coordinates are transformed into the Albers Equal Area projection for the length computation.

```
sql = "SELECT category, COUNT(*),sum(ST_Length("+
    "ST_Transform(ST_Intersection("+layer+
    ".geom,ST_GeomFromText('"+fp_webmerc+
    "', 3857)), 102003))) AS sum_length_fp "+
    "FROM "+layer+" WHERE ST_Intersects("+layer+
    ".geom,ST_GeomFromText('"+fp_webmerc+
    "', 3857)) GROUP BY category ORDER BY category"
cur.execute(sql)
list = []
for row in cur:
    # Collect results in list of lists...
```

Existing: Summary



EISPC Energy Resource Type	Total Number within Analysis Area	Total Operating Capacity (MW) within Analysis Area	Total Number within 25 Miles of Analysis Area	Total Operating Capacity (MW) within 25 Miles of Analysis Area	Total Number within 50 Miles of Analysis Area	Total Operating Capacity (MW) within 50 Miles of Analysis Area
Natural Gas	11	8,716.60	14	11,408.50	20	14,705.50
Other/Unknown	36	186.14	39	365.34	48	838.18
Nuclear	2	4,851.80	4	6,843.30	6	10,461.90
Biomass	7	77.30	11	97.30	17	397.08
Coal	5	4,333.10	10	6,971.80	24	12,253.20
Solar	7	26.95	7	26.95	9	30.15
Hydro	36	1,127.88	54	1,829.68	82	5,308.68
TOTAL	104	19,319.74	139	27,542.86	206	43,994.89

Source: [Platts/Bentley Energy Power Plant - Existing](#)

Fig. 3: Portion of EZMT Power Plant Report

Results in JSON format:

```
{
  "existing_trans_sum": {
    "header": [
      "Voltage Category",
      "Total Length (mi) within Analysis Area",
      "Total Length (mi) within 1.0 Miles...",
      "Total Length (mi) within 5.0 Miles..."
    ],
    "data": {
      "115kV - 161kV": [209.24, 259.38, 477.57],
      "100kV or Lower": [124.94, 173.15, 424.08],
      "345kV - 450kV": [206.67, 239.55, 393.97]
    },
    "metadata": {
      "shortname": "transmission_line_platts",
      "feature_type": "multilinestring"
    }
  }
}
```

```
sql += " AND wps_runs.pywps_process_id = "
sql += str(procId)+" GROUP BY sma_code"
cur.execute(sql)
list = []
for row in cur:
    # Collect results in list of lists...
```

Footprint Analysis of Watershed Areas Stored as Polygon Features, with Joined Tables

The Environmental Risk Calculator [?] involves analysis of animal and plant species that have been formally designated by the United States as threatened or endangered. The ERC estimates the risk of water-related impacts related to power generation. Reports and maps focus on watershed areas and use U.S. Geological Survey watershed boundary GIS data (stored in the huc_8 table in the database). Each watershed has a Hydrologic Unit Code (HUC) as a unique identifier. The huc8_species_natser table identifies species occurring in each HUC, and the sensitivity table has further information about each species. The ERC report uses a footprint analysis similar to those employed in the previous examples. The query below joins the wps_runs, huc8_poly, huc8_species_natser, and sensitivity tables to list sensitivity information for each species for a particular report run for each species occurring in the HUCs overlapped by the footprint. Some example results are listed in Table 2.

```
sql = "SELECT sens.species,sens.taxa,sens.status"
sql += " FROM sensitivity sens"
sql += " INNER JOIN huc8_species_natser spec"
sql += " ON sens.species = spec.global_cname"
sql += " INNER JOIN huc8_poly poly"
sql += " ON spec.huc8 = poly.huc_8"
sql += " INNER JOIN wps_runs runs"
sql += " ON ST_Intersects(poly.geom,"
sql += " ST_GeomFromText ('+fp_wkt'", 3857))"
sql += " AND runs.pywps_process_id = "
sql += str(procId)
sql += " group by sens.species,sens.taxa,"
sql += "sens.status"
cur.execute(sql)
list = []
```

Footprint Analysis of Land Jurisdictions Stored as Polygon Features

In the Solar Mapper report for Protected Lands, the first section describes the land jurisdictions within a footprint, and a 5-mile area around it, with areas. The sma_code field contains jurisdiction types. The query below uses ST_Intersects to isolate the features overlapping the outer buffer and computes the areas within the buffer and footprint for each jurisdiction that it finds for a particular report run. For the area computations, ST_Intersection is used to remove extents outside of the footprint or buffer, and ST_Transform is used to convert the coordinates to an Albers Equal Area projection before the area computation is performed.

```
table_name = "sma_anl_090914"
sql = "SELECT sma_code,sum(ST_Area(ST_Transform("+
  "ST_Intersection('"+table_name+".geom,"+
  "ST_GeomFromText ('+fp_wkt+', 3857)), 102003)))"+
  "as footprint_area"
sql += " , sum(ST_Area(ST_Transform(ST_Intersection("+
  "table_name+".geom, ST_GeomFromText ('+buffer_wkt+",
  "3857)), 102003))) as affected_area"
sql += " FROM '"+table_name
sql += " JOIN wps_runs ON ST_Intersects('"+table_name+
  ".geom, ST_GeomFromText ('+buffer_wkt+', 3857))"
```


Species	Taxa	Status
California Red-legged Frog	Amphibian	T
California Tiger Salamander - Sonoma County	Amphibian	E
Colusa Grass	Plant	T
Conservancy Fairy Shrimp	Invertebrate	E
Fleshy Owls clover	Plant	T

TABLE 2: Example Ecorisk Calculator Results Listing Threatened and Endangered Species Occurring in a Watershed

```
for row in cur:
    # Collect results in list of lists...
```

Footprint Analysis of Imperiled Species Sensitivity Stored as Raster (Cell-based) Data

Many of the layers used in the mapping tools are stored as raster (cell-based) data rather than vector (coordinate-based) data. The `ST_Clip` method can retrieve raster or vector data and returns the data within the footprint. The `WHERE` clause is important for performance because images in the database are usually stored as many records, each with a tile. `ST_Intersects` restricts the much more processing-intensive `ST_Clip` method to the tiles overlapping the footprint. When the footprint overlaps multiple image tiles, multiple records are returned to the cursor, and results are combined in the loop.

```
list = []
sql = "SELECT (pvc).value as val,sum((pvc).count) "+
      "FROM (SELECT ST_ValueCount(ST_Clip(rast,1, "+
      "ST_GeomFromText('"+fp_wkt"', 3857))) as pvc "+
      "FROM "+layer+" as x "+
      "WHERE ST_Intersects(rast, ST_GeomFromText('"+
      fp_wkt"',3857))) as y "+ "GROUP BY val ORDER BY val"
cur.execute(sql)
for row in cur:
    list.append([row[0],row[1]])
```

Results in JSON format:

```
{
  "Imperiled Species": {
    "header": [
      "Value",
      "Count"
    ],
    "data": [
      [0.0, 21621], [10.0, 1181], [100.0, 484],
      [1000.0, 1610], [10000.0, 42]
    ],
    "metadata": {
      "shortname": "imperiled_species_area",
      "feature_type": "raster"
    }
  }
}
```

Elevation Profile along User-Specified Corridor Centerline Using Elevation Data Stored as Raster Data

The Corridor Report in the EZMT includes elevation profiles along the user-input corridor centerline. In this example, an elevation layer is sampled along a regular interval along the centerline. First, the coordinate of the sample point is generated with `ST_Line_Interpolate_Point`, next, the elevation data are retrieved from the layer with `ST_Value`.

```
d = {}
d['data'] = []
```

```
minval = 999999.0
maxval = -999999.0
interval = 0.1
samplepct = 0.0
i = 0.0
while i <= 1.0:
    sql = "SELECT ST_AsText(ST_Line_Interpolate_Point("
    sql += "line, "+str(i)+") "
    sql += "FROM (SELECT ST_GeomFromText('"+line
    sql += "') as line) As point"
    cur.execute(sql)
    samplepoint = cur.fetchone()[0]

    sql = "SELECT ST_Value(rast,ST_GeomFromText('"
    sql += samplepoint+"',3857)) FROM "+table_name
    sql += " WHERE ST_Intersects(rast,ST_GeomFromText('"
    sql+= samplepoint+"',3857))"
    cur.execute(sql)
    value = cur.fetchone()[0]
    if minval > value:
        minval = value
    if maxval < value:
        maxval = value
    d['data'].append(value)
    i+= interval
d['min'] = minval
d['max'] = maxval
```

Results:

```
"Elevation Profiles": {
  "header": [
    "From Milepost (mi)",
    "To Milepost (mi)",
    "Data"
  ],
  "data": [
    [0.0, 10.0, {
      "header": [ "Values" ],
      "data": {
        "data": [
          137.0, 135.0, 134.0,
          ...
          194.0, 190.0, 188.0
        ],
        "max": 198.0,
        "min": 131.0
      }
    },
    "metadata": {
      "shortname": "dem_us_250m",
      "feature_type": "raster"
    }
  ]
}
```

Footprint Analysis of Raster Population Density Data

In this example, the input data consist of population density values in raster format, and we want to estimate the total population within the footprint. As in the previous example, `ST_Intersects` is used in the `WHERE` clause to limit the tiles processed by the rest of the query, and multiple records will be output if the footprint overlaps multiple tiles. First, image cells overlapped by the footprint are collected and converted to polygons (`ST_DumpAsPolygons`). Next, the polygons are trimmed with the footprint (`ST_Intersection`) to remove portions of cells outside the footprint and are converted to an equal area projection (`ST_Transform`); and then the area is computed. Finally, the total population is computed (`density * area`), prorated by the proportion of the cell within the footprint.

```
sql = "SELECT orig_dens * orig_area * new_area/"
      "orig_area as est_total "+
```



```

"FROM (SELECT val as orig_dens,"+
" (ST_Area(ST_Transform(ST_GeomFromText("+
"ST_AsText(geom),3857),102003))"+
"/1000000.0) As orig_area,(ST_Area("+
"ST_Transform(ST_GeomFromText("+
"ST_AsText(ST_Intersection(geom,"+
"ST_GeomFromText('"+fp_wkt+
"',3857))),3857),102003))/1000000.0) "+
"as new_area "+
"FROM (SELECT (ST_DumpAsPolygons(ST_Clip("+
"rast,1,ST_GeomFromText('"+
fp_wkt+"',3857))))).* "+
"FROM "+table_name+" WHERE ST_Intersects("+
"rast,ST_GeomFromText('"+
fp_wkt+"',3857)) As sample) as x"
cur.execute(sql)
totpop = 0.0
for row in cur:
    totpop += row[0]

```

Computation of Suitability for Wind Turbines Using Raster Data Using GRASS

The suitability models implemented in the EZMT use GRASS software for computations, which are accessed in Python through WPSs. The code below shows the main steps followed when running a suitability model in the EZMT. The models use a set of raster layers as inputs, each representing a siting factor such as wind energy level, land cover, environmental sensitivity, proximity to existing transmission infrastructure, etc. Each input layer is coded with values ranging from 0 (Completely unsuitable) to 100 (Completely suitable), and weights are assigned to each layer representing its relative importance. A composite suitability map is computed using a weighted geometric mean. Figure 4 shows the EZMT model launcher with the default settings for land-based wind turbines with 80-meter hub heights.

Processing in the Python code follows the same steps that would be used in the command-line interface. First, the processing resolution is set using `g.region`. Then, the input layers are processed to normalize the weights to sum to 1.0 (this approach simplifies the model computation). Next, an expression is generated, specifying the formula for the model, and `r.mapcalc` is called to perform the model computation. `r.out.gdal` is used to export the model result from GRASS format to GeoTiff for compatibility with GeoServer, and the projection is set using `gdal_translate` from the Geospatial Data Abstraction Library [GDAL] plugin for GRASS.

```

# Set the processing resolution
WPSProcess.cmd(self, "g.region res=250")

outmap = "run"+str(self.process_run_id)
layers = []
weights = []
# Calculate sum of weights
total = 0.0
for l in model['layers']:
    total = total + model['layers'][l]['weight']

# Create input array of layer names, and
# normalize weights
for l in model['layers']:
    layers.append({
        # The reclass method applies user-specified
        # suitability scores to an input layer
        'name': self.reclass(model, l),
        'weight': model['layers'][l]['weight']/total
    })

geometric_exp = []

```

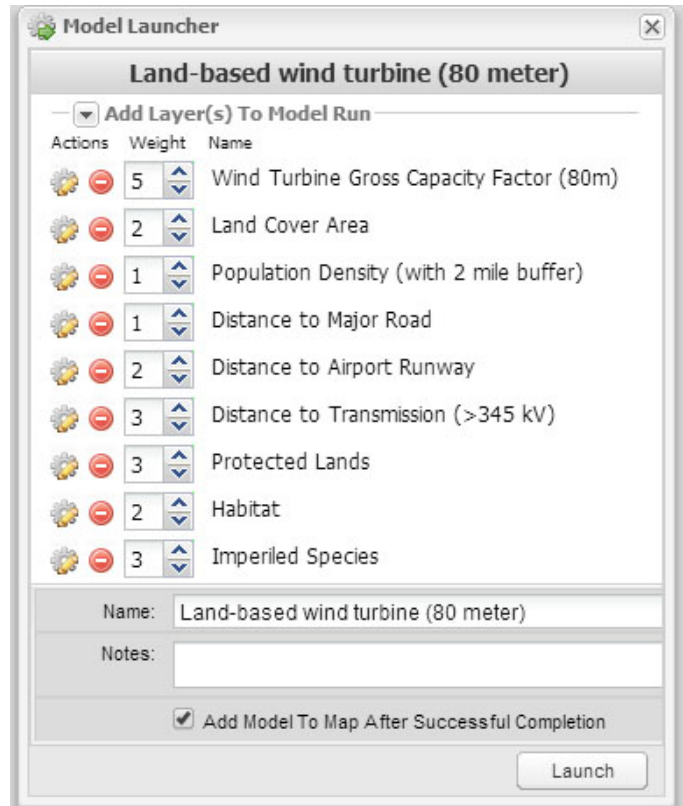


Fig. 4: Land-based Wind Turbine Suitability Model Launcher in the EISPC Energy Zones Mapping Tool

```

total_weight = 0.0
for l in layers:
    total_weight = total_weight + l['weight']
    geometric_exp.append(("pow("+l['name']+","+
str(l['weight'])+")"))
func = "round("+
string.join(geometric_exp, "*")+")"

# Run model using r.mapcalc
WPSProcess.cmd(self, "r.mapcalc "+outmap+
"="+str(func))

user_dir = "/srv/ez/shared/models/users/"+
str(self.user_id)
if not os.path.exists(user_dir):
    os.makedirs(user_dir)

# Export the model result to GeoTIFF format
WPSProcess.cmd(self, "r.out.gdal -c input="+
outmap+" output="+outmap+".tif.raw"+
" type=Byte format=GTiff nodata=255 "+
"createopt='TILED=YES', 'BIGTIFF=IF_SAFER'")

# Set the projection of the GeoTIFF to EPSG:3857
WPSProcess.cmd(self,
"gdal_translate -a_srs EPSG:3857 "+outmap+
".tif.raw "+user_dir+"/"+outmap+".tif")

```

CONCLUSIONS

Python is the de-facto standard scripting language in both the open source and proprietary GIS world. Most, if not all, of the major GIS software systems provide Python libraries for system integration, analysis, and automation, including ArcGIS, GeoPandas [GeoP], geoDjango [geoD], GeoServer, GRASS, PostGIS, pySAL [pySAL], and Shapely [Shp]. Some of these systems, such

as ArcGIS and geoDjango, provide frameworks for web-based mapping applications different from the approach we described in the SOFTWARE ENVIRONMENT section. While it is outside the scope of this paper to discuss the merits of these other approaches, we recommend considering them as alternatives when planning projects.

The examples in this paper include vector and raster data, as well as code for converting projections, creating buffers, retrieving features within a specified area, computing areas and lengths, computing a raster-based model, and exporting raster results in GeoTIFF format. All examples are written in Python and run within the OGC-compliant WPS framework provided by PyWPS.

One of the key points we make is that the Web Mercator projection should not be used for generating buffers or computing lengths or areas because of the distortion inherent in the projection. The examples illustrate how these computations can be performed easily in PostGIS. We chose to use the Albers Equal Area projection, which is commonly used for regional and national maps for the United States. Different projections should be used for more localized areas.

So far our Web-based mapping applications include fairly straightforward analysis and modeling services. However, the same approaches can be used for much more sophisticated applications that tap more deeply into PostGIS and GRASS, or the abundant libraries available in the Python ecosystem. Matplotlib, NetworkX, NumPy, RPy2, and SciPy can each be integrated with Python to provide powerful visualization, networking, mathematics, statistical, scientific, and engineering capabilities.

ACKNOWLEDGMENTS

This work was supported by the U.S. Department of Energy, Office of Electricity Delivery and Energy Reliability; and the U.S. Department of Interior, Bureau of Land Management, through U.S. Department of Energy contract DE-AC02-06CH11357. The submitted manuscript has been created by the University of Chicago as Operator of Argonne National Laboratory ("Argonne") under contract No. DE-AC02-06CH11357 with the U.S. Department of Energy. The U.S. Government retains for itself, and others acting on its behalf, a paid-up, nonexclusive, irrevocable worldwide license in said article to reproduce, prepare derivative works, distribute copies to the public, and perform publicly and display publicly, by or on behalf of the Government.

REFERENCES

- [OpLyr] <http://openlayers.org>
- [PGIS] <http://postgis.net/docs/manual-2.0/reference.html>
- [pySAL] <http://pysal.readthedocs.org/en/v1.7>
- [PyWPS] <http://pywps.wald.intevation.org>
- [RoR] <http://rubyonrails.org>
- [Sen] <http://www.sencha.com/products/extjs>
- [Shp] <http://pypi.python.org/pypi/Shapely>
- [Sol] <http://solarmapper.anl.gov>
- [Sol13] Kuiper, J., Ames, D., Koehler, D., Lee, R., and Quinby, T., "Web-Based Mapping Applications for Solar Energy Project Planning," in *Proceedings of the American Solar Energy Society, Solar 2013 Conference*. Available at http://proceedings.ases.org/wp-content/uploads/2014/02/SOLAR2013_0035_final-paper.pdf.
- [Ub] <http://www.ubuntu.com>
- [VM] <http://www.vmware.com>
- [Arg13] Argonne National Laboratory, *Energy Zones Study: A Comprehensive Web-Based Mapping Tool to Identify and Analyze Clean Energy Zones in the Eastern Interconnection*, ANL/DIS-13/09, September 2013. Available at <https://eispctools.anl.gov/document/21/file>
- [Btsrp] <http://getbootstrap.com>
- [DOI12] U.S. Department of the Interior, Bureau of Land Management, and U.S. Department of Energy, *Final Programmatic Environmental Impact Statement for Solar Energy Development in Six Southwestern States*, FES 12-24, DOE/EIS-0403, July 2012. Available at <http://solareis.anl.gov/documents/fpeis>
- [Erc] <http://bogi.evs.anl.gov/erc/portal>
- [Ezmt] <http://eispctools.anl.gov>
- [GDAL] <http://www.gdal.org>
- [geoD] <http://geodjango.org>
- [GeoP] <http://geopandas.org>
- [Ggl] <http://maps.google.com>
- [GRASS] <http://grass.osgeo.org>
- [Gsrvr] <http://geoserver.org>
- [NGA] http://earth-info.nga.mil/GandG/wgs84/web_mercator/index.html
- [OGP] <http://www.epsg.org>

Scaling Polygon Adjacency Algorithms to Big Data Geospatial Analysis

Jason Laura^{‡*}, Sergio J. Rey[‡]

http://www.youtube.com/watch?v=kNcA-yE_iNI



Abstract—Adjacency and neighbor structures play an essential role in many spatial analytical tasks. The computation of adjacency structures is non-trivial and can form a significant processing bottleneck as the total number of observations increases. We quantify the performance of synthetic and real world binary, first-order, adjacency algorithms and offer a solution that leverages Python's high performance containers. A comparison of this algorithm with a traditional spatial decomposition shows that the former outperforms the latter as a function of the geometric complexity, i.e the number of vertices and edges.

Index Terms—adjacency, spatial analysis, spatial weights

Introduction

Within the context of spatial analysis and spatial econometrics the topology of irregularly shaped and distributed observational units plays an essential role in modeling underlying processes [Anselin1988]. First and higher order spatial adjacency is leveraged across a range of spatial analysis techniques to answer the question - who are my neighbors? In answering this simple question, more complex models can be formulated to leverage the spatial distribution of the process under study. Three example applications are: spatial regionalization, spatial regression models, and tests for global spatial autocorrelation¹.

Spatial regionalization algorithms seek to aggregate n polygon units into r regions ($r < n$) under some set of constraints, e.g., minimization of some attribute variance within a region [Duque2012]. A key constraint shared across spatial regionalization algorithms is that of contiguity as regions are required to be conterminous. One common application of spatial regionalization is political redistricting, where large scale census units, are aggregated into political districts with a contiguity constraint and one or more additional constraints, e.g. equal population. In this context, the generation of a representation of the spatial adjacency can become prohibitively expensive.

At its most basic, spatial regression [Ward2007] seeks to formulate a traditional regression model with an added structure to capture the spatially definable interaction between observations when spatial autocorrelation (or spatial heterogeneity) is present. The addition of a spatial component to the model requires the

generation of some measure of the interaction between neighbors. First and higher order spatial adjacency fulfills that requirement. An example application could be the formulation of a model where the dependent variable is home value and the independent variables are income and crime. Making the assumption that the data is spatially autocorrelated, a spatial regression model can be leveraged.

Moran's I [Anselin1996a] concurrently measures some attribute value and its spatial distribution to identify spatially clustered (positively autocorrelated), random, or dispersed (negatively autocorrelated) data. Therefore, a spatial adjacency structure is an essential input. Measures of global spatial autocorrelation find a large range of applications including the identification of SIDS deaths [Anselin2005] and the identification of cancer clusters [OSullivan2010]. As the total number of observations becomes large, the cost to generate that data structure can become prohibitive.

The computation of a spatial adjacency structure is most frequently a precursor to more complex process models, i.e. a pre-processing step. This processing step occurs dynamically, i.e. the data is not loaded into a spatial database where efficient indexing structures can be pre-generated. Therefore, the computational cost of generating these data structures is often overlooked in the assessment of global algorithmic performance.

Within the spatial analysis domain, ever increasing data sizes due to improved data collection and digitization efforts, render many spatial analytical intractable in a Big Data environment due to unoptimized, serial algorithm implementations [Yang2008]. Therefore, improved serial and distributed algorithms are required to avoid the application of data reduction techniques or model simplification. Binary spatial adjacency is one example of an algorithm that does not scale to large observation counts due to the complexity of the underlying algorithm. For example, a key requirement of Exploratory Spatial Data Analysis (ESDA) [Anselin1996b] is the rapid computation and visualization of some spatially defined measures, e.g. Local Moran's I. Within the Python Spatial Analysis Library (PySAL), the computation of a local indicator of spatial autocorrelation utilizes binary adjacency in computing Local Moran's I as a means to identify the cardinality of each observation. In a small data environment ($n < 3000$) a naive implementation is sufficiently performant, but as the resolution of the observational unit increases (a move from U.S. counties or county equivalents² to census tracts³) compute time increases non-linearly. When combined with the compute cost to perform the primary analytical technique, and potential

* Corresponding author: jlaura@asu.edu

‡ School of Geographical Sciences and Urban Planning, Arizona State University

network transmission costs in a Web based environment, ESDA at medium to large data sizes is infeasible.

Scaling to even larger observation counts where longer run-times are expected, heuristically solved regionalization models, e.g., Max-P-Regions [Duque2012], require that a spatial contiguity constraint be enforced. In large data setting, where a high number of concurrent heuristic searches are to be performed, the computation of adjacency can be a serial processing bottleneck⁴. Improved adjacency metrics are required within this domain for two reasons. First, in a distributed environment with shared resources, reduction of pre-processing directly correlates with time available for analysis. Using heuristic search methods this translates to additional time available to search a solution space and potential identify a maxima. Second, the scale at which regionalization is initiated is an essential decision in research design as underlying attribute data or processes may only manifest at some limited scale range. Therefore, a significant bottleneck in adjacency computation can render the primary analytical task infeasible.

This work targets one processing step in larger analytical workflows with the goal supporting increased data sizes and reducing the total compute time. The application of an improved adjacency algorithm solves one limitation in the application of ESDA to Big Data and reduces the overall pre-processing time required for spatial regionalization problems.

Spatial Weights Object

A spatial weights object or weights matrix, W , is an adjacency matrix that represents potential interaction between each i, j within a given study area of n spatial units. This interaction model yields W a, typically sparse, $n \times n$ adjacency matrix. Within the context of spatial analysis, the interaction between observational units is generally defined as either binary, $w_{i,j} = 0, 1$, depending on whether or not i and j are considered neighbors, or a continuous value reflecting some general distance relationship, e.g. inverse distance weighted, between observations i and j .

In the context of this work, we focus on binary weights where the adjacency criteria requires either a shared vertex (Queen case) or a shared edge (Rook case). Using regular lattice data, Figure (1) illustrates these two adjacency criteria. In the Queen case implementation is in line with expectations, i.e. a single shared vertex is sufficient to assign adjacency. The Rook case, adjacency is more complex and two shared vertices are not sufficient to assert adjacency, i.e. a queen case implementation with a counter for the number of shared vertices. Full geometry edges must be compared as it is feasible that two shared vertices do not indicate a shared edge. For example, a crescent geometry can share two vertices with another geometry but fail to share an edge is another, interceding geometry is present.

PySAL

This work is cited and implemented within the larger PySAL (Python Spatial Analysis Library) project. PySAL is an open-source, pure Python library that provides a broad array of spatial computational methods [Rey2010]. This library has been selected for three reasons. First, PySAL provides data structure, i.e. infrastructure for reading common spatial formats and rendering spatial weights matrices, as a W class instance. This existing functionality facilitated rapid development that could focus on algorithm implementation and testing. Second, PySAL implements two spatial adjacency algorithms that serve as benchmarks and validation tools: (1) spatial decomposition through binning and (2)

r-tree generation and search. Finally, PySAL is a mature, open-source project with a wide user base providing exposure of this implementation to the community for further development and testing.

Algorithms

Problem Definition

The population of an adjacency list, A , or adjacency matrix must identify all polygon geometries which are conterminous. The definition of adjacent is dependent upon the type of adjacency matrix to be generated. Each adjacency algorithm requires a list of polygon geometries, L , composed of sublists of vertices, $L = [p_1, p_2, \dots, p_n]$. Traditionally, the vertices composing each polygon, p_i , are stored in a fixed winding order (clockwise or counter-clockwise) and share a common origin-termination vertex, $p_i = [v_1, v_2, v_3, \dots, v_1]$. This latter constrain facilitates differentiation between a polygon and polyline.

Below we review three adjacency computation algorithms: a naive approach, a binning approach, and an r-tree approach. We then introduce an improved adjacency algorithm using high performance containers.

Naive Approach

The naive approach to compute spatial adjacency requires that each vertex, in the case of rook contiguity, or edge, in the case of queen contiguity, be compared to each other vertex or edge, respectively. This is accomplished by iterating over a list or array of input geometries, popping the first geometry from the list, and then comparing all vertices or edges to all remaining geometries within L . This approach leverages the fact that an adjacency matrix, and by extension an adjacency list is diagonally symmetrical, i.e. the upper right and lower left triangles are identical. This algorithm is $O(\frac{n^2}{2})$ as each input vertex or edge is compared against each remaining, unchecked vertex or edge. A minor modification to this approach allows the algorithm to break once adjacency has been confirmed, thereby avoiding duplicate checks on known neighbors.

Spatial Binning

Binning seeks to leverage the spatial distribution of L to reduce the total number of vertex or edge checks. Binning approaches can be static, whereby the size of each bin is computed a priori and without consideration for the underlying data density or adaptive, whereby the size of each bin is a function of the number of geometries contained within. A quad-tree approach is a classic example of the latter technique. Using a static binning approach as an example, a regular grid or lattice can be overlaid with L and the intersection of all p into a specific grid cell, $g_{i,j}$ computed. Using binning, polygons may span one or more grid cells. Once the global dataset has been decomposed into a number discrete grid cells, all geometries which intersect a given cell are tested for adjacency. This test can be performed by storing either a dictionary (hash) of cell identifiers to member polygon identifiers or a dictionary of geometries identifiers to cell identifiers. The end result is identical, a subset of the global geometries that may be conterminous.

The primary advantage of this approach over the naive algorithm is the reduction in the total number of edge or vertex checks to be performed. Those polygons which do not intersect the same grid cell will never be checked and the spatial distribution of the

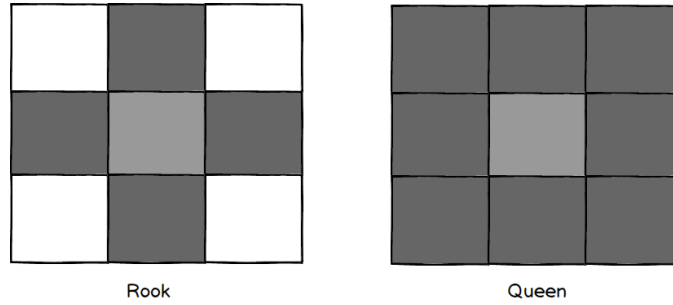


Fig. 1: Rook (shared edge) and Queen (shared vertex) adjacency on a regular 3 by 3 lattice.

data is leveraged. The application of a quad-tree decomposition also accounts for variation density. These advantages are not without cost; the time to compute the initial decomposition can exceed the benefits attained.

Parallel Spatial Binning

One approach to improve the performance of the binning algorithm would be to utilize multiple processing cores (workers). In this implementation binning is performed in serial and then each bin is mapped to an available processing core for processing. Therefore, the expensive $O(\frac{n^2}{2})$ computation can be performed concurrently, up to the number of available processing cores. An implementation of this type requires three processing steps, with only the second step being performed concurrently. First, derive a domain decomposition and assign each geometry to one or more bins⁵. Second, concurrently apply the naive algorithm to all geometries within a bin. This requires that the full geometries be communicated from the mother process to the worker process or that the geometries be stored in a globally accessible shared memory space. Finally, aggregate the results from each worker. Boundary crossing geometries will be processed by more than one worker that does not have knowledge of adjacent bins. Therefore, this step is required to remove redundant adjacencies and generate a single adjacency list.

Like the binning approach, decomposition is a non-trivial compute cost. Additionally, the cost to communicate native python data structures is high in parallel environment. Representation in efficient arrays requires the generation of those arrays, another upfront processing cost.

R-Tree

Like the binning approach, the r-tree seeks to leverage the spatial distribution of the geometries to reduce the total number of $O(\frac{n^2}{2})$ computations that must be performed. An r-tree is composed of multiple levels composed of multiple, ideally balanced nodes, that store aggregated groups of geometry Minimum Bounding Rectangles (MBR). At the most coarse, the MBR associated with each geometry is a leaf in the tree. Each step up a branch aggregates leaves into multi-geometry MBRs or multi-geometry MBRs into larger MBRs. When generating an r-tree two key considerations are the maximum size of each node and the method used to split a node into sub-nodes⁶. An r-tree query uses a depth first search to traverse the tree and identify those MBRs which intersect the provided MBR. For example, assume that geometry A has an MBR of A_{MBR} . An r-tree query begins at level 0 and steps down only those branches which could contain or intersect A_{MBR} .

The primary disadvantage to the r-tree is the cost of generation. In addition to computing the MBR for each input geometry, it is necessary to recursively populate the tree structure using some bulk loading technique. These techniques seek to ensure high query performance, but add significantly to the cost. The implementation tested here utilizes a k-means clustering algorithm to split full nodes and is shown by [Gutman1984] to outperform the standard r-tree and compete with the R*-tree. Even with this improved performance, generation of the data structure is computationally expensive as a function of total compute time. Additionally, scaling to large data sets in memory constrained environments can introduce memory constraints. This is a significantly less common disadvantage, but should nonetheless be addressed.

High Performance Containers and Set Operations

Each of the preceding algorithms, save the naive approach, leverage a decomposition strategy to improve performance. Even with decomposition, the inter-cell or inter-MBR computation is still $O(\frac{n^2}{2})$. Combined with the cost to generate intermediary data structures required to capture the decomposition, it is possible to leverage a higher number of lower cost operations and robust error checking to significantly improve performance. At the heart of our approach is the hashtable (dictionary), that provides average case $O(1)$ lookup by key, the set that provides $O(\text{length}(\text{set}_a) + \text{length}(\text{set}_b))$ set unions and lookup tables that facilitate $O(1)$ list (array) access by element. By minimizing data allocation time and set unions, it is therefore possible to develop an implementation where the majority of computation is, average case, $O(1)$.

In implementation, Algorithm (), the algorithm utilizes a *defaultdict* where the key is the vertex coordinate and the value is a set of those polygon identifiers which contain that vertex (Queen case). Stepping over an input shapefile, line 9, this data structure is iteratively populated. In line 10, we slice the vertex list such that the final vertex is ignored, knowing that it is a duplicate of the first vertex. The inner for loop, line 11, iterates over the list of vertices for a given geometry and adds them to the vertices default dict, line 8. Once this data structure is generated, the algorithm creates another dictionary of sets where the key is a polygon identifier and the value is a set of those polygons which are adjacent. Stepping over the previous dictionary, line 15, the algorithm iterates over the value, a set of neighbors, and populates a new dictionary of sets which are keyed to the polygon identifiers. This yields a dictionary with keys that are polygon ids and values which are sets of neighbors. We define this as a two step algorithm due to the two outer for loops.


```

1 def twostep(fname):
2     shpFileObject = fname
3     if shpFileObject.type != ps.cg.Polygon:
4         return
5     numPoly = len(shpFileObject)
6
7     vertices = collections.defaultdict(set)
8     for i, s in enumerate(shpFileObject):
9         newvertices = s.vertices[:-1]
10        for v in newvertices:
11            vertices[v].add(i)
12
13    w = collections.defaultdict(set)
14    for neighbors in vertices.itervalues():
15        for neighbor in neighbors:
16            w[neighbor] = w[neighbor] | neighbors
17
18    return w

```

Two step algorithm using higher performance containers for the Queen case.

The Rook case is largely identical with the initial vertex dictionary being keyed by shared edges (pairs of vertices) instead of single vertices.

Experiment

Hardware

All tests were performed on a 3.1 Ghz, dual core Intel i3-2100 machine with 4GB of RAM running Ubuntu 64-bit 14.04 LTS. The IPython [Perez2007] notebook environment was used to initiate and analyse all tests. All other non-system processes were terminated.

Experiments

We perform two sets of experiments, one using synthetically generated data and one using U.S. Census data. These tests were performed to quantify the performance of the list based contiguity algorithm as compared to r-tree and binning implementations with the goal of testing three hypothesis. First, we hypothesize that the list based algorithm will be faster than r-tree and binning algorithms across all datasets due to the reduced asymptotic cost. Second, we expect the list based algorithm to scale as a function of the total number of neighbors and the average number of vertices (or edges in the Rook case) per geometry. We anticipate that this scaling remains linear. Third, we hypothesize that the algorithm should not scale significantly worse within the memory domain than either the r-tree or binning approaches due to the reduced number of data structures.

To test these hypotheses we generate both regularly tessellating and randomly distributed synthetic data ranging in size from 1024 geometries to 262,144 geometries⁷. We utilize triangles, squares and hexagons as evenly tessellating geometries with easily controlled vertex count, edge count, and average neighbor cardinality. We also densify the 4096 hexagon lattice to test the impact of increased vertex count as the number of edges remains static. To assess algorithm performance with real world data we utilize U.S. census block group data.

Results

Across all synthetic data tests we report that the r-tree implementation was 7 to 84 times slower than the binning implementation and 22 to 1400 times slower than the list based contiguity measure. Additionally, we see that the r-tree implementation required significant quantities of RAM to store the tree structure. We therefore

illustrate only the binning and list based approach in subsequent figures.

Figure (2)(a - d) illustrate the results of four experiments designed to compare the performance of the list based and binning approaches as a function of total geometry count, total vertex count (and by extension edge count), average neighbor cardinality, and data distribution. Figure (2)(a) illustrates the scaling performance of the list and binning algorithms. The former scales linearly as the total number of polygons is increased and the latter scales quadratically. As anticipated, the Rook contiguity measures require slightly more processing time than the associated Queen contiguity measures. In Figure (2)(b), the algorithm exhibits increased computational cost as a function of geometric complexity, e.g. the number of vertices, number of edges, and mean number of neighbors. This is illustrated by the general trend of compute times with the triangular tessellation requiring the least time and the hexagon tessellation requiring the most. Densification of the 4096 hexagon polygon with between 6 and 300 additional vertices per edge highlights an inversion point, where binning regains dominance over the list based approach, Figure (2)(c). Finally, in Figure (2)(d) the total compute time using randomly distributed polygon datasets are shown. Again, we report quadratic scaling for the existing binning approach and linear scaling for the list based approach.

To test algorithm performance with real world data, we utilize four, increasingly large subsets of the global U.S. census block dataset, Figure (3). We report that neither binning nor our list based solution are dominant in all use cases. We report that, as a function of the total geometry count, it appears that a threshold exists around $n = 32500$ (lower x-axis). Utilizing the upper x-axis, the previous assertion appear erroneous; overall algorithm scaling is a function of the total count, but comparative performance is a function of the geometric complexity with parity existing around $n = 275$ and dominance of the list based method being lost between $275 < n < 575$.

Discussion

Our list based adjacency algorithm significantly outperforms the current r-tree implementation within the PySAL library. We believe that this is a function of the increased overhead required to generate a the tree structure. Across all synthetic data tests, save the vertex densification, we see the list based approach performs well. As anticipated, this method scales with the number of vertices.

Utilizing real world data, the selection of algorithm becomes significantly more challenging as the list based approach does not behave in a linear manner. We suggest that the constant time set operations become large as a function of total compute time. Having gained this insight, we ran additional tests with a read threshold. In this implementation a subset of the input dataset is read, processed, and written to an in-memory W object. This process iterates until the entire dataset is read. Using this method, we see that the list based approach, in the Queen case, can be as performant as the binning approach as a function of the mean number of vertices. Since this information is not available via the binary shapefile header, we suggest that the list based approach may be performant enough across all use cases, i.e. the performance does not significantly degrade at extremely high vertex counts. The list based approach still dominates the binned approach in the Rook case.

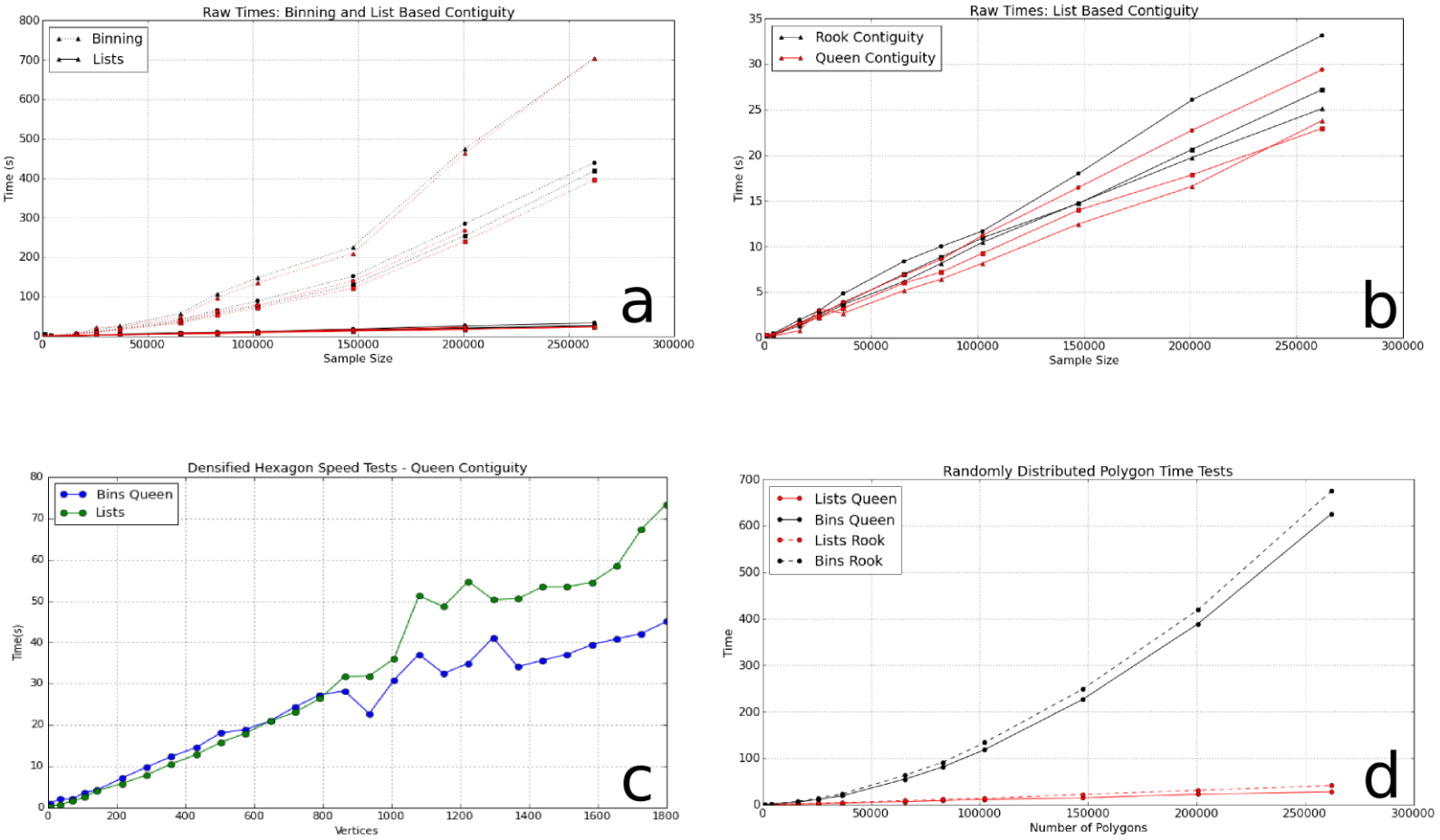


Fig. 2: Spatial binning and list based performance comparison showing: (a) scaling a total synthetic data size increases, (b) list based scaling using synthetic data, (c) scaling performance as the total number of vertices is increased, and (d) randomly distributed data with varying neighbor cardinality and vertex counts.

Utilizing real world data, the binning approach is also able to leverage an essential break function, where known neighbors are no longer checked. This is not, to our knowledge, feasible using the list based approach and two neighbors with n shared vertices must be compared n times. The introduction of a break, if feasible, should continue to improve performance of the list based approach.

Finally, in profiling both the binning and list based approaches, we see that reading the input shapefile requires at least one third of the processing time. Therefore, I/O is the largest current processing bottleneck for which parallelization maybe a solution.

Next Steps

As described above, the r-tree implementation was significantly slower than anticipated. To that end, we intend to profile and potentially optimize the PySAL r-tree implementation with the goal of identifying whether poor performance is a function of the implementation or a product of the necessary overhead required to generate the tree structure.

The improved adjacency algorithm provides multiple avenues for future work. First, we have identified file i/o as the current processing bottleneck and have shown that the algorithm can leverage concurrent streams of geometries. Therefore, parallel i/o and a map reduce style architecture may provide significant performance improvements without major algorithm alterations.

This could be realized in a Hadoop style environment or with a cluster computing environment. Second, we believe that error and accuracy of spatial data products remain an essential research topic and suggest that the integration of a 'fuzzy' checker whereby some tolerance value can be used to determine adjacency is an important algorithm addition. Finally, we will continue integration into PySAL of these methods into more complex spatial analytical methods so that total algorithm processing time is improved, not just the more complex analytical components.

REFERENCES

[Anselin1988] Anselin, L. *Spatial econometrics: Methods and models*, Matrinus Nijhoff, Dordrecht, the Netherlands. 1988.

1. In contrast to local measures which identify local, statistically significant autocorrelation.
2. {32, 64, 128, 160, 192, 256, 288, 320, 384, 448, 512} geometries squared.
3. $n = 3, 144$
4. Clearly this can be overcome, in a distributed environment, using an excess computation strategy, but the increased cost due to algorithm performance still exists.
5. $n = 74, 134$ in the 2010 census
6. Conversely assign each bin to those geometries it contains.
7. While this section describes the function of an r-tree from fine to coarse, they are generated from coarse to fine.

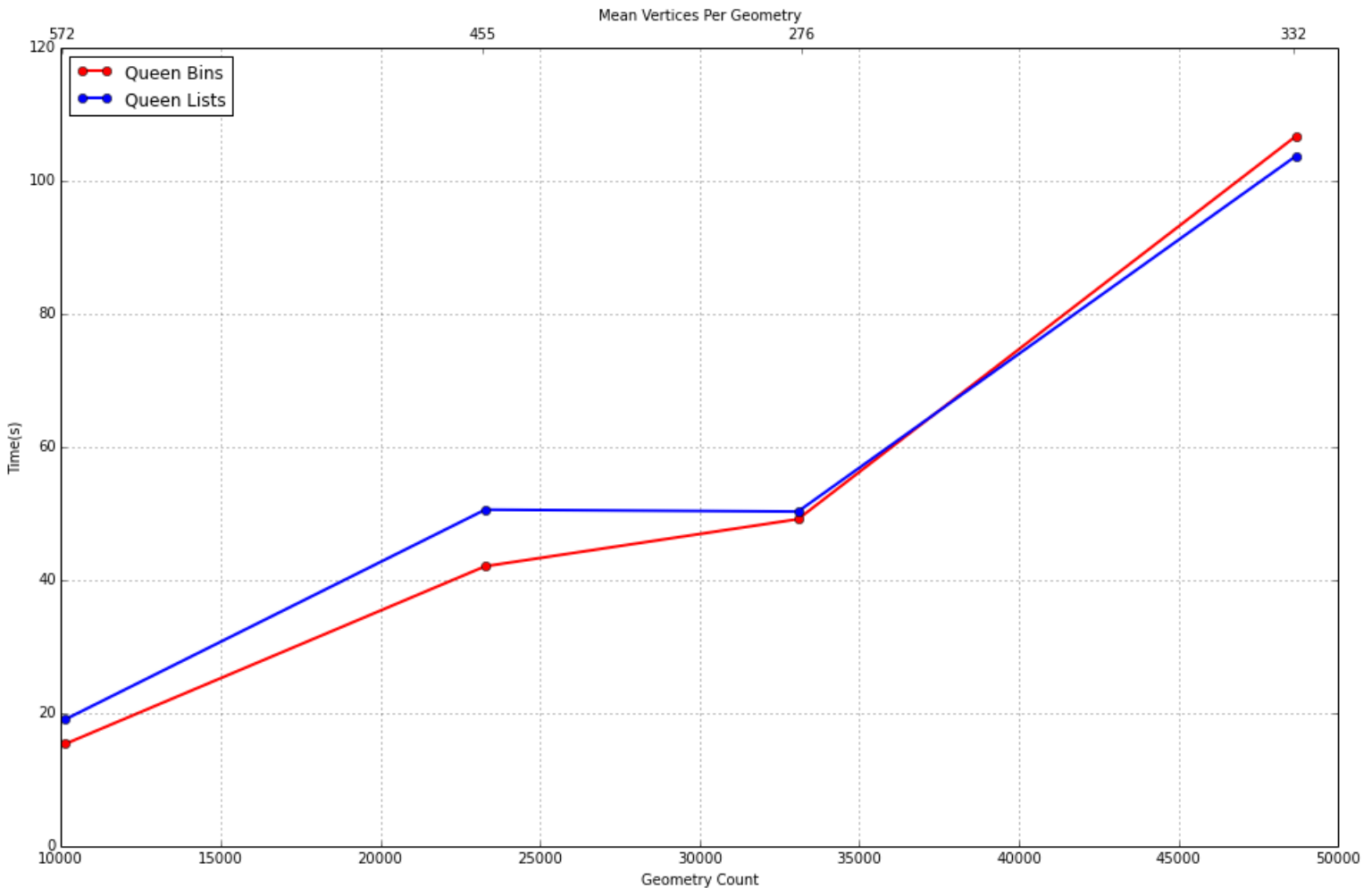


Fig. 3: Spatial binning and list performance for the Queen contiguity case using four subsets of census blocks in the Western United states with varying spatial densities, geometry counts, and mean vertex counts. Plot origin is based upon the number of geometries (lower x-axis).

- [Anselin1996a] Anselin, L. and Smirnov, O. *Efficient algorithms for constructing proper higher order spatial lag operators*, Journal of Regional Science, vol. 36, no. 1, pp.67 – 89, 1996.
- [Anselin1996b] Anselin, L., Yong, W., and Syabri, I. *Web-based analytical tools for the exploration of spatial data*, Journal of Geographical Systems, vol. 6, no. 2, pp. 197-218, 2004.
- [Anselin2005] Anselin, L. *Exploring Spatial Data with GeoDa: A Workbook*, Center for Spatially Integrated Social Science, University of Illinois, Urbana-Champaign, 2005.
- [Duque2012] Duque, J. C., Anselin, L., and Rey, S. J. *The Max-P-Regions Problem*, Journal of Regional Science, 52(3):pp. 397–419, 2012.
- [Gutman1984] Gutman1984, A. *R-Trees: A dynamic index structure for spatial searching*, Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data, 1984.
- [OSullivan2010] O’Sullivan, D. and Unwin, D.J. *Area Objects and Spatial Autocorrelation*, Geographic Information Analysis, Wiley, Ch. 7, 2010.
- [Perez2007] Pérez, F. and Granger, Brian E., *IPython: A System for Interactive Scientific Computing*, Computing in Science and Engineering, vol. 9, no. 3, pp. 21-29, 2007. URL: <http://ipython.org>
- [Rey2010] Rey, S. J. and Anselin, L. *PySAL: A Python library of spatial analytical methods*, In Fischer, M.M ; Getis, A., editor, Handbook of Applied Spatial Analysis, pp. 175–193. Springer, 2010.
- [Ward2007] Ward, M. D. and Gleditsch, K. S. *An Introduction to spatial regression models in the social sciences**, <https://web.duke.edu/methods/pdfs/SRMbook.pdf>, 2007, Retrieved June 12, 2014.
- [Yang2008] Yang, C., Li, W., Xie, J., and Zhou, B. *Distributed geospatial information processing: sharing distributed geospatial resources to support Digital Earth*, International Journal of Digital Earth, pp. 259-278, 2008.

Campaign for IT literacy through FOSS and Spoken Tutorials

Kannan M. Moudgalya^{‡*}

Abstract—This article explains an approach to promote Information Technology (IT) literacy in India, which has evolved into a pyramid structure. We begin this article by explaining the design decisions, such as the use of FOSS and being a friendly interface between beginners and experts, in undertaking this activity.

A Spoken Tutorial is a ten minute audio video tutorial on open source software, created to provide training on important IT topics. Spoken Tutorials are made suitable for self learning, through a novice check of the underlying script. The spoken part of these tutorials is dubbed in all Indian languages, to help children who are weak in English, while retaining employment potential. The effectiveness of conducting workshops using spoken tutorials is explained. A total of 400,000 students have been trained in the past three years through one or more Spoken Tutorial based Education and Learning through Free FOSS study (SELF) workshops.

Students who undergo SELF workshops can create textbook companions, which contain code for the solved problems of given textbooks using a particular software. A Python Textbook Companion is a collection of Python code for a given textbook. These companions and the associated textbook, together, form a low cost document for Python in general, and the textbook, in particular. We have completed approximately 80 Python Textbook Companions and another 80 are in progress. From textbook companions, the students can progress to lab migration activity, the objective of which is to migrate labs based on proprietary software to FOSS. Interested students are trained to use FOSS systems in their projects and to contribute to the development of new FOSS systems. Using this approach and Python as a glue language, we have developed the following new FOSS systems: 1. OScad, an electronic design automation tool, and a FOSS alternative to commercial software, such as ORCAD. 2. Sandhi, a block diagram based data acquisition for real time control, and a FOSS alternative to commercial software, such as LabVIEW.

The pyramid structure explained in this work helps the beginners to become IT literate. Our design solutions are especially suitable to poor and marginalised sections of the society, which is at the bottom of the social pyramid. Our efforts to create and promote the world's lowest cost computing system Aakash is briefly addressed in this work.

Index Terms—Python, spoken tutorials, FOSSEE

Introduction

This article explains the approach we have taken to promote IT literacy in India. While India has the capability to create software to help improve the lives of people around the world, its citizens do not have the capability to absorb it. One main reason is that India does not have a good infrastructure. In addition, the economic

* Corresponding author: kannan@iitb.ac.in

‡ Dept. of Chemical Engineering, Education Technology, and System and Control Groups, IIT Bombay, India

Copyright © 2014 Kannan M. Moudgalya. This is an open-access article distributed under the terms of the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original author and source are credited.

conditions do not allow our citizens in general, and students in particular, to buy expensive hardware and software. Lack of good quality educational institutions and good teachers compound the problem.

This paper begins with the design decisions we have taken to achieve our objective while keeping in mind the problems mentioned above. Some of the decisions have been arrived at by trial and error, while some decisions, such as the use of FOSS, have been a cornerstone of our efforts. The next section explains a procedure we use to train a large number of students through FOSS and spoken tutorials. The textbook companion section explains how we create low cost documentation for FOSS through textbook companions. We follow this with a section on lab migration that helps shift lab courses based on proprietary software to FOSS. After this, we explain how these activities lead to a pyramid structure. Finally, we discuss the relevance of the low cost computing device Aakash, and conclude. We emphasize the role of Python in all of the above.

Approach and Design Decisions

We started in 2009 the effort explained in this article, to promote FOSS, with the Indian Government support and funding [nm09]. Although there are many good FOSS systems, they are difficult to use because of lack of documentation and support. If we could bridge this gap, FOSS would become accessible to our students, who do not have access to good software otherwise, as many of them are proprietary in nature. We also did not want to promote commercial software with tax payer's money. Our funding agency [nm09] had also made a policy decision to exclusively use FOSS.

It does not mean that we wanted to erect a wall between us and those who use commercial software. We do work with students who only know how to use the MS Windows operating system, for example. We do accept tutorials that have been dubbed with Windows Movie Maker. We believe that by creating a good ecosystem, we can shift proprietary software users to equivalent FOSS systems. We have always wished to be an inclusive project.

We have been using simple technologies to create our instructional material, as these allow school going students also to participate in the creation effort. For example, we have eschewed the use of instructional methodologies that help embed interaction in our tutorials. In contrast, open source screen recording software, such as RecordMyDesktop [rmd], has limited features. Nevertheless, we have made the learning effective by developing strict processes in creation [guidelines] and use of Spoken Tutorials, such as, the side-by-side method (see Fig. 1) and SELF workshops, explained in the next section.

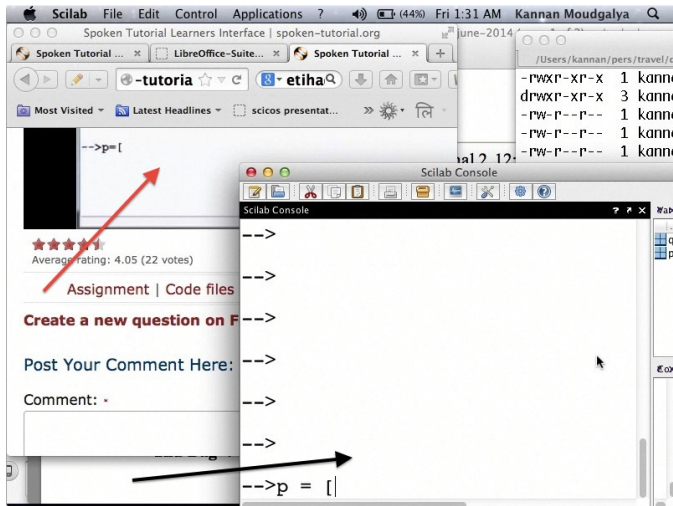


Fig. 1: A side by side arrangement for effective use of Spoken Tutorials. The red arrow points to the video tutorial and the black arrow points to the software studied, in this case, Scilab.

It is our policy that even “ordinary” people can participate in our project, using whatever equipment they have, and get recognised and also paid an honorarium. If an instructional tutorial is below professional quality, but reasonably good, we accept it. At the beginning of the project, we found that if we insisted on professional quality, it would necessitate the use of pin drop silence quality studios, established at a cost of millions of dollars. This would make the cost of spoken tutorials prohibitively expensive. Moreover, our project would become an elite one, filtering most “ordinary” people from participating in it.

We also wanted our methodology to be beginner friendly. When we started this work in 2009, most *Forums* that supported FOSS were quite unfriendly to the questions posed by beginners. For example, standard answers were,

- First learn how to post in the forum
- It is already answered, check the archives first
- Read the instruction manual first before asking a question

The reasons for the response given by the experts are understandable: there are only a few experts to answer the questions of large numbers of beginners, many of which may be repetitive. If the experts spend all their time in answering often repetitive questions, when will they have time to do their own work, develop the FOSS, remove bugs, write documents, etc., seem to be valid questions. Nevertheless, we can confidently say that only a small fraction of FOSS exploring/loving beginners stay with FOSS because of the above mentioned scoldings and the lack of support structure. The system we have built is beginner friendly, with enthusiastic participation from the experts.

When we started the project in 2009, we decided to make our project a friendly interface between beginners and experts. One way to do this was through a *Novice Check* of the script, before creating a video tutorial. Unless a script is approved by a novice, it is not accepted for recording.

We illustrate the novice check with a bash script that the author reviewed as a beginner. The script asked the learner to download a bash file to the current directory and to type the name of the file on the console to execute it. On following the above instruction, the following error message appeared: Command not found. The

FOSS category	No. of Workshops	No. of Students
C and C++	1,840	84,728
Linux	1,819	80,882
PHP and MySQL	997	44,414
Scilab	1,026	41,306
Java	672	31,795
LaTeX	771	30,807
LibreOffice (all components)	776	26,364
Python	419	18,863
Total	8,320	359,159

TABLE 1: Total number of workshops conducted and the students trained in the past three years. The methodology is explained in the next section.

script writer forgot to state that there should be a `./` (dot-slash) before the file name, as the current directory is not in the path of beginner. After correcting this mistake, the same error message appeared. The reason for this is that this file is not executable. The script writer missed the following statement: the downloaded file should be made executable by the `chmod` command. These corrections were incorporated into the script before recording it.

Thus, a spoken tutorial is recorded only after the script, created by experts, is validated by beginners. After recording them, we run pilot workshops with the spoken tutorials. If there are minor difficulties, we mention the corrections in an instruction sheet. If there are major difficulties, the tutorials are re-created.

Although the details to be addressed in our tutorials seem to be excessive, the benefits are enormous. In Table 1, we give the total number of workshops that we have conducted and the number of students trained. The methodology developed by us to achieve such large numbers is explained in the next section.

An expert who knows that their tutorial will be watched 10,000 times will not mind spending a lot of effort to create outstanding instructional material. Insistence of passing through a novice check makes beginners important and also make them feel important. From the expert’s point of view, once it is created, all beginners can be directed to see the tutorial. Finally, as we discuss next, the novice check and pilot workshops make our tutorials suitable for self learning, which in turn has resulted in large scale training, as demonstrated in Table 1.

The fact that a large number of people have undergone our LibreOffice workshops demonstrates that we are reaching out to the clerical staff and those who are at the doorsteps of IT literacy, and hence are at the bottom of the pyramid.

Our efforts to reach out to beginners has resulted in a pyramid structure: once the beginners are trained in a FOSS, they are encouraged to create textbook companions, to be explained below. Motivated students are then invited to participate in migrating lab courses to FOSS, and to use FOSS to create new software systems. Thus, bringing a large number of developers to our fold has the beneficial effect of producing a large number of FOSS developers as well. We begin with our training effort.

Spoken Tutorial

A Spoken Tutorial is an audio - video instructional material created for self learning through the Screencast technology. When this project started in 2009, the main objective was to create documentation for FOSS, so that it is accessible to everyone. A

detailed set of objectives and the method followed to achieve them are summarised in [kmm14].

We will begin with the reasons for calling this instructional material as a Spoken Tutorial. When this work started, there were a large number of *silent* Screencast tutorials on the Internet. To distinguish ours from these, we used the word *spoken*. This word is even more important, as we dub the spoken part into all Indian languages. As we do not capture the face of the person creating the tutorials, it is strictly not a video tutorial. Owing to the fact that one can use Spoken Tutorial to learn a topic, we call it a tutorial.

Spoken Tutorials have been released under a Creative Commons license and are freely downloadable from [Spoken]. There are about 500 original spoken tutorials in English and more than 2,000 dubbed tutorials in various Indian languages.

The Python Team created a set of 14 Spoken Tutorials on Python at the beginning. On using these tutorials, it was found that the pace of some tutorials was fast and that some topics were left out. A fresh set of 37 Spoken Tutorials have been created since then. These have also been dubbed into a few Indian languages.

At present, we have the following Python Spoken Tutorials at the basic level: 1) Getting started with IPython. 2) Using the plot command interactively. 3) Embellishing a plot. 4) Saving plots. 5) Multiple plots. 6) Additional features of IPython. 7) Loading data from files. 8) Plotting the data. 9) Other types of plots - this helps create scatter plot, pie and bar charts, for example. 10) Getting started with sage notebook. 11) Getting started with symbolics. 12) Using Sage. 13) Using sage to teach.

At the intermediate level, we have the following tutorials: 1) Getting started with lists. 2) Getting started with for. 3) Getting started with strings. 4) Getting started with files. 5) Parsing data. 6) Statistics. 7) Getting started with arrays. 8) Accessing parts of arrays. 9) Matrices. 10) Least square fit. 11) Basic data types and operators. 12) I O. 13) Conditionals. 14) Loops. 15) Manipulating lists. 16) Manipulating strings. 17) Getting started with tuples. 18) Dictionaries. 19) Sets.

At the advanced level, we have the following tutorials: 1) Getting started with functions. 2) Advanced features of functions. 3) Using Python modules. 4) Writing Python scripts. 5) Testing and debugging.

Spoken tutorials are created for self learning. The side-by-side method, a term defined in [kmm14] and illustrated in Fig. 1 is recommended for the effective use of spoken tutorials. This is a typical screen of the student running the tutorial. The learner is supposed to reproduce all the steps demonstrated in the tutorial. To achieve this, all supplementary material required for a tutorial are provided. We illustrate this with the Python Spoken Tutorial, loading data from files. In Fig. 2, in the command line, `cat` of the file `primes.txt` is demonstrated. By clicking the Code files link, shown with a red arrow, one may download the required files. In the figure, we have shown the window that pops up when the Code files link is clicked. This popped up window asserts the availability of the file `prime.txt` and also other files that are required for this tutorial. By clicking the link Video, also at the second last line of this figure, one can download the tutorial for offline use.

As these are created for self learning, and are freely downloadable, one should be able to learn from spoken tutorials directly from the website [Spoken]. Nevertheless, there are many reasons why we have been conducting organised workshops [kmm14] using spoken tutorials. As these are created for self learning, a domain expert is not required to explain the use of spoken tutorials

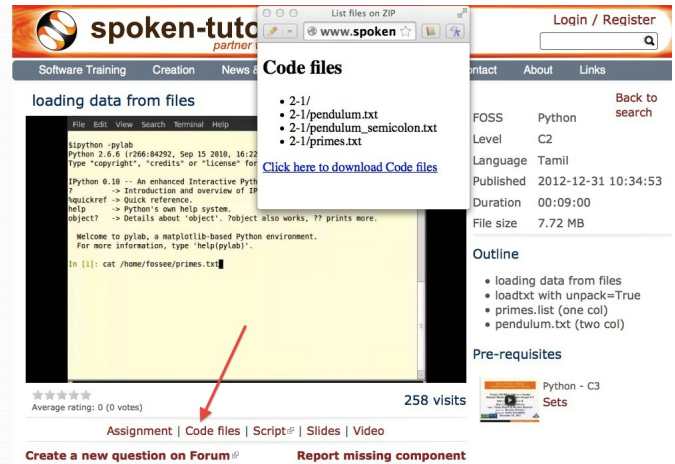


Fig. 2: Resources available for a spoken tutorial explained with an example. The file used in the tutorial is available through the Code files link, indicated by an arrow. On clicking this link, the available code files are displayed in a new window.

- a volunteer can organise these workshops. Based on trial and error, we have decided that our workshops should be of two hour duration and should be conducted as SELF workshops, as mentioned previously. Although these workshops are of only two hour duration, quite a bit can be learnt in a two hour workshop. For example, by no other method can a beginner learn LaTeX topics, such as compilation, letter writing, report writing, mathematical typesetting and introduction to beamer, in a two hour workshop [kmm11-TUGboat]. Although no domain experts may be available during these workshops, one may get one's questions answered through a specifically designed forum [forums].

Most students in India do not have access to good bandwidth and hence cannot access our web page. As a result, we need to provide the tutorials for offline use. In the previous paragraph, we have explained how to download a single video. To be consistent with our ethos, we have implemented a tool that allows the creation of an image consisting of many tutorials and downloading it for offline use. On choosing at [Spoken], Software Training > Download Tutorials > Create your own disk image, one reaches the page shown in Fig. 3. Through this shopping cart like facility, we can create an image consisting of different FOSS families of spoken tutorials, in languages of one's choice. In this figure, one can see that the Python spoken tutorials in English and Tamil have been selected and these will take up about 680 MB. One may add many more FOSS categories, in one or more languages to the Selected Items list. Once all required tutorials are selected, one may click the Submit button. The image consisting of all the tutorials will be download as a zip file. On unzipping this file and opening the index.html file contained therein in a web browser, such as Firefox, all the selected videos can be played from the local drive. This zip file can be copied to all computer systems that are meant to be used in a workshop.

The Spoken Tutorial Team helps conduct SELF workshops [events-team]. The workshops are offered on about 20 topics, such as Python, Scilab, C, C++, Java, LibreOffice, LaTeX, PHP, Octave and GNU/Linux. Organisers of SELF workshops at different institutions download the required spoken tutorials using the facility explained through Fig. 3, install the software to learn and ensure that the computer system, audio/video player and the headphone are in working condition. These organised workshops create a

Fig. 3: The automatic CD content creation facility, available through [Spoken], by clicking Software Training > Download Tutorials > Create your own disk image. See that English and Tamil versions of Python tutorials are selected, with a size estimate of about 680 MB.

conductive ecosystem to learn through spoken tutorials.

As two hours may not be sufficient, one may not learn all the tutorials during a two hour workshop. After the workshop, the students are encouraged to download the tutorials and to practise by themselves at their home or office. The learners can post their difficulties, if any, on the Spoken Tutorial Forum [forums] based on the time line of a spoken tutorial. This special forum helps even beginners to locate previous discussions relating to spoken tutorials. An online exam is conducted a few weeks after the workshop and the participants who pass the exam are provided with certificates.

It is possible to get details of SELF workshops conducted by our team. In [python-ws-info], one can see summary details of the Python workshops that have taken place in the state of Gujarat. One can reach this information on [Spoken] by clicking the map of India, choosing Gujarat and sorting the result by FOSS. A screenshot is given in Fig. 4. In this figure, we have shown a red rectangle around a particular workshop that took place in Surat on 12 July 2013. By clicking the lens symbol, one can see the details of where the workshop took place, who conducted this workshop and so on. When the number of students who attended this workshop is shown in red (in this case, it is 51), it means that some of them have given their feedback. By clicking the number in red, one may locate the feedback given by students. A typical feedback is shown in Fig. 5.

We present some statistics of the people who have undergone Python SELF workshops. The number of SELF workshops conducted until now is 417, training close to 19,000 students, with 9,300 of them being females. It is interesting because it is believed that generally females do not take up programming in large numbers. Some of the reasons for this could be that they also find our methodology appealing, they are equally interested in employment, etc. Python SELF workshops have taken place in 23 states of India. Year wise break up of workshops is given in Table 2.

It should be pointed out that less than one half of the year is over in 2014.

The Python SELF workshops are effective. We have the following testimonials:

Through this workshop one can easily understand the basics of Python, which in turn can develop an

Institution	City	FOSS	Workshop_Date	No. of students
Chandubhai S. Patel Institute of Technology, Charotar University of Science And Technology	Mahuva	Python	2012-02-01	33
Rutuja Creation	Ahmedabad	Python	2011-11-13	30
Chandubhai S. Patel Institute of Technology, Charotar University of Science And Technology	Mahuva	Python	2012-02-01	27
Chandubhai S. Patel Institute of Technology, Charotar University of Science And Technology	Mahuva	Python	2012-02-01	30
C U Shah Science College	Ahmedabad	Python	2011-12-12	17
Parul Institute Of Engineering & Technology-MCA	Vadodara	Python	2014-04-05	90
Chandubhai S. Patel Institute of Technology, Charotar University of Science And Technology	Mahuva	Python	2012-02-01	31
Babaria Institute of Technology	Vadodara	Python	2012-09-12	30
Babaria Institute of Technology	Vadodara	Python	2012-09-11	30
Shree Swami Atmanand Saraswati Institute of Technology	Surat	Python	2013-08-03	44
Shree Swami Atmanand Saraswati Institute of Technology	Surat	Python	2013-07-12	51

Fig. 4: Summary of Python workshops, obtained by clicking the India map in [Spoken], choosing Gujarat and then sorting by FOSS.

Fig. 5: Feedback given by a student of Shree Swami Atmanand Saraswati Institute of Technology, Surat, Gujarat.

Year	No. of workshops	No. of students
2011	21	945
2012	144	6,562
2013	116	4,857
2014	138	6,499
Total	419	18,863

TABLE 2: Python SELF workshops, yearly statistics

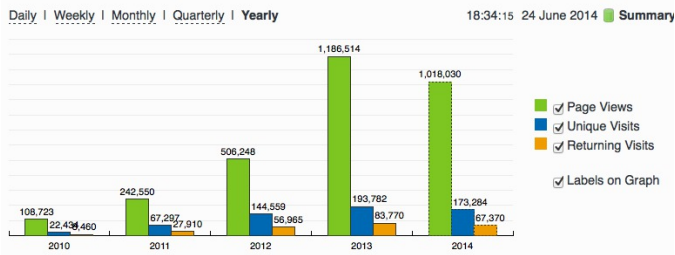


Fig. 6: Number of page views on [Spoken], since the beginning of this website. As there are many offline viewers in India, the effective number of page views may be considered to be at least twice these numbers.

interest in one's mind to learn more about Python. Thank you very much for this wonderful workshop.

—Brindersingh - Guru Nanak Institute of Technology,
West Bengal

Got the initiative of how to work on Python that makes the programming comparative easy. Apart from this, graphical representation of mathematical formulation is quite good.

—Abhishek Bhargava - Institute of Engineering &
Technology, Alwar

Our website [Spoken] is becoming popular. Fig. 6 gives details of page views on our website. One can see that the number of page views are doubling every year. The number of people who benefit from our work is much higher than the numbers indicated in this figure. This is because, there are a lot of students in India who access our material through offline mechanisms, as explained earlier. For example, even though more than 80,000 students have undergone SELF workshops on Linux (Table 1), the largest number of server-recorded page views for any Linux video is only about 2,500. It seems that the equivalent number of page views on our page is at least ten times the number indicated in Fig. 6.

A student who is trained through a SELF workshop is ready to contribute to the community. A textbook companion is the easiest way for them to contribute and in the process also get rewarded. This is explained in the next section.

Textbook Companion

One of the major shortcomings of FOSS tools is the lack of documentation. Proprietary software creators can deploy a lot of money and other resources to develop good documentation. We explain now how we have addressed this important issue through Textbook Companions.

We wanted to create documents for FOSS using India's abundantly available work force, namely, students. Unfortunately, creating a document requires a lot of experience in teaching. Students are good in writing programs, not documents. We explored the possibility of addressing this by solving the inverse problem: ask the students to write programs for existing documents. Textbooks can be considered as good documents. After doing a pilot with six students from different parts of India in the summer of 2010, we came up with the formula of one student, one month, one textbook companion.

Textbook companion (TBC) activity creates code for solved examples of standard textbooks using FOSS. These are created

by students and the faculty of colleges from different parts of India. Students who create these books are given an honorarium of Rs. 10,000 for each companion. We were initially giving Rs. 5,000 honorarium to the teachers of these students for review and quality assurance. This has not worked well, as the teachers are generally not as knowledgeable and not as serious as the student who created the TBC. We have now shifted the review work to a general pool of experts, who are often students.

If anyone wants to understand what a program does, all that they have to do is to go through the corresponding example in the associated textbook. If TBCs are available for all textbooks used in educational programmes, students and teachers would not need proprietary software, at least for classroom use.

This programme is so flexible that almost anyone can contribute to the Python Textbook Companion (PTC) activity: from students to employees, teachers and freelancers. They can choose a textbook of their choice from engineering, science or social sciences, the only requirement being that Python is suitable for solving example problems. Upon successful completion of a PTC, the participant is awarded with a certificate and a handsome honorarium. PTCs are presented in the form of IPython Notebooks.

The PTC interface [PTC] displays all the completed books together with a screen-shot of code snippets, so that the user can easily download the PTC of their interest. The interface also allows the users to view all the codes of a chapter as an IPython notebook, which makes learning Python easy.

We use the following process to develop a PTC:

- 1) A student uploads Python code for the examples of one of the chapters of a chosen textbook. They should ensure that this book is not already completed nor under progress. They should also propose two other textbooks for PTC creation, in case the one they selected is already allocated to someone else.
- 2) Based on the Python code received for one chapter, our reviewers decide whether the student knows sufficient Python to complete the PTC. In case the selected textbook is already allocated to someone else, one of the other two chosen books is assigned. The student is given a time period of three months to complete the PTC.
- 3) The student has to upload the Python code in a specified format, on our portal.
- 4) Our reviewers check the correctness of the submitted code. They check whether the answers given by the code agree with those given in the textbooks.
- 5) Students who get all the code correct during the first review itself get a bonus, in addition to the honorarium mentioned above. Those who increase the work of reviewers by submitting wrong code are penalised and their honorarium gets reduced.

We currently have PTCs in the following categories: Fluid Mechanics, Chemical Engineering, Thermodynamics, Mechanical Engineering, Signal Processing, Digital Communications, Electrical Technology, Mathematics & Pure Science, Analog Electronics, Computer Programming and others. Currently, there are 80 completed PTCs and 80 are in progress. PTCs so created are available for free download at [PTC].

The creators of PTC learn Python in a practical and effective way. One may see below testimonials from a few of the participants:

I experienced that even an inexperienced person can do coding/programming. I gradually got to advance my skills in Python as I approached further in it. I got the IIT-B certificate, plus I got paid a handsome amount of cheque after completion which was good enough for me at then. -- Amitesh Kumar

I learnt Python from Spoken-Tutorials available on the website. The Python TBC team also helped me a lot in starting my internship. Till now, I have completed 3 TBCs and now, I know pretty much about python. I plan to take this project forward and Python is really helping me shine my resume. -- Deepak Shakya

This internship provided me a perfect platform and environment to learn Python. It helped me to incorporate both my ideas and practical work skills to the best. Especially, those concepts of C which are not present in Python gave me an awesome experience. Moreover, experience gained from it will make me capable of facing and overcoming the upcoming challenges under its applications. -- Ramgopal Pandey

We would like to point out some of the processes we have followed in the creation of PTC. Initially we tried to use the Sprint route to create PTCs. This involved a few people jointly coding all the problems, including unsolved problems, of a given book in one sitting. Solving unsolved problems made the task difficult. A book could not be completed in one session and those who coded for a part of the textbook often did not follow up to complete the work. There was also no ownership of the activity as many people were involved in one book. In contrast, the Scilab group used the approach explained previously and found it to be more effective, and more productive: there are 377 completed Scilab TBC and 266 are in progress. As a result, the Python group also changed the strategy for the creation of PTCs and this has yielded good results, as explained above. We are also in the process of contacting all who created Scilab TBC urging them to take up the PTC work.

The FOSSEE Team at IIT Bombay [FOSSEE] supports the following well known FOSS systems: Python, Scilab, OpenFOAM, COIN-OR. It also supports the following FOSS systems developed at IIT Bombay: OScad (a locally developed for Electronic Design Automation and an alternative to OrCAD), Sandhi (an alternative to LabVIEW) and OpenFormal. We are in the process of creating TBCs for all of these systems.

Lab Migration

Students who successfully complete textbook companions, discussed in the previous section, are ready to help their colleges participate in lab migration, to be explained now.

Most of the academic programmes in India have laboratory courses that expect the students to carry out about ten experiments in a semester, in as many lab sessions, each lasting about three hours. Providing FOSS code through textbook companions does not necessarily enforce its use. On the other hand, if a FOSS system were to be used in a lab course, because of its compulsory nature, the use of FOSS system gets into the main stream. Similarly, the use of proprietary software in lab courses perpetuates its use. So long as a proprietary software is used in a lab course, any number of FOSS textbook companions will not wean the students away from the former.

Commercial software	FOSS equivalent
Matlab	Scilab
ORCAD	Oscad
Fluent	OpenFOAM
AnyLogic, Arena Witness, ExtendSim Quest, FlexSIM	SimPy
LabVIEW	Sandhi

TABLE 3: Migration of commercial software based labs to FOSS based labs

The FOSSEE team helps migrate commercial software based labs to FOSS. Once a faculty member in a college wants to migrate a lab to FOSS, we ask them or others in our network to come up with the required code in an equivalent FOSS and pay an honorarium. This code is made available to the public. Our team carries out lab migration given in Table 3. The most successful of them is Matlab to Scilab lab migration [LM]. We have migrated about 25 labs from Matlab to Scilab and about 15 more are in progress. On other FOSS families, we have migrated only a few labs, but the interest is growing. Although its progress is slower than that of TBC, lab migration can have a profound and lasting impact in promoting FOSS.

There is an important difference between a TBC and lab migration. The former is for a standard textbook and its utility is of general value: it may be of use to many students at more than one institution. A TBC is considered useful whether it is used or not in any one particular college. In contrast, the problem statements of a lab could be specific to a particular institution. Because of this, if the institution that participates in lab migration does not use the FOSS code it creates, the effort may be wasted. We insist that lab migration should not be just on paper, but be put in practice. Naturally, the progress in lab migration is slower compared to the TBC effort.

Completing the Pyramid Structure

In this section, we explain how our FOSS efforts help result in a pyramid structure of trained students. We started with SELF workshop based training, progressed to TBC and then reached lab migration, with each stage having increased complexity, as explained in the previous sections. In this section, we explain how a few other higher level activities that we have undertaken help result in a pyramid structure.

The next complicated task we have recently undertaken is to help our students do full projects using the FOSS that we support. Here is a feedback from a student who completed his Master's thesis using OScad:

With intensive cooperation and guidance on OScad EDA tool, from all of You, I have completed the project on "Design and Performance Analysis of OTA based SNR Meter" successfully and also submitted the project report today. Sincere thanks to all of You. OScad is really user friendly and also highly accurate which is the main reason for completion of the project so smoothly.

We at Mangalore Institute of Technology and Engineering have decided to use OScad for two of the labs "Linear Integrated Circuits and Analog communication"

and “Power Electronics” labs. Your support is very much needed mainly for power electronics lab. Hope you will provide it. Thanks a lot. -- Harish Bhat

The next task is to help improve the FOSS itself or to use the FOSS to create new software. Typically, existing FOSS tools are used to create new FOSS systems. Python turns out to be an excellent glue language. We have used Python extensively in the creation of Oscad [oscad-book], [oscad-lj]. We are using Python extensively, once again, in the creation of Sandhi, a FOSS alternative to LabVIEW. Sandhi is yet to be released to the public. We have been using Python also to create online evaluation software to administer post SELF workshop tests.

The next level in this progression is possibly entrepreneurship. It is next level, because, an entrepreneurship is a lot more difficult compared to being a programmer. We also hope that the entrepreneurs who would come out of our work would be good in programming at the minimum. We are exploring the feasibility of grooming potential entrepreneurs from the students whom we train. At present we train about 200,000 students a year through SELF workshops. We expect about 1% of them to be good, possibly as a result of our FOSS promotion efforts. If 10% of this 1% are interested in becoming entrepreneurs, we will have about 200 people to train. Initial enquiries convince us that many groups that want to promote entrepreneurship may be interested in working with our selection. We believe that we can generate an army of entrepreneurs. If we succeed in this endeavour, we would really have achieved a pyramid structure.

The benefits of our effort are also in a pyramid structure. At the lowest level, the learners get IT literacy. At the next level, we have students passing exams, because of our training material, see a testimonial:

In my college, one of the students in btech 3rd year 1st sem was having a makeup exam and and he was looking for guidance in learning Java. We gave the spoken-tutorial CD material on Java, and gave explanation on the contents of the CD. After the exam he came and told that the spoken tutorial CD on java helped him a lot and that he developed confidence in Java by going thru all the tutorials and doing the assignments. He also told that the video tutorials cleared most of his doubts on java and helped him in passing the makeup exam. -- Prof. K. V. Nagarjuna, Sree Dattha Inst. of Engg. and Science

Then, there are several levels of employment, starting from routine IT work, all the way up to work in niche areas, with attractive salaries. Finally, there is a possibility of one starting one’s own company.

Aakash: World’s lowest cost computing device

The agency that funded our FOSS promotion projects has created several e-content resources. It has also provided large amounts of bandwidth to educational institutions. These two do not get utilised effectively if the students do not have an affordable access device. If a student does not have an access device, they cannot participate in some of our projects. This affects their learning, while simultaneously resulting in loss of possible honorarium income. Aakash is the result of our efforts to address this problem [mpsv13], [sp13].



Fig. 7: Spoken Tutorials run on Aakash

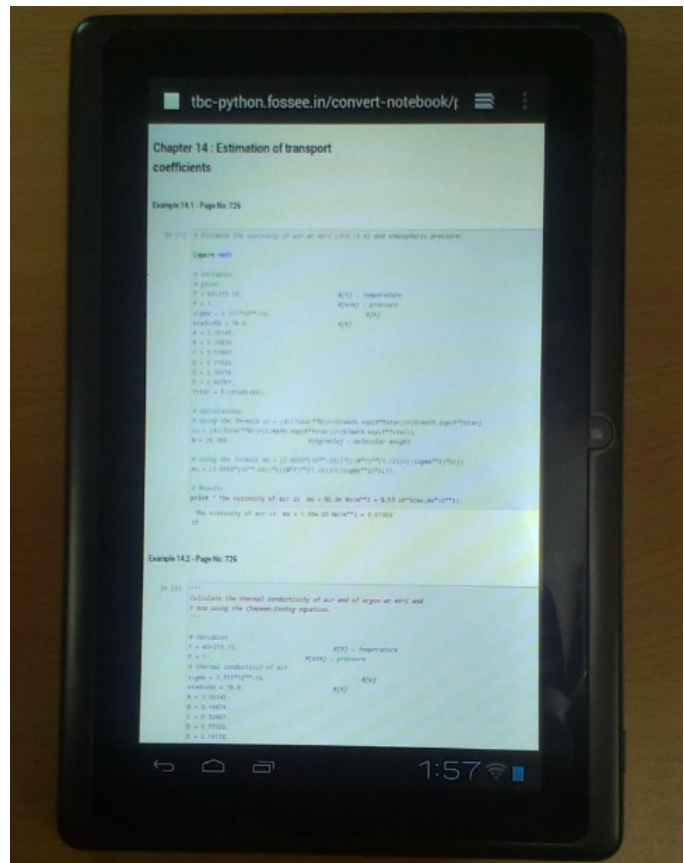


Fig. 8: A Python Textbook Companion on Aakash, the world’s lowest cost computing device.

Aakash has indeed become a convergence device for our projects. Spoken Tutorials can be played on Aakash, see Fig. 7. A PTC running on Aakash is shown in Fig. 8.

Conclusions and Future Work

This article has summarised how the FOSS promotion work we started in 2009 has evolved into a massive training programme that caters to the bottom of the pyramid and also to those at the top. Given that our approach has worked for IT skills development, we are exploring the possibility of replicating this method to other skills development areas as well. It will be great if we can succeed in this, as India has a big shortage of skilled personnel and a large

IT	Information Technology
FOSS	Free and open source software
FOSSEE	Free and open source software for education
PTC	Python Textbook Companion
SELF	Spoken Tutorial based Education and Learning through free FOSS study
ST	Spoken Tutorial
TBC	Textbook Companion

[python-ws-info]	Spoken Tutorial Team. List of Python workshops in Gujarat. http://www.spoken-tutorial.org/completed_workshops_list/GUJ?page=42&sort=asc&order=FOSS , Last seen on 29 June 2014.
[rmd]	recordMyDesktop Team., http://sourceforge.net/projects/recordmydesktop/ , Last seen on 27 June 2014.
[sp13]	S. Patil and S. Patnaik. GNU/Linux on Aakash. CSI Communications, pages 28–31, July 2013. Available at http://aakashlabs.org/media/pubs/GNU_Linux_on_Aakash.pdf .
[Spoken]	Spoken Tutorial Project. Official web page. http://spoken-tutorial.org/ , seen on 11 Feb. 2014.

TABLE 4

number of youngsters who want employment. The training may have to start at school level and this is an order of magnitude larger problem. Finally, all our material and processes are in the open and are available to FOSS enthusiasts all over the world.

Abbreviations

Acknowledgements

The work reported in this article has been carried out by the 100+ staff members of the FOSSEE and Spoken Tutorial teams. The author wishes to acknowledge the contributions of the Principal Investigators of these projects. The author wants to thank Prabhu Ramachandran for his help in converting this article to the required format.

REFERENCES

- [events-team] Spoken Tutorial Project. Events team contact details. http://process.spoken-tutorial.org/index.php/Software-Training#Organising_Workshops, seen on 29 June 2014.
- [forums] Spoken Tutorial Project. Online forum. <http://forums.spoken-tutorial.org/>, seen on 11 Feb. 2014.
- [FOSSEE] FOSSEE Team. Free and open source software in education. <http://fossee.in>, Seen on 11 Feb. 2014.
- [guidelines] Spoken Tutorial Team. Guidelines to create spoken tutorials. See http://process.spoken-tutorial.org/index.php/FOSS_Stages/Checklist, seen on 11 Feb. 2014.
- [kmm11-TUGboat] K. M. Moudgalya. LaTeX Training through Spoken Tutorials. TUGboat, 32(3):251–257, 2011.
- [kmm14] K. M. Moudgalya. Pedagogical and organisational issues in the campaign for it literacy through spoken tutorials. In R. Huang, Kinshuk, and N.-S. Chen, editors, *The new development of technology enhanced learning*, chapter 13. Springer-Verlag, Berlin Heidelberg, 2014.
- [LM] Scilab Team of FOSSEE. Matlab to Scilab lab migration. http://www.scilab.in/Lab_Migration_Project, Last seen on 5 July 2014.
- [mpsv13] K. M. Moudgalya, D. B. Phatak, N. K. Sinha, and Pradeep Varma. Genesis of Aakash 2. CSI Communications, pages 21--23 and 29, Jan. 2013. Available at <http://aakashlabs.org/media/pubs/genesis-reprint.pdf>, seen on 11 Feb. 2014.
- [nm09] Ministry of Human Resource Development. National mission on education through ICT. <http://www.sakshat.ac.in>, Last seen on 11 Feb. 2014.
- [oscad-book] Y. Save, R Rakhi, N. D. Shambulingayya, R. M. Rokade, A. Srivastava, M. R. Das, L. Pereira, S. Patil, S. Patnaik, and K. M. Moudgalya. Oscad: An open source EDA tool for circuit design, simulation, analysis and PCB design. Shroff Publishers, Mumbai, 2013.
- [oscad-lj] R. Rakhi and K. M. Moudgalya. Oscad: open source computer aided design tool. Linux Journal, pages 96–113, May 2014.
- [PTC] Python Team of FOSSEE. Python textbook companion. <http://tbc-python.fossee.in>, Seen on 19 June 2014.

Python for research and teaching economics

David R. Pugh^{‡*}

<http://www.youtube.com/watch?v=xHkGW115X8k>

Abstract—Together with theory and experimentation, computational modeling and simulation has become a “third pillar” of scientific inquiry. I am developing a curriculum for a three part, graduate level course on computational methods designed to increase the exposure of graduate students and researchers in the School of Economics at the University of Edinburgh to basic techniques used in computational modeling and simulation using the Python programming language. My course requires no prior knowledge or experience with computer programming or software development and all current and future course materials will be made freely available on-line via GitHub.

Index Terms—python, computational economics, dynamic economic models, numerical methods

Introduction

Together with theory and experimentation, computational modeling and simulation has become a “third pillar” of scientific inquiry. In this paper, I discuss the goals, objectives, and pedagogical choices that I made in designing and teaching a Python-based course on computational modeling and simulation to first-year graduate students in the Scottish Graduate Programme in Economics (SGPE) at the University of Edinburgh. My course requires no prior knowledge or experience with computer programming or software development and all current and future course materials will be made freely available [on-line](#).¹

Like many first-year PhD students, I began my research career with great faith in the analytic methods that I learned as an undergraduate and graduate student. While I was aware that economic models without closed-form solutions did exist, at no time during my undergraduate or graduate studies was I presented with an example of an important economic result that could not be analytically derived. While these analytic results were often obtained by making seemingly restrictive assumptions, the manner in which these assumptions were often justified gives the impression that such assumptions did not substantially impact the economic content of the result. As such, I started work as a PhD student under the impression that most all “interesting” economic research

questions could, and perhaps even should, be tackled analytically. Given that both of the leading graduate-level micro and macro-economics textbooks, [mas-colell1995] and [romer2011], fail to mention that computational methods are needed to fully solve even basic economic models, I do not believe that I was alone in my ignorance of the import of these methods in economics.

Fortunately (or unfortunately?) I was rudely awakened to the reality of modern economics research during my first year as a PhD student. Most economic models, particularly dynamic economic models, exhibit essential non-linearities or binding constraints that render them analytically intractable. Faced with reality I was confronted with two options: give up my original PhD research agenda, which evidently required computational methods, in favor of a modified research program that I could pursue with analytic techniques; or teach myself the necessary numerical techniques to pursue my original research proposal. I ended up spending the better part of two (out of my allotted three!) years of my PhD teaching myself computational modeling and simulation methods. The fact that I spent two-thirds of my PhD learning the techniques necessary to pursue my original research agenda indicated, to me at least, that there was a substantial gap in the graduate economics training at the University of Edinburgh. In order to fill this gap, I decided to develop a three-part course on computational modeling and simulation.

The first part of my course is a suite of Python-based, interactive laboratory sessions designed to expose students to the basics of scientific programming in Python. The second part of the course is a week-long intensive computational methods “boot camp.” The boot camp curriculum focuses on deepening students’ computer programming skills using the Python programming language and teaching important software design principles that are crucial for generating high-quality, reproducible scientific research using computational methods. The final part of the course, which is very much under development, will be an advanced training course targeted at PhD students and will focus on applying more cutting edge computational science techniques to economic problems via a series of interactive lectures and tutorials.

Why Python?

Python is a modern, object-oriented programming language widely used in academia and private industry, whose clean, yet expressive syntax, makes it an easy programming language to learn while still remaining powerful enough for serious scientific computing.² Python’s syntax was designed from the start with the human reader in mind and generates code that is easy to understand and debug which shortens development time relative to

* Corresponding author: pugh@maths.ox.ac.uk

‡ School of Economics, University of Edinburgh; Institute for New Economic Thinking at the Oxford Martin School and Oxford Mathematical Institute, University of Oxford

Copyright © 2014 David R. Pugh. This is an open-access article distributed under the terms of the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original author and source are credited.

1. This course would not have been possible without generous funding and support from the Scottish Graduate Programme in Economics (SGPE), the Scottish Institute for Research in Economics (SIRE), the School of Economics at the University of Edinburgh, and the Challenge Investment Fund (CIF).

low-level, compiled languages such as Fortran and C++. Among the high-level, general purpose languages, Python has the largest number of Matlab-style library modules (both installed in the standard library and through additional downloads) which meaning that one can quickly construct sophisticated scientific applications. While the Python programming language has found widespread use in private industry and many fields within academia, the capabilities of Python as a research tool remain relatively unknown within the economics research community. Notable exceptions are [stachurski2009] and [sargent2014].

Python is completely free and platform independent, making it a very attractive option as a teaching platform relative to other high-level scripting languages, particularly Matlab. Python is also open-source, making it equally attractive as a research tool for scientists interested in generating computational results that are more easily reproducible.³ Finally, Python comes with a powerful interactive interpreter that allows real-time code development and live experimentation. The functionality of the basic Python interpreter can be greatly increased by using the Interactive Python (or IPython) interpreter. Working via the Python or IPython interpreter eliminates the time-consuming (and productivity-destroying) compilation step required when working with low-level languages at the expense of slower execution speed. In many cases, it may be possible to achieve the best of both worlds using "mixed language" programming as Python can be easily extended by wrapping compiled code written in Fortran, C, or C++ using libraries such as f2Py, Cython, or swig. See [oliphant2007], [peterson2009], [behnel2011], [van2011] and references therein for more details.

Motivating the use of numerical methods in economics

The typical economics student enters graduate school with great faith in the analytical mathematical tools that he or she was taught as an undergraduate. In particular this student is under the impression that virtually all economic models have closed-form solutions. At worst the typical student believes that if he or she were to encounter an economic model without a closed-form solution, then simplifying assumptions could be made that would render the model analytically tractable without sacrificing important economic content.

The typical economics student is, of course, wrong about general existence of closed-form solutions to economic models. In fact the opposite is true: most economic models, particular dynamic, non-linear models with meaningful constraints (i.e., most any *interesting* model) will fail to have an analytic solution. In order to demonstrate this fact and thereby motivate the use of numerical methods in economics, I begin my course with a laboratory session on the Solow model of economic growth [solow1956].

Economics graduate student are very familiar with the Solow growth model. For many students, the Solow model will have

2. A non-exhaustive list of organizations currently using Python for scientific research and teaching: MIT's legendary *Introduction to Computer Science and Programming*, CS 6.00, is taught using Python; Python is the in-house programming language at Google; NASA, CERN, Los Alamos National Labs (LANL), Lawrence Livermore National Labs (LLNL), and Industrial Light and Magic (ILM) all rely heavily on Python.

3. The Python Software Foundation License (PSFL) is a BSD-style license that allows a developer to sell, use, or distribute his Python-based application in anyway he sees fit. In addition, the source code for the entire Python scientific computing stack is available on GitHub making it possible to directly examine the code for any specific algorithm in order to better understand exactly how a result has been obtained.

been one of the first macroeconomic models taught to them as undergraduates. Indeed, the dominant macroeconomics textbook for first and second year undergraduates, [mankiw2010], devotes two full chapters to motivating and deriving the Solow model. The first few chapters of [romer2011], one of the most widely used final year undergraduate and first-year graduate macroeconomics textbook, are also devoted to the Solow growth model and its descendants.

The Solow growth model

The Solow model boils down to a single non-linear differential equation and associated initial condition describing the time evolution of capital stock per effective worker, $k(t)$.

$$\dot{k}(t) = sf(k(t)) - (n + g + \delta)k(t), \quad k(t) = k_0$$

The parameter $0 < s < 1$ is the fraction of output invested and the parameters n, g, δ are the rates of population growth, technological progress, and depreciation of physical capital. The intensive form of the production function f is assumed to be strictly concave with

$$f(0) = 0, \quad \lim_{k \rightarrow 0} f' = \infty, \quad \lim_{k \rightarrow \infty} f' = 0.$$

A common choice for the function f which satisfies the above conditions is known as the Cobb-Douglas production function.

$$f(k) = k^\alpha$$

Assuming a Cobb-Douglas functional form for f also makes the model analytically tractable (and thus contributes to the typical economics student's belief that all such models "must" have an analytic solution). [sato1963] showed that the solution to the model under the assumption of Cobb-Douglas production is

$$k(t) = \left[\left(\frac{s}{n + g + \delta} \right) \left(1 - e^{-(n+g+\delta)(1-\alpha)t} \right) + k_0^{1-\alpha} e^{-(n+g+\delta)(1-\alpha)t} \right]^{\frac{1}{1-\alpha}}.$$

A notable property of the Solow model with Cobb-Douglas production is that the model predicts that the shares of real income going to capital and labor should be constant. Denoting capital's share of income as $\alpha_K(k)$, the model predicts that

$$\alpha_K(k) \equiv \frac{\partial \ln f(k)}{\partial \ln k} = \alpha$$

Unfortunately, from figure 1 it is clear that the prediction of constant factor shares is strongly at odds with the empirical data for most countries. Fortunately, there is a simple generalization of the Cobb-Douglas production function, known as the constant elasticity of substitution (CES) function, that is capable of generating the variable factor shares observed in the data.

$$f(k) = \left[\alpha k^\rho + (1 - \alpha) \right]^{\frac{1}{\rho}}$$

where $-\infty < \rho < 1$ is the elasticity of substitution between capital and effective labor in production. Note that

$$\lim_{\rho \rightarrow 0} f(k) = k^\alpha$$

and thus the CES production function nests the Cobb-Douglas functional form as a special case. To see that the CES production function also generates variable factor shares note that

$$\alpha_K(k) \equiv \frac{\partial \ln f(k)}{\partial \ln k} = \frac{\alpha k^\rho}{\alpha k^\rho + (1 - \alpha)}$$

which varies with k .

This seemingly simple generalization of the Cobb-Douglas production function, which is necessary in order for the Solow model generate variable factor share, an economically important feature of the post-war growth experience in most countries, renders the Solow model analytically intractable. To make progress solving a Solow growth model with CES production one needs to resort to computational methods.

Numerically solving the Solow model

A computational solution to the Solow model allows me to demonstrate a number of numerical techniques that students will find generally useful in their own research.

First and foremost, solving the model requires efficiently and accurately approximating the solution to a non-linear ordinary differential equation (ODE) with a given initial condition (i.e., an non-linear initial value problem). Finite-difference methods are commonly employed to solve such problems. Typical input to such algorithms is the Jacobian matrix of partial derivatives of the system of ODEs. Solving the Solow growth model allows me to demonstrate the use of finite difference methods as well as how to compute Jacobian matrices of non-linear systems of ODEs.

Much of the empirical work based on the Solow model focuses on the model's predictions concerning the long-run or steady state equilibrium of the model. Solving for the steady state of the Solow growth model requires solving for the roots of a non-linear equation. Root finding problems, which are equivalent to solving systems of typically non-linear equations, are one of the most widely encountered computational problems in economic applications. Typical input to root-finding algorithms is the Jacobian matrix of partial derivatives of the system of non-linear equations. Solving for the steady state of the Solow growth model allows me to demonstrate the use of various root finding algorithms as well as how to compute Jacobian matrices of non-linear systems of equations.

Finally, given some data, estimation of the model's structural parameters (i.e., g , n , s , α , δ , ρ) using either as maximum likelihood or non-linear least squares requires solving a non-linear, constrained optimization problem. Typical inputs to algorithms for solving such non-linear programs are the Jacobian and Hessian of the objective function with respect to the parameters being estimated.⁴ Thus structural estimation also allows me to demonstrate the symbolic and numerical differentiation techniques needed to compute the Jacobian and Hessian matrices.

Course outline

Having motivated the need for computational methods in economics, in this section I outline the three major components of my computational methods course: laboratory sessions, an intensive week-long Python boot camp, and an advanced PhD training course. The first two components are already up and running (thanks to funding support from the SGPE, SIRE, and the CIF). I am still looking to secure funding to develop the advanced PhD training course component.

Laboratory sessions

The first part of the course is a suite of Python-based laboratory sessions that run concurrently as part of the core macroeconomics

sequence. There are 8 labs in total: two introductory sessions, three labs covering computational methods for solving models that students are taught in macroeconomics I (fall term), three labs covering computational methods for solving models taught in macroeconomics II (winter term). The overall objective of these laboratory sessions is to expose students to the basics of scientific computing using Python in a way that reinforces the economic models covered in the lectures. All of the laboratory sessions make use of the excellent IPython notebooks.

The material for the two introductory labs draws heavily from [part I](#) and [part II](#) of [Quantitative Economics](#) by Thomas Sargent and John Stachurski. In the first lab, I introduce and motivate the use of the Python programming language and cover the bare essentials of Python: data types, imports, file I/O, iteration, functions, comparisons and logical operators, conditional logic, and Python coding style. During the second lab, I attempt to provide a quick overview of the Python scientific computing stack (i.e., IPython, Matplotlib, NumPy, Pandas, and SymPy) with a particular focus on those pieces that students will encounter repeatedly in economic applications.

The material for the remaining 6 labs is designed to complement the core macroeconomic sequence of the SGPE and thus varies a bit from year to year. During the 2013-2014 academic year I covered the following material:

- **Initial value problems:** Using the [\[solow1956\]](#) model of economic growth as the motivating example, I demonstrate finite-difference methods for efficiently and accurately solving initial value problems of the type typically encountered in economics.
- **Boundary value problems:** Using the neo-classical optimal growth model of [\[ramsey1928\]](#), [\[cass1965\]](#), and [\[koopmans1965\]](#) as the motivating example, I demonstrate basic techniques for efficiently and accurately solving two-point boundary value problems of the type typically encountered in economics using finite-difference methods (specifically forward, reverse, and multiple shooting).
- **Numerical dynamic programming:** I demonstrate basic techniques for solving discrete-time, stochastic dynamic programming problems using a stochastic version of the neo-classical optimal growth model as the motivating example.
- **Real business cycle models:** I extend the stochastic optimal growth model to incorporate a household labor supply decision and demonstrate how to approximate the model solution using *dynare++*, a C++ library specializing in computing k -order Taylor approximations of dynamic stochastic general equilibrium (DSGE) models.

In future versions of the course I hope to include laboratory sessions on DSGE monetary policy models, DSGE models with financial frictions, and models of unemployment with search frictions. These additional labs are likely to be based around dissertations being written by current MSc students.

Python boot camp

Whilst the laboratory sessions expose students to some of the basics of programming in Python as well as numerous applications of computational methods in economics, these lab sessions are inadequate preparation for those students wishing to apply such methods as part of their MSc dissertations or PhD theses.

4. The Hessian matrix is also used for computing standard errors of parameter estimates.

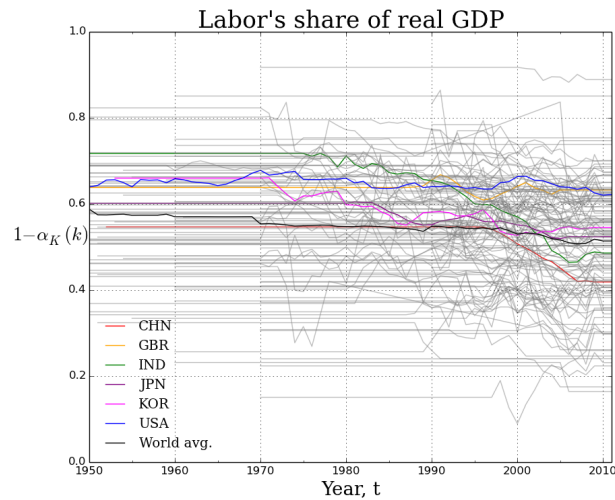


Fig. 1: Labor's share of real GDP has been declining, on average, for much of the post-war period. For many countries, such as India, China, and South Korea, the fall in labor's share has been dramatic.

In order to provide interested students with the skills needed to apply computational methods in their own research I have developed a week-long intensive computational methods "boot camp." The boot camp requires no prior knowledge or experience with computer programming or software development and all current and future course materials are made freely available online.

Each day of the boot camp is split into morning and afternoon sessions. The morning sessions are designed to develop attendees Python programming skills while teaching important software design principles that are crucial for generating high-quality, reproducible scientific research using computational methods. The syllabus for the morning sessions closely follows [Think Python](#) by Allen Downey.

In teaching Python programming during the boot camp I subscribe to the principle of "learning by doing." As such my primary objective on day one of the Python boot camp is to get attendees up and coding as soon as possible. The goal for the first morning session is to cover the first four chapters of *Think Python*.

- [Chapter 1](#): The way of the program;
- [Chapter 2](#): Variables, expressions, and statements;
- [Chapter 3](#): Functions;
- [Chapter 4](#): Case study on interface design.

The material in these introductory chapters is clearly presented and historically students have generally had no trouble interactively working through the all four chapters before the lunch break. Most attendees break for lunch on the first day feeling quite good about themselves. Not only have they covered a lot of material, they have managed to write some basic computer programs. Maintaining student confidence is important: as long as students are confident and feel like they are progressing, they will remain focused on continuing to build their skills. If students get discouraged, perhaps because they are unable to solve a certain exercise or decipher a cryptic error traceback, they will lose their focus and fall behind.

The second morning session covers the next three chapters of *Think Python*:

- [Chapter 5](#): Conditionals and recursion;

- [Chapter 6](#): Fruitful functions;
- [Chapter 7](#): Iteration.

At the start of the session I make a point to emphasize that the material being covered in chapters 5-7 is substantially more difficult than the introductory material covered in the previous morning session and that I do not expect many students to make it through the all of material before lunch. The idea is to manage student expectations by continually reminding them that the course is designed in order that they can learn at their own pace

The objective of for the third morning session is the morning session of day three the stated objective is for students to work through the material in chapters 8-10 of [Think Python](#).

- [Chapter 8](#): Strings;
- [Chapter 9](#): A case study on word play;
- [Chapter 10](#): Lists.

The material covered in [chapter 8](#) and [chapter 10](#) is particularly important as these chapters cover two commonly used Python data types: strings and lists. As a way of drawing attention to the importance of chapters 8 and 10, I encourage students to work through both of these chapters before returning to [chapter 9](#).

The fourth morning session covers the next four chapters of *Think Python*:

- [Chapter 11](#): Dictionaries;
- [Chapter 12](#): Tuples;
- [Chapter 13](#): Case study on data structure selection;
- [Chapter 14](#): Files.

The morning session of day four is probably the most demanding. Indeed many students take two full session to work through this material. Chapters 11 and 12 cover two more commonly encountered and important Python data types: dictionaries and tuples. [Chapter 13](#) is an important case study that demonstrates the importance of thinking about data structures when writing library code.

The final morning session is designed to cover the remaining five chapters of [Think Python](#) on object-oriented programming (OOP):

- [Chapter 15](#): Classes and Objects;
- [Chapter 16](#): Classes and Functions;
- [Chapter 17](#): Classes and Methods;
- [Chapter 18](#): Inheritance;
- [Chapter 19](#): Case Study on Tkinter.

While this year a few students managed to get through at least some of the OOP chapters, the majority of students managed only to get through chapter 13 over the course of the five, three-hour morning sessions. Those students who did manage to reach the OOP chapters in general failed to grasp the point of OOP and did not see how they might apply OOP ideas in their own research. I see this as a major failing of my teaching as I have found OOP concepts to be incredibly useful in my own research. [stachurski2009], and [sargent2014] also make heavy use of OOP techniques.

While the morning sessions focus on building the foundations of the Python programming language, the afternoon sessions are devoted to demonstrating the use of Python in scientific computing by exploring in greater detail the Python scientific computing stack. During the afternoon session on day one I motivate the use of Python in scientific computing and spend considerable time getting students set up with a suitable Python environment and demonstrating the basic scientific work flow.

I provide a quick tutorial of the Enthought Canopy distribution. I then discuss the importance of working with a high quality text editor, such as Sublime, and make sure that students have installed both Sublime as well as the relevant Sublime plug-ins (i.e., SublimeGit and LatexTools for Git and LaTeX integration, respectively; SublimeLinter for code linting, etc). I make sure that students can install Git and stress the importance of using distributed version control software in scientific computing and collaboration. Finally I cover the various flavors of the IPython interpreter: the basic IPython terminal, IPython QTconsole, and the IPython notebook.

The afternoon curriculum for days two through five is built around the [Scientific Programming in Python](#) lecture series and supplemented with specific use cases from my own research. My goal is to cover all of the material in lectures 1.3, 1.4, and 1.5 covering NumPy, Matplotlib and SciPy, respectively. In practice I am only able to cover a small subset of this material during the afternoon sessions.

Advanced PhD training course

The final part of the course (for which I am still seeking funding to develop!) is a six week PhD advanced training course that focuses on applying cutting edge computational science techniques to economic problems via a series of interactive lectures and tutorials. The curriculum for this part of the course will derive primarily from [judd1998], [stachurski2009], and parts III and IV of [sargent2014]. In particular, I would like to cover the following material.

- Linear equations and iterative methods: Gaussian elimination, LU decomposition, sparse matrix methods, error analysis, iterative methods, matrix inverse, ergodic distributions over-identified systems.
- Optimization: 1D minimization, multi-dimensional minimization using comparative methods, Newton's method for multi-dimensional minimization, directed set methods for multi-dimensional minimization, non-linear least squares, linear programming, constrained non-linear optimization.

- Non-linear equations: 1D root-finding, simple methods for multi-dimensional root-finding, Newton's method for multi-dimensional root-finding, homotopy continuation methods.
- Approximation methods: local approximation methods, regression as approximation, orthogonal polynomials, least-squares orthogonal polynomial approximation, uniform approximation, interpolation, piece-wise polynomial interpolation, splines, shape-preserving approximation, multi-dimensional approximation, finite-element approximations.
- Economic applications: finite-state Markov chains, linear state space models, the Kalman filter, dynamic programming, linear-quadratic control problems, continuous-state Markov chains, robust control problems, linear stochastic models.

Conclusion

In this paper I have outlined the three major components of my computational methods course: laboratory sessions, an intensive week-long Python boot camp, and an advanced PhD training course. The first two components are already up and running (thanks to funding support from the SGPE, SIRE, and the CIF). I am still looking to secure funding to develop the advanced PhD training course component.

I have been pleasantly surprised at the eagerness of economics graduate students both to learn computational modeling and simulation methods and to apply these techniques to the analytically intractable problems that they are encountering in their own research. Their eagerness to learn is, perhaps, a direct response to market forces. Both within academia, industry, and the public sector there is an increasing demand for both applied and theoretical economists interested in inter-disciplinary collaboration. The key to developing and building the capacity for inter-disciplinary research is effective communication using a common language. Historically that common language has been mathematics. Increasingly, however, this language is becoming computation. It is my hope that the course outlined in this paper might served as a prototype for other Python-based computational methods courses for economists and other social scientists.

REFERENCES

- [behnel2011] S. Behnel, et al. *Cython: The best of both worlds*, Computing in Science and Engineering, 13(2):31-39, 2011.
- [cass1965] D. Cass. *Optimum growth in an aggregative model of capital accumulation*, Review of Economic Studies, 32, 233-240.
- [judd1998] K. Judd. *Numerical Methods for Economists*, MIT Press, 1998.
- [koopmans1965] T. Koopmans. *On the concept of optimal economic growth*, Econometric Approach to Development Planning, 225-87. North-Holland, 1965.
- [mankiw2010] N.G. Mankiw. *Intermediate Macroeconomics, 7th edition*, Worth Publishers, 2010.
- [mas-colell1995] A.Mas-Colell, et al. *Microeconomic Theory, 7th edition*, Oxford University Press, 1995.
- [oliphant2007] T. Oliphant. *Python for scientific computing*, Computing in Science and Engineering, 9(3):10-20, 2007.
- [peterson2009] P. Peterson. *F2PY: a tool for connecting Fortran and Python programs*, International Journal of Computational Science and Engineering, 4(4):296-305, 2009.
- [ramsey1928] F. Ramsey. *A mathematical theory of saving*, Economic Journal, 38(152), 543-559.

- [romer2011] D. Romer. *Advanced Macroeconomics, 4th edition*, MacGraw Hill, 2011.
- [sargent2014] T. Sargent and J. Stachurski. *Quantitative Economics*, 2014.
- [sato1963] R. Sato. *Fiscal policy in a neo-classical growth model: An analysis of time required for equilibrating adjustment*, Review of Economic Studies, 30(1):16-23, 1963.
- [solow1956] R. Solow. *A contribution to the theory of economic growth*, Quarterly Journal of Economics, 70(1):64-95, 1956.
- [stachurski2009] J. Stachurski. *Economic dynamics: theory and computation*, MIT Press, 2009.
- [van2011] S. Van Der Walt, et al. *The NumPy array: a structure for efficient numerical computation*, Computing in Science and Engineering, 13(2):31-39, 2011.

Validated numerics with Python: the ValidiPy package

David P. Sanders^{‡*}, Luis Benet[§]



Abstract—We introduce the ValidiPy package for *validated numerics* in Python. This suite of tools, which includes interval arithmetic and automatic differentiation, enables *rigorous* and *guaranteed* results using floating-point arithmetic. We apply the ValidiPy package to two classic problems in dynamical systems, calculating periodic points of the logistic map, and simulating the dynamics of a chaotic billiard model.

Index Terms—validated numerics, Newton method, floating point, interval arithmetic

Floating-point arithmetic

Scientific computation usually requires the manipulation of real numbers. The standard method to represent real numbers internally in a computer is floating-point arithmetic, in which a real number a is represented as

$$a = \pm 2^e \times m.$$

The usual double-precision (64-bit) representation is that of the [IEEE 754 standard](#) [[IEEE754](#)]: one bit is used for the sign, 11 bits for the exponent e , which ranges from -1022 to $+1023$, and the remaining 52 bits are used for the "mantissa" m , a binary string of 53 bits, starting with a 1 which is not explicitly stored.

However, most real numbers are *not explicitly representable* in this form, for example 0.1, which in binary has the infinite periodic expansion

$$0.0\ 0011\ 0011\ 0011\ 0011\ \dots,$$

in which the pattern 0011 repeats forever. Representing this in a computer with a finite number of digits, via truncation or rounding, gives a number that differs slightly from the true 0.1, and leads to the following kinds of problems. Summing 0.1 many times -- a common operation in, for example, a time-stepping code, gives the following *unexpected* behaviour.

```
a = 0.1
total = 0.0
print("%20s %25s" % ("total", "error"))
for i in xrange(1000):
    if i%100 == 0 and i>0:
        error = total - i/10
```

* Corresponding author: dpsanders@ciencias.unam.mx
[‡] Department of Physics, Faculty of Sciences, National Autonomous University of Mexico (UNAM), Ciudad Universitaria, México D.F. 04510, Mexico
[§] Institute of Physical Sciences, National Autonomous University of Mexico (UNAM), Apartado postal 48-3, Cuernavaca 62551, Morelos, Mexico

Copyright © 2014 David P. Sanders et al. This is an open-access article distributed under the terms of the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original author and source are credited.

```
print("%20.16g %25.16g" % (total, error))
total += a
```

total	error
9.999999999999998	-1.953992523340276e-14
20.000000000000001	1.4210854715202e-14
30.000000000000016	1.56319401867222e-13
40.00000000000003	2.984279490192421e-13
50.00000000000044	4.405364961712621e-13
60.00000000000058	5.826450433232822e-13
70.0000000000003	2.984279490192421e-13
79.9999999999973	-2.700062395888381e-13
89.9999999999916	-8.384404281969182e-13

Here, the result oscillates in an apparently "random" fashion around the expected value.

This is already familiar to new users of any programming language when they see the following kinds of outputs of elementary calculations [[Gold91](#)]:

```
3.2 * 4.6
14.719999999999999
```

Suppose that we now apply an algorithm starting with an initial condition $x_0 = 0.1$. The result will be erroneous, since the initial condition used differs slightly from the desired value. In chaotic systems, for example, such a tiny initial deviation may be quickly magnified and destroy all precision in the computation. Although there are methods to estimate the resulting errors [[High96](#)], there is no guarantee that the true result is captured. Another example is certain ill-conditioned matrix computations, where small changes to the matrix lead to unexpectedly large changes in the result.

Interval arithmetic

Interval arithmetic is one solution for these difficulties. In this method, developed over the last 50 years but still relatively unknown in the wider scientific community, all quantities in a computation are treated as closed intervals of the form $[a, b]$. If the initial data are contained within the initial intervals, then the result of the calculation is *guaranteed* to contain the true result. To accomplish this, the intervals are propagated throughout the calculation, based on the following ideas:

- 1) All intervals must be *correctly rounded*: the lower limit a of each interval is rounded downwards (towards $-\infty$) and the upper limit b is rounded upwards (towards $+\infty$). [The availability of these rounding operations is standard on modern computing hardware.] In this way, the interval is guaranteed to contain the true result. If we do not apply rounding, then this might not be the case; for example, the interval given by $I = Interval(0.1, 0.2)$

does not actually contain the true 0.1 if the standard floating-point representation for the lower endpoint is used; instead, this lower bound corresponds to 0.10000000000000000555111....

- 2) Arithmetic operations are defined on intervals, such that the result of an operation on a pair of intervals is the interval that is the *result of performing the operation on any pair of numbers, one from each interval*.
- 3) Elementary functions are defined on intervals, such that the result of an elementary function f applied to an interval I is the *image* of the function over that interval, $f(I) := \{f(x) : x \in I\}$.

For example, addition of two intervals is defined as

$$[a, b] + [c, d] := \{x + y : x \in [a, b], y \in [c, d]\},$$

which turns out to be equivalent to

$$[a, b] + [c, d] := [a + c, b + d].$$

The exponential function applied to an interval is defined as

$$\exp([a, b]) := [\exp(a), \exp(b)],$$

giving the exact image of the monotone function \exp evaluated over the interval.

Once all required operations and elementary functions (such as \sin , \exp etc.) are correctly defined, and given a technical condition called "inclusion monotonicity", for any function $f : \mathbb{R} \rightarrow \mathbb{R}$ made out of a combination of arithmetic operations and elementary functions, we may obtain the *interval extension* \tilde{f} . This is a "version" of the function which applies to intervals, such that when we apply \tilde{f} to an interval I , we obtain a new interval $\tilde{f}(I)$ that is *guaranteed to contain* the true, mathematical image $f(I) := \{f(x) : x \in I\}$.

Unfortunately, $\tilde{f}(I)$ may be strictly larger than the true image $f(I)$, due to the so-called *dependency problem*. For example, let $I := [-1, 1]$. Suppose that $f(x) := x * x$, i.e. that we wish to square all elements of the interval. The true image of the interval I is then $f(I) = [0, 1]$.

However, thinking of the squaring operation as repeated multiplication, we may try to calculate

$$I * I := \{xy : x \in I, y \in I\}.$$

Doing so, we find the *larger* interval $[-1, 1]$, since we "do not notice" that the x 's are "the same" in each copy of the interval; this, in a nutshell, is the dependency problem.

In this particular case, there is a simple solution: we calculate instead $I^2 := \{x^2 : x \in I\}$, so that there is only a single copy of I and the true image is obtained. However, if we consider a more complicated function like $f(x) = x + \sin(x)$, there does not seem to be a generic way to solve the dependency problem and hence find the exact range.

This problem may, however, be solved to an arbitrarily good approximation by splitting up the initial interval into a union of subintervals. When the interval extension is instead evaluated over those subintervals, the union of the resulting intervals gives an enclosure of the exact range that is increasingly better as the size of the subintervals decreases [Tuck11].

Validated numerics: the ValidiPy package

The name "validated numerics" has been applied to the combination of interval arithmetic, automatic differentiation, Taylor methods and other techniques that allow the rigorous solution of problems using finite-precision floating point arithmetic [Tuck11].

The ValidiPy package, a Python package for validated numerics, was initiated during a Masters' course on validated numerics that the authors taught in the Postgraduate Programmes in Mathematics and Physics at the National Autonomous University of Mexico (UNAM) during the second half of 2013. It is based on the excellent textbook *Validated Numerics* by Warwick Tucker [Tuck11], one of the foremost proponents of interval arithmetic today. He is best known for [Tuck99], in which he gave a rigorous proof of the existence of the Lorenz attractor, a strange (fractal, chaotic) attractor of a set of three ordinary differential equations modelling convection in the atmosphere that were computationally observed to be chaotic in 1963 [Lorenz].

Naturally, there has been previous work on implementing the different components of Validated Numerics in Python, such as `pyinterval` and `mpmath` for interval arithmetic, and `AlgoPy` for automatic differentiation. Our project is designed to provide an understandable and modifiable code base, with a focus on ease of use, rather than speed.

An incomplete sequence of IPython notebooks from the course, currently in Spanish, provide an introduction to the theory and practice of interval arithmetic; they are available on [GitHub](#) and for online viewing at [NbViewer](#).

Code in Julia is also available, in our package `ValidatedNumerics.jl` [ValidatedNumerics].

Implementation of interval arithmetic

As with many other programming languages, Python allows us to define new types, as `class` es, and to define operations on those types. The following working sketch of an `Interval` class may be extended to a full-blown implementation (which, in particular, must include directed rounding; see below), available in the [ValidiPy] repository.

```
class Interval(object):
    def __init__(self, a, b=None):
        # constructor

        if b is None:
            b = a

        self.lo = a
        self.hi = b

    def __add__(self, other):
        if not isinstance(other, Interval):
            other = Interval(other)
        return Interval(self.lo+other.lo,
                        self.hi+other.hi)

    def __mul__(self, other):
        if not isinstance(other, Interval):
            other = Interval(other)

        S = [self.lo*other.lo, self.lo*other.hi,
             self.hi*other.lo, self.hi*other.hi]
        return Interval(min(S), max(S))

    def __repr__(self):
        return "{}, {}".format(self.lo, self.hi)
```

Examples of creation and manipulation of intervals:

```

i = Interval(3)
i

[3, 3]

i = Interval(-3, 4)
i

[-3, 4]

i * i

[-12, 16]

def f(x):
    return x*x + x + 2

f(i)

[-13, 22]

```

To attain multiple-precision arithmetic and directed rounding, we use the `gmpy2` package [gmpy2]. This provides a wrapper around the MPFR [MPFR] C package for correctly-rounded multiple-precision arithmetic [Fous07]. For example, a simplified version of the `Interval` constructor may be written as follows, showing how the precision and rounding modes are manipulated using the `gmpy2` package:

```

import gmpy2
from gmpy2 import RoundDown, RoundUp

ctx = gmpy2.get_context()

def set_interval_precision(precision):
    gmpy2.get_context().precision = precision

def __init__(self, a, b=None):
    ctx.round = RoundDown
    a = mpfr(str(a))

    ctx.round = RoundUp
    b = mpfr(str(b))

    self.lo, self.hi = a, b

```

Each arithmetic and elementary operation must apply directed rounding in this way at each step; for example, the implementations of multiplication and exponentiation of intervals are as follows:

```

def __mult__(self, other):

    ctx.round = RoundDown
    S_lower = [ self.lo*other.lo, self.lo*other.hi,
                self.hi*other.lo, self.hi*other.hi ]
    S1 = min(S_lower)

    ctx.round = RoundUp
    S_upper = [ self.lo*other.lo, self.lo*other.hi,
                self.hi*other.lo, self.hi*other.hi ]
    S2 = max(S_upper)

    return Interval(S1, S2)

def exp(self):
    ctx.round = RoundDown
    lower = exp(self.lo)

    ctx.round = RoundUp

```

```

upper = exp(self.hi)

return Interval(lower, upper)

```

The Interval Newton method

As applications of interval arithmetic and of `ValidiPy`, we will discuss two classical problems in the area of dynamical systems. The first is the problem of locating all periodic orbits of the dynamics, with a certain period, of the well-known logistic map. To do so, we will apply the *Interval Newton method*.

The Newton (or Newton--Raphson) method is a standard algorithm for finding zeros, or roots, of a nonlinear equation, i.e. x^* such that $f(x^*) = 0$, where $f: \mathbb{R} \rightarrow \mathbb{R}$ is a nonlinear function.

The Newton method starts from an initial guess x_0 for the root x^* , and iterates

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)},$$

where $f': \mathbb{R} \rightarrow \mathbb{R}$ is the derivative of f . This formula calculates the intersection of the tangent line to the function f at the point x_n with the x -axis, and thus gives a new estimate of the root.

If the initial guess is sufficiently close to a root, then this algorithm converges very quickly ("quadratically") to the root: the number of correct digits doubles at each step.

However, the standard Newton method suffers from problems: it may not converge, or may converge to a different root than the intended one. Furthermore, there is no way to guarantee that all roots in a certain region have been found.

An important, but too little-known, contribution of interval analysis is a version of the Newton method that is modified to work with intervals, and is able to locate *all* roots of the equation within a specified interval I , by isolating each one in a small sub-interval, and to either guarantee that there is a unique root in each of those sub-intervals, or to explicitly report that it is unable to determine existence and uniqueness.

To understand how this is possible, consider applying the interval extension \tilde{f} of f to an interval I . Suppose that the image $\tilde{f}(I)$ does *not* contain 0. Since $f(I) \subset \tilde{f}(I)$, we know that $f(I)$ is *guaranteed* not to contain 0, and thus we guarantee that there *cannot be a root* x^* of f inside the interval I . On the other hand, if we evaluate f at the endpoints a and b of the interval $I = [a, b]$ and find that $f(a) < 0 < f(b)$ (or vice versa), then we can guarantee that there is *at least one root within the interval*.

The Interval Newton method does not just naively extend the standard Newton method. Rather, a new operator, the Newton operator, is defined, which takes an interval as input and returns as output either one or two intervals. The Newton operator for the function f is defined as

$$N_f(I) := m - \frac{f(m)}{\tilde{f}'(I)},$$

where $m := m(I)$ is the midpoint of the interval I , which may be treated as a (multi-precision) floating-point number, and $\tilde{f}'(I)$ is an interval extension of the derivative f' of f . This interval extension may easily be calculated using *automatic differentiation* (see below). The division is now a division by an interval, which is defined as for the other arithmetic operations. In the case when the interval $\tilde{f}'(I)$ contains 0, this definition leads to the result being the union of *two disjoint intervals*: if $I = [-a, b]$ with $a > 0$ and $b > 0$, then we define $1/I = (1/[-a, -0]) \cup (1/[0, b]) = [-\infty, -1/a] \cup [1/b, \infty]$.

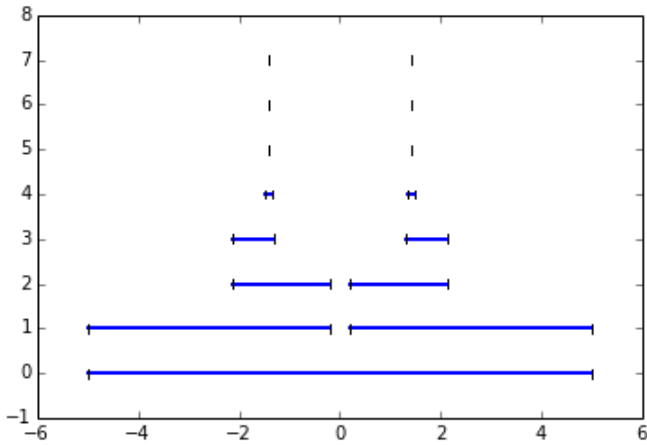


Fig. 1: Convergence of the Interval Newton method to the roots of 2.

The idea of this definition is that the result of applying the operator N_f to an interval I will necessarily contain the result of applying the standard Newton operator at all points of the interval, and hence will contain *all* possible roots of the function in that interval.

Indeed, the following strong results may be rigorously proved [Tuck11]: 1. If $N_f(I) \cap I = \emptyset$, then I contains no zeros of f ; 2. If $N_f(I) \subset I$, then I contains exactly one zero of f .

If neither of these options holds, then the interval I is split into two equal subintervals and the method proceeds on each. Thus the Newton operator is sufficient to determine the presence (and uniqueness) or absence of roots in each subinterval.

Starting from an initial interval I_0 , and iterating $I_{n+1} := I_n \cap N_f(I_n)$, gives a sequence of lists of intervals that is guaranteed to contain the roots of the function, as well as a guarantee of uniqueness in many cases.

The code to implement the Interval Newton method completely is slightly involved, and may be found in an IPython notebook in the `examples` directory at <<https://github.com/computo-fc/ValidiPy/tree/master/examples>>.

An example of the Interval Newton method in action is shown in figure 1, where it was used to find the roots of $f(x) = x^2 - 2$ within the initial interval $[-5, 5]$. Time proceeds vertically from bottom to top.

Periodic points of the logistic map

An interesting application of the Interval Newton method is to dynamical systems. These may be given, for example, as the solution of systems of ordinary differential equations, as in the Lorenz equations [Lor63], or by iterating maps. The *logistic map* is a much-studied dynamical system, given by the map

$$f(x) := f_r(x) := rx(1-x).$$

The dynamics is given by iterating the map:

$$x_{n+1} = f(x_n),$$

so that

$$x_n = f(f(f(\dots(x_0)\dots))) = f^n(x_0),$$

where f^n denotes $f \circ f \circ \dots \circ f$, i.e. f composed with itself n times.

Periodic points play a key role in dynamical system: these are points x such that $f^p(x) = x$; the minimal $p > 0$ for which this

is satisfied is the *period* of x . Thus, starting from such a point, the dynamics returns to the point after p steps, and then eternally repeats the same trajectory. In chaotic systems, periodic points are dense in phase space [Deva03], and properties of the dynamics may be calculated in terms of the periodic points and their stability properties [ChaosBook]. The numerical enumeration of all periodic points is thus a necessary part of studying almost any such system. However, standard methods usually do not guarantee that all periodic points of a given period have been found.

On the contrary, the Interval Newton method, applied to the function $g_p(x) := f^p(x) - x$, guarantees to find all zeros of the function g_p , i.e. all points with period at most p (or to explicitly report where it has failed). Note that this will include points of lower period too; thus, the periodic points should be enumerated in order of increasing period, starting from period 1, i.e. fixed points x such that $f(x) = x$.

To verify the application of the Interval Newton method to calculate periodic orbits, we use the fact that the particular case of f_4 the logistic map with $r = 4$ is *conjugate* (related by an invertible nonlinear change of coordinates) to a simpler map, the tent map, which is a piecewise linear map from $[0, 1]$ onto itself, given by

$$T(x) := \begin{cases} 2x, & \text{if } x < \frac{1}{2}; \\ 2-2x, & \text{if } x > \frac{1}{2}. \end{cases}$$

The n th iterate of the tent map has 2^n "pieces" (or "laps") with slopes of modulus 2^n , and hence exactly 2^n points that satisfy $T^n(x) = x$.

The i th "piece" of the n th iterate (with $i = 0, \dots, 2^n - 1$) has equation

$$T_i^n(x) = \begin{cases} 2^n x - i, & \text{if } i \text{ is even and } \frac{i}{2^n} \leq x < \frac{i+1}{2^n} \\ i+1 - 2^n x, & \text{if } i \text{ is odd and } \frac{i}{2^n} \leq x < \frac{i+1}{2^n} \end{cases}$$

Thus the solution of $T_i^n(x) = x$ satisfies

$$x_i^n = \begin{cases} \frac{i}{2^n - 1}, & \text{if } i \text{ is even;} \\ \frac{i+1}{1+2^n}, & \text{if } i \text{ is odd,} \end{cases}$$

giving the 2^n points which are candidates for periodic points of period n . (Some are actually periodic points with period p that is a proper divisor of n , satisfying also $T^p(x) = x$.) These points are shown in figure 2.

It turns out [Ott] that the invertible change of variables

$$x = h(y) = \sin^2\left(\frac{\pi y}{2}\right)$$

converts the sequence (y_n) , given by iterating the tent map,

$$y_{n+1} = T(y_n),$$

into the sequence (x_n) given by iterating the logistic map f_4 ,

$$x_{n+1} = f_4(x_n) = 4x_n(1-x_n).$$

Thus periodic points of the tent map, satisfying $T^m(y) = y$, are mapped by h into periodic points x of the logistic map, satisfying $T^m(x) = x$, shown in figure 3.

The following table (figure 4) gives the midpoint of the intervals containing the fixed points x such that $f_4^4(x) = x$ of the logistic map, using the Interval Newton method with standard double precision, and the corresponding exact values using the correspondence with the tent map, together with the difference. We see that the method indeed works very well. However, to find periodic points of higher period, higher precision must be used.

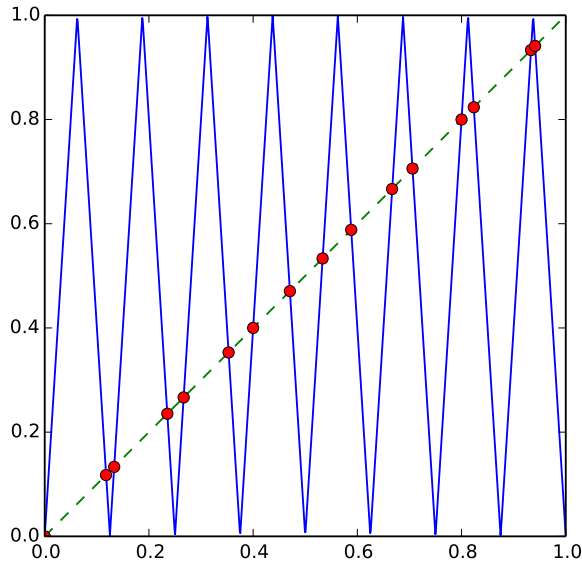


Fig. 2: Periodic points of the tent map with period dividing 4.

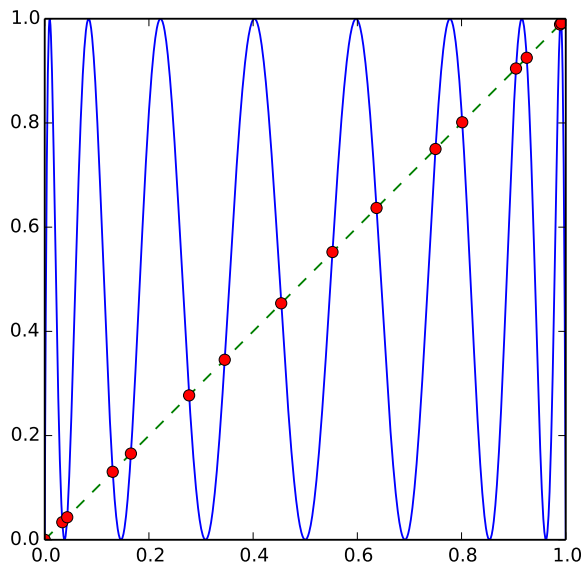


Fig. 3: Periodic points of the logistic map with period dividing 4.

Automatic differentiation

A difficulty in implementing the Newton method (even for the standard version), is the calculation of the derivative f' at a given point a . This may be accomplished for any function f by *automatic (or algorithmic) differentiation*, also easily implemented in Python.

The basic idea is that to calculate $f'(a)$, we may split a complicated function f up into its constituent parts and propagate the values of the functions and their derivatives through the calculations. For example, f may be the product and/or sum of

0.0000000000000000	0.0000000000000000	0.0000000000000000
0.0337638852978221	0.0337638852978221	-0.0000000000000000
0.0432272711786996	0.0432272711786995	0.0000000000000000
0.1304955413896703	0.1304955413896704	-0.0000000000000001
0.1654346968205710	0.1654346968205709	0.0000000000000001
0.2771308221117308	0.2771308221117308	0.0000000000000001
0.3454915028125262	0.3454915028125263	-0.0000000000000001
0.4538658202683487	0.4538658202683490	-0.0000000000000003
0.5522642316338270	0.5522642316338265	0.0000000000000004
0.6368314950360415	0.6368314950360414	0.0000000000000001
0.7500000000000000	0.7499999999999999	0.0000000000000001
0.8013173181896283	0.8013173181896283	0.0000000000000000
0.9045084971874738	0.9045084971874736	0.0000000000000002
0.9251085678648071	0.9251085678648070	0.0000000000000001
0.9890738003669028	0.9890738003669027	0.0000000000000001
0.9914865498419509	0.9914865498419507	0.0000000000000002

Fig. 4: Period 4 points: calculated, exact, and the difference.

simpler functions. To combine information on functions u and v , we use

$$\begin{aligned} (u+v)'(a) &= u'(a) + v'(a), \\ (uv)'(a) &= u'(a)v(a) + u(a)v'(a), \\ (g(u))'(a) &= g'(u(a))u'(a). \end{aligned}$$

Thus, for each function u , it is sufficient to represent it as an ordered pair $(u(a), u'(a))$ in order to calculate the value and derivative of a complicated function made out of combinations of such functions.

Constants C satisfy $C'(a) = 0$ for all a , so that they are represented as the pair $(C, 0)$. Finally, the identity function $\text{id} : x \mapsto x$ has derivative $\text{id}'(a) = 1$ at all a .

The mechanism of operator overloading in Python allows us to define an `AutoDiff` class. Calculating the derivative of a function $f(x)$ at the point a is then accomplished by calling `f(AutoDiff(a, 1))` and extracting the derivative part.

```
class AutoDiff(object):
    def __init__(self, value, deriv=None):

        if deriv is None:
            deriv = 0.0

        self.value = value
        self.deriv = deriv

    def __add__(self, other):
        if not isinstance(other, AutoDiff):
            other = AutoDiff(other)

        return AutoDiff(self.value+other.value,
                        self.deriv+other.deriv)

    def __mul__(self, other):
        if not isinstance(other, AutoDiff):
            other = AutoDiff(other)

        return AutoDiff(self.value*other.value,
                        self.value*other.deriv +
                        self.deriv*other.value)

    def __repr__(self):
        return "{}, {}".format(
            self.value, self.deriv)
```

As a simple example, let us differentiate the function $f(x) = x^2 + x + 2$ at $x = 3$. We define the function in the standard way:

```
def f(x):
    return x*x + x + 2
```

We now define a variable `a` where we wish to calculate the derivative and an object `x` representing the object that we will use in the automatic differentiation. Since it represents the function $x \rightarrow x$ evaluated at `a`, it has derivative 1:

```
a = 3
x = AutoDiff(a, 1)
```

Finally, we simply apply the standard Python function to this new object, and the automatic differentiation takes care of the rest:

```
result = f(x)
print("a={}; f(a)={}; f'(a)={}".format(
    a, result.value, result.deriv))
```

giving the result

```
a=3; f(a)=14; f'(a)=7.0
```

The derivative $f'(x) = 2x + 1$, so that $f(a = 3) = 14$ and $f'(a = 3) = 7$. Thus both the value of the function and its derivative have been calculated in a completely *automatic* way, by applying the rules encoded by the overloaded operators.

Simulating a chaotic billiard model

A dynamical system is said to be *chaotic* if it satisfies certain conditions [Deva03], of which a key one is *sensitive dependence on initial conditions*: two nearby initial conditions separate *exponentially* fast.

This leads to difficulties if we want precise answers on the long-term behaviour of such systems, for example simulating the solar system over millions of years [Lask13]. For certain types of systems, there are *shadowing theorems*, which say that an approximate trajectory calculated with floating point arithmetic, in which a small error is committed at each step, is close to a true trajectory [Palm09]; however, these results tend to be applicable only for rather restricted classes of systems which do not include those of physical interest.

Interval arithmetic provides a partial solution to this problem, since it automatically reports the number of significant figures in the result which are guaranteed correct. As an example, we show how to solve one of the well-known "Hundred-digit challenge problems" [Born04], which consists of calculating the position from the origin in a certain billiard problem.

Billiard problems are a class of mathematical models in which pointlike particles (i.e. particles with radius 0) collide with fixed obstacles. They can be used to study systems of hard discs or hard spheres with elastic collisions, and are also paradigmatic examples of systems which can be proved to be chaotic, since the seminal work of Sinai [Chern06].

Intuitively, when two nearby rays of light hit a circular mirror, the curvature of the surface leads to the rays separating after they reflect from the mirror. At each such collision, the distance in phase space between the rays is, on average, multiplied by a factor at each collision, leading to exponential separation and hence chaos, or *hyperbolicity*.

The trajectory of a single particle in such a system will hit a sequence of discs. However, a nearby initial condition may, after a few collisions, miss one of the discs hit by the first particle, and will then follow a completely different future trajectory. With standard floating-point arithmetic, there is no information about when this occurs; interval arithmetic can guarantee that this has *not* occurred, and thus that the sequence of discs hit is correct.

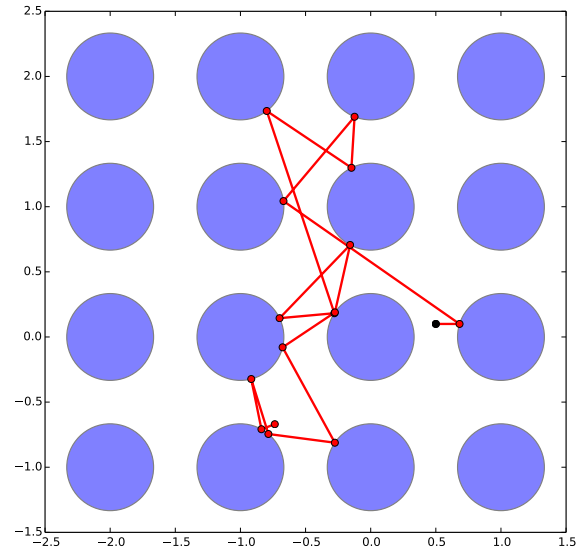


Fig. 5: Trajectory of the billiard model up to time 10; the black dot shows the initial position.

The second of the Hundred-digit challenge problems [Born04] is as follows:

A point particle bounces off fixed discs of radius $\frac{1}{3}$, placed at the points of a square lattice with unit distance between neighbouring points. The particle starts at $(x, y) = (0.5, 0.1)$, heading due east with unit speed, i.e. with initial velocity $(1, 0)$. Calculate the distance from the origin of the particle at time $t = 10$, with 10 correct significant figures.

To solve this, we use a standard implementation of the billiard by treating it as a single copy of a unit cell, centred at the origin and with side length 1, and periodic boundary conditions. We keep track of the cell that is reached in the corresponding "unfolded" version in the complete lattice.

The code used is a standard billiard code, that may be written in an *identical* way to use either standard floating-point method or interval arithmetic using `ValidiPy`, changing only the initial conditions to use intervals instead of floating-point variables. Since 0.1 and $1/3$ are not exactly representable, they are replaced by the smallest possible intervals containing the true values, using directed rounding as discussed above.

It turns out indeed to be necessary to use multiple precision in the calculation, due to the chaotic nature of the system. In fact, our algorithm requires a precision of at least 96 binary digits (compared to standard double precision of 53 binary digits) in order to guarantee that the correct trajectory is calculated up to time $t = 10$. With fewer digits than this, a moment is always reached at which the intervals have grown so large that it is not guaranteed whether a given disc is hit or not. The trajectory is shown in figure 5.

With 96 digits, the uncertainty on the final distance, i.e. the diameter of the corresponding interval, is 0.0788. As the number of digits is increased, the corresponding uncertainty decreases exponentially fast, reaching 4.7×10^{-18} with 150 digits, i.e. at least 16 decimal digits are guaranteed correct.

Extensions

Intervals in higher dimensions

The ideas and methods of interval arithmetic may also be applied in higher dimensions. There are several ways of defining intervals in 2 or more dimensions [Moo09]. Conceptually, the simplest is perhaps to take the Cartesian product of one-dimensional intervals:

$$I = [a, b] \times [c, d]$$

We can immediately define, for example, functions like $f(x, y) := x^2 + y^2$ and apply them to obtain the corresponding interval extension $\tilde{f}([a, b], [c, d]) := [a, b]^2 + [c, d]^2$, which will automatically contain the true image $f(I)$. Similarly, functions $f: \mathbb{R}^2 \rightarrow \mathbb{R}^2$ will give an interval extension producing a two-dimensional rectangular interval. However, the result is often much larger than the true image, so that the subdivision technique must be applied.

Taylor series

An extension of automatic differentiation is to manipulate Taylor series of functions around a point, so that the function u is represented in a neighbourhood of the point a by the tuple $(a, u'(a), u''(a), \dots, u^{(n)}(a))$. Recurrence formulas allow these to be manipulated relatively efficiently. These may be used, in particular, to implement arbitrary-precision solution of ordinary differential equations.

An implementation in Python is available in ValidiPy, while an implementation in the Julia is available separately, including Taylor series in multiple variables [TaylorSeries].

Conclusions

Interval arithmetic is a powerful tool which has been, perhaps, under-appreciated in the wider scientific community. Our contribution is aimed at making these techniques more widely known, in particular at including them in courses at masters', or even undergraduate, level, with working, freely available code in Python and Julia.

Acknowledgements

The authors thank Matthew Rocklin for helpful comments during the open refereeing process, which improved the exposition. Financial support is acknowledged from DGAPA-UNAM PAPIME grants PE-105911 and PE-107114, and DGAPA-UNAM PAPIIT grants IG-101113 and IN-117214. LB acknowledges support through a Cátedra Moshinsky (2013).

REFERENCES

- [IEEE754] *IEEE Standard for Floating-Point Arithmetic*, 2008, IEEE Std 754-2008.
- [Gold91] D. Goldberg (1991), What Every Computer Scientist Should Know About Floating-Point Arithmetic, *ACM Computing Surveys* **23** (1), 5-48.
- [High96] N.J. Higham (1996), *Accuracy and Stability of Numerical Algorithms*, SIAM.
- [Tuck11] W. Tucker (2011), *Validated Numerics: A Short Introduction to Rigorous Computations*, Princeton University Press.
- [Tuck99] W. Tucker, 1999, The Lorenz attractor exists, *C. R. Acad. Sci. Paris Sér. I Math.* **328** (12), 1197-1202.
- [ValidiPy] D.P. Sanders and L. Benet, ValidiPy package for Python, <<https://github.com/computo-fc/ValidiPy>>
- [ValidatedNumerics] D.P. Sanders and L. Benet, ValidatedNumerics.jl package for Julia, <<https://github.com/dpsanders/ValidatedNumerics.jl>>
- [gmpy2] GMPY2 package, <<https://code.google.com/p/gmpy/>>
- [MPFR] MPFR package, <<http://www.mpfr.org>>
- [Fous07] L. Fousse et al. (2007), MPFR: A multiple-precision binary floating-point library with correct rounding, *ACM Transactions on Mathematical Software* **33** (2), Art. 13.
- [Lor63] E.N. Lorenz (1963), Deterministic nonperiodic flow, *J. Atmos. Sci.* **20** (2), 130-141.
- [ChaosBook] P. Cvitanović et al. (2012), *Chaos: Classical and Quantum*, Niels Bohr Institute. <<http://ChaosBook.org>>
- [Ott] E. Ott (2002), *Chaos in Dynamical Systems*, 2nd edition, Cambridge University Press.
- [Deva03] R.L. Devaney (2003), *An Introduction to Chaotic Dynamical Systems*, Westview Press.
- [Lask13] J. Laskar (2013), Is the Solar System Stable?, in *Chaos: Poincaré Seminar 2010* (chapter 7), B. Duplantier, S. Nonnenmacher and V. Rivasseau (eds).
- [Palm09] K.J. Palmer (2009), Shadowing lemma for flows, *Scholarpedia* **4** (4). http://www.scholarpedia.org/article/Shadowing_lemma_for_flows
- [Born04] F. Bornemann, D. Laurie, S. Wagon and J. Waldvogel (2004), *The SIAM 100-Digit Challenge: A Study in High-Accuracy Numerical Computing*, SIAM.
- [Chem06] N. Chernov and R. Markarian (2006), *Chaotic Billiards*, AMS.
- [TaylorSeries] L. Benet and D.P. Sanders, TaylorSeries package, <<https://github.com/lbenet/TaylorSeries.jl>>
- [Moo09] R.E. Moore, R.B. Kearfott and M.J. Cloud (2009), *Introduction to Interval Analysis*, SIAM.
- [Lorenz] E.N. Lorenz (1963), Deterministic nonperiodic flow, *J. Atmos. Sci.* **20** (2), 130-148.

Creating a browser-based virtual computer lab for classroom instruction

Ramalingam Saravanan^{‡*}

<http://www.youtube.com/watch?v=LiZJMYxvJbQ>

Abstract—With laptops and tablets becoming more powerful and more ubiquitous in the classroom, traditional computer labs with rows of expensive desktop computers are slowly beginning to lose their relevance. An alternative approach for teaching Python is to use a browser-based virtual computer lab, with a notebook interface. The advantages of physical computer labs, such as face-to-face interaction, and the challenge of replicating them in a virtual environment are discussed. The need for collaborative features like terminal/notebook sharing and chatting is emphasized. A virtual computer lab is implemented using the GraphTerm server, with several experimental features including a virtual dashboard for monitoring tasks and progressively fillable notebooks for ensuring step-by-step completion of a sequence of tasks.

Index Terms—virtual computer lab, notebook interface, cloud computing, browser-based terminal

Introduction

A computer lab, with rows of identical desktop computers, is a commonly used resource when teaching programming or scientific computing [Thompson11]. However, with the increasing popularity of *Bring Your Own Device* solutions everywhere, computer labs are slowly losing their relevance. Physical labs are expensive to provision and maintain. Personal laptop computers and even tablets have more than sufficient computing horsepower for pedagogical use. As infrastructure costs increase, cloud-based virtual computing environments look increasingly attractive as replacements for physical computer labs.

As we inevitably, albeit slowly, move away from hardware computer labs, it is worth analyzing the pros and cons of the physical vs. the virtual approach. Some of the advantages of a physical lab are:

- Uniform software without installation or compatibility issues
- Ability to walk around and monitor students' progress
- Students raise their hand to request assistance from the instructor
- Students can view each other's screens and collaborate
- Large files and datasets can be shared through cross-mounted file systems

* Corresponding author: sarava@tamu.edu

‡ Texas A&M University

Copyright © 2014 Ramalingam Saravanan. This is an open-access article distributed under the terms of the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original author and source are credited.

Some of the shortcomings of physical computer labs are:

- Need to purchase and maintain hardware, ensuring security
- Need to create user accounts and install course-specific software
- Instructor may not want or may not have root access, leading to delays in fixing problems
- Students typically need to be physically present to use the lab

Many of the advantages of the physical computer lab are difficult to replicate when students use laptops in an *ad hoc* fashion, with differing software installations and without shared file systems or collaborative features. A browser-based virtual computing lab running on a remote server can address many of the shortcomings of physical computer labs, while still retaining the advantages of a uniform software environment and shared files. However, the human interaction aspects of a physical lab will never be fully reproducible in a virtual environment.

This study documents experiences gained from using hybrid physical-virtual computer lab in teaching an introductory programming course for meteorology undergraduates during Spring 2014. The course was aimed at students with no prior knowledge of programming. The goal was to teach them to write code that can access and visualize meteorological data, and Python is ideally suited for this task [Lin12]. The students had access to a physical lab with identical iMac computers, but several expressed an interest in using their laptops so that they could continue to work on assignments at home.

Students began using the IPython Notebook interface [Perez12] early on during the course. Some of them installed Enthought or Anaconda distributions on their laptop computers and used the bundled notebook server. They were also given the option of remotely accessing a browser-based virtual computer lab using GraphTerm, which is an open-source graphical terminal interface that is backwards compatible with the `xterm` terminal, and also supports a lightweight notebook interface [Saravanan13]. Some of the students used the remote GraphTerm option to work on their assignments and collaborate on their group project.

There are several "virtual computer lab" implementations on university campuses which typically use a Citrix server to provide remote desktop access to Windows computers. There are also many commercial products providing Python computing environments in cloud, such as PythonAnywhere and Wakari [Wakari]. This study focuses on alternative "roll your own" solutions using open-source software that are specifically targeted for use in

an interactive classroom instruction setting, with collaborative features that mimic physical computer labs. Creating such a virtual computing lab usually involves instantiating a server using a cloud infrastructure provider, such as Amazon. A new server can be set-up within minutes, with a scientific Python distribution automatically installed during set-up. Students can then login to their own accounts on the server using a browser-based interface to execute Python programs and visualize graphical output. Typically, each student would use a notebook interface to work on assignments.

The different approaches to providing a virtual computing environment for Python, and the associated challenges, are discussed. Options for providing a multi-user environment include running a public IPython Notebook server, or using alternative free/commercial solutions that incorporate the notebook interface. Enhancements to the notebook interface that promote step-by-step instruction are described, as are collaborative features that are important if the virtual environment is to retain some of the advantages a physical computer lab. User isolation and security issues that arise in a multi-user software environment are also considered.

Multi-user virtual computing environments for Python

The simplest approach to creating a shared environment for teaching Python would be to run a public IPython Notebook server [IPython]. At the moment, the server does not support a true multi-user environment, but multiple notebooks can be created and edited simultaneously. (Full multi-user support is planned in the near future.) The obvious disadvantage is that there is no user isolation, and all notebooks are owned by the same user.

One can get around the current single-user limitation by running multiple server processes, one for each student. This could be done simply by creating a separate account for each student on a remote server, or using more sophisticated user isolation approaches. One of the most promising solutions uses Docker, which is an emerging standard for managing Linux containers [Docker]. Unlike virtual machines, which work at the operating system level, lightweight Docker isolation works at the application level.

JiffyLab is an open source project that uses Docker to provide multi-user access to the IPython Notebook interface [JiffyLab]. It creates a separate environment for each user to run the notebook server. New accounts are created by entering an email address. JiffyLab addresses the user isolation issue, but does not currently provide collaborative features.

In the commercial world, Wakari is a cloud Python hosting solution from the providers of the Anaconda distribution, with a free entry-level account option [Wakari]. It supports browser-based terminal and editing capabilities, as well as access to IPython Notebooks. Wakari provides user isolation and the ability to share files and notebooks for collaboration.

Perhaps the most comprehensive free solution currently available for a shared virtual Python environment is the Sage Math Cloud (SMC) [Sage]. It provides support for command line terminals, LaTeX editing and includes numerous math-related programs such as R, Octave, and the IPython Notebook. SMC is being used for course instruction and now supports a real-time collaborative version of the IPython Notebook [Stein13].

This study describes an alternative open-source solution using GraphTerm that is derived from the terminal interface, with graphical and notebook interfaces that appear as an extension of

terminal [GraphTerm]. It includes all features of the `xterm`-based command-line interface (CLI) along with additional graphical user interface (GUI) options. In particular, users can use CLI editors like `vim` or Javascript-based graphical editors to modify programs. Inline `matplotlib` graphics is supported, rather like the Qt Console for IPython [QtConsole]. Multiple users can access the server simultaneously, with collaborative features such as being able to view each others' terminals. GraphTerm also implements a lightweight notebook interface that is compatible with the IPython Notebook interface.

A browser-based Python Integrated Development Environment (IDE) such as Wakari or SMC typically consists of the following components: a graphical file manager, a Javascript-based editor, a shell terminal, and a notebook window. A web GUI is used to bind these components together. GraphTerm also serves as an IDE, but it blurs some of the distinctions between the different components. For example, the same GraphTerm window may function at times like a plain `xterm`, a Qt Console with inline graphics, or a simplified IPython Notebook, depending upon the command being executed.

For the introductory programming course, a remote computer was set up to run the GraphTerm server, and students were able to automatically create individual accounts on it using a group access code. (*Appendices 1 and 2 provide details of the installation and remote access procedures involved in creating the virtual computing lab.*) Students used the virtual lab accounts to execute shell commands on the remote terminal, and also to use the notebook interface, either by using GraphTerm's own notebook implementation or by running the full IPython Notebook server on their account. (The distinction between GraphTerm and IPython notebooks will be explained later.) Having a custom, lightweight notebook interface enabled the implementation and testing of several experimental features to the GraphTerm server to support collaboration and a new feature called *progressively fillable* notebooks. This feature allows an instructor to assign a set of notebook-based tasks to students, where each task must be completed before the automatically displaying the correct solution for the task and proceeding to the next task, which may depend on the correct solutions to all the previous tasks.

Sharing terminal sessions

One of the common sights in a physical computer lab is a group of students huddled around a computer animatedly discussing something visible on the screen. It would be nice to reproduce this ability to view each other's terminals and communicate in the virtual computer lab. If students use their laptop computers in a regular classroom with row seating, rather than a lab, then collaborative features in the virtual setting could make a big difference. Such features would also allow the students to work with each other after hours. Another crucial feature of the physical computer lab is the instructor's ability to grab a student's mouse/keyboard to make some quick fixes to his/her code. This feature would very much be desirable to have in a virtual computer lab.

Although the default multi-user account setup in GraphTerm isolates users with Unix account permissions, the instructor can choose to enable terminal sharing for all, or create specific user groups for shared work on projects etc. As super user, the instructor has access to the students' terminals. (A list of all users currently watching a terminal session can be accessed from the menu.)

For the programming course, group-based sharing was enabled to allow students to work together on the end-of-semester project. Students were able to *watch* someone else's terminal, without controlling it, or *steal* control of someone else's terminal, if the terminal owner had permitted it. (To regain control, the terminal owner would have to steal it back.)

GraphTerm supports a rudimentary chat command for communication between all watchers for a terminal session. The command displays a *chat* button near the top right corner. Any user who is currently watching a terminal session can type lines of text that will be displayed as a feed, translucently overlaid on the terminal itself. When chatting, an *alert* button also becomes available to attract the attention of the terminal watchers (which may include the instructor).

There is also an experimental *tandem control* option, which allows two or more people to control a terminal simultaneously. This needs to be used with caution, because it can lead to unpredictable results due to the time lags between terminal operations by multiple users.

Notebook interface

The IPython Notebook interface was a huge hit with students in the most recent iteration of the programming course, as compared to the clunky text-editor/command-line/graphics-window development environment that was used in previous iterations. In addition to running the IPython Notebook server locally on the lab computers, students accessed the notebook interface on the remote server in two ways, depending upon individual preference:

1. Activating the lightweight notebook interface built into the remote GraphTerm terminal. This can be as simple as typing *Shift-Enter* after starting the standard command line Python interpreter.
2. Running the public IPython Notebook server on the remote computer and accessing it using a browser on the local computer. (A separate server process is started for each user who initiates it by typing a command, with a unique port number and a password that is the same as the user's access code.)

The two notebook implementations run separately, although they share the user's home directory.

The GraphTerm notebook interface is implemented as a wrapper on top of the standard Python command line interface. It provides basic notebook functionality, but is not a full-featured environment like IPython Notebook. It does support the same notebook format, which means that notebooks can be created in a GraphTerm window, saved as `.ipynb` files and opened later using IPython Notebook, and *vice versa*. Notebooks are opened within GraphTerm using the standard `python` (or `ipython`) command, and pre-loading the GraphTerm-compatible `pylab` environment (Fig. 1):

```
python -i $GTERM_DIR/bin/gpylab.py notebook.ipynb
```

A shortcut command, `gpython notebook.ipynb`, can also be used instead of the long command line shown above. Like the IPython Notebook, typing *Control-Enter* executes code in-place, and *Shift-Enter* executes code and moves to the next cell. The GraphTerm notebook interface is integrated into the terminal (Fig. 2), allowing seamless switching between the python command line and notebook mode, "live sharing" of notebooks across shared terminals, and inline graphics display that can work across SSH login boundaries [Saravanan13].

```
view terminal command notebook help share ubuntu@mmckeown/tty8:3 run
...
~$ python -i $GTERM_DIR/bin/gpylab.py
NOTE: Enabled interactive plotting mode, ion()
To disable, use ioff()
NOTE: Enabled notebook mode (affects auto printing of expressions)
To disable, use gterm.nbmode(False)
>>> gterm.open_notebook()
>>>

import netCDF4
import numpy as np
import matplotlib.pyplot as plt
import mpl_toolkits.basemap as bm
import pylab

yr = int(raw_input("Enter year: "))
mo = int(raw_input("Enter month: "))
dy = int(raw_input("Enter day: "))
tm = int(raw_input("Enter time: "))

def getdate(year, month, day, time):
    if year==2008 and 1<=month<=12 and 1<=day<=31 and time==0 or time==6:
        e = "%d" % (year)
        f = "%02d" % (month)
        g = "%02d" % (day)
        h = "%02d" % (time)
        return [e, f, g, h]
```

Fig. 1: Snippet showing a portion of a notebook session in the virtual lab.

```
view terminal command notebook help share ubuntu@mmckeown/tty8:2 run ste
plt.xlabel(n1, inline=3, fontsize=6)
n2 = pylab.contourf(lonproj, latproj, variable1_dict["variable_array"],
CBI = plt.colorbar(n2, orientation="horizontal", shrink=0.8)
pylab.title(title_name)

pylab.figure(figsize=(13,13))
pylab.subplot(221)
plotdata(gecht, vor, "500mb Geopotential Heights and Vorticity", 'spectral')
pylab.subplot(222)
plotdata(RH, vertvel, "700mb Relative Humidity and Vertical Velocities", 'G'
pylab.subplot(223)
plotdata(Temp, Pres, "Surface Temperature and Pressure", 'jet')
pylab.subplot(224)
plotdata(gecht2, wind, "300mb Geopotential Heights and Wind Speed", 'spectra

Enter year: 2008
Enter month: 3
Enter day: 27
Enter time: 0
```

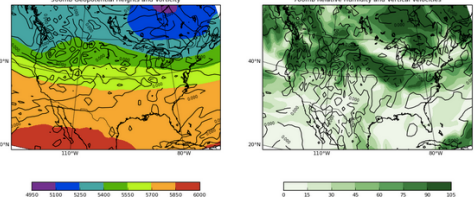


Fig. 2: Another snippet showing a notebook session in the virtual lab, with inline graphics.

GraphTerm Hosts

User: **ubuntu**

Available hosts:

1. [acassel](#)
2. [aharte](#)
3. [cgrimes](#)
4. [fpequeno49](#)
5. [hcrockett](#)
6. [jblake](#)
7. [jcoy](#)
8. [jespinoza](#)
9. [jgarcia](#)
10. [jklosterman](#)
11. [jklosterman3](#)
12. [jmccarthy](#)
13. [kszeliga](#)
14. [local](#)

Fig. 3: The instructor "dashboard" in the virtual computer lab, showing all currently logged in users. Clicking on the user name will open a list of terminals for that user.

```

view terminal command notebook help share ubuntu@local/tty1: 4
~$ gadmin -a sessions
acassel/projaccs- idle 21min
acassel/qui4 idle 17min
fpequeno49/aprill17:Untitled3#3
fpequeno49/tty1:Untitled4#1
fpequeno49/tty2 idle 3min
fpequeno49/tty3- idle 18min
fpequeno49/tty4- idle 18min
jgarcia/quiz:Untitled8#2 idle 2min
jgarcia/tty1:Untitled7#4- idle 24min
jmccarthy/tty1:Untitled3 idle 12min
jmccarthy/tty2:Untitled4#1 idle 1min
jmccarthy/tty3- idle 7min
jmccarthy/tty4:exercise6#2 idle 2min
mefries/quiz4a
mmckeown/quiz4:Untitled4#2
mmckeown/tty1:exercise7#1 idle 8min
myoung/tty1:Untitled14#3- idle 21min
myoung/tty2:Untitled15#1
myoung/tty3- idle 17min
myoung/tty4:ex7#1
sarava/tty1:Untitled9#8- idle 16min
sarava/tty2:Untitled10#1 idle 2min
~$ █

```

Fig. 4: The instructor "dashboard" in the virtual computer lab, with a listing of all user terminals, including notebook names and the last modified cell count, generated by the `gadmin` command. Clicking on the terminal session name will open a view of the terminal.

A dashboard for the lab

An important advantage of a physical computer lab is the ability to look around and get a feel for the overall level of student activity. The GraphTerm server keeps track of terminal activity in all the sessions (Fig. 3). The idle times of all the terminals can be viewed to see which users are actively using the terminal (Fig. 4). If a user is running a notebook session, the name of the notebook and the number of the last modified cell are also tracked. During the programming course, this was used to assess how much progress was being made during notebook-based assignments.

The `gadmin` command is used to list terminal activity, serving as a *dashboard*. Regular expressions can be used to filter the list of terminal sessions, restricting it to particular user names, notebook names, or alert status. As mentioned earlier, students have an *alert* button available when they enable the built-in chat feature. This alert button serves as the virtual equivalent of *raising a hand*, and can be used to attract the attention of the instructor by flagging the terminal name in `gadmin` output.

The terminal list displayed by `gadmin` is hyperlinked. As the super user has access to all terminals, clicking on the output of `gadmin` will open a specific terminal for monitoring (Fig. 5). Once a terminal is opened, the chat feature can be used to communicate with the user.

Progressively fillable notebooks

A common difficulty encountered by students on their first exposure to programming concepts is the inability to string together simple steps to accomplish a complex task. For example, they may grasp the concept of an `if` block and a `for` loop separately, but putting those constructs together turns out to be much harder. When assigned a multi-step task to perform, some of the students will get stuck on the first task and never make any progress. One can address this by progressively revealing the solutions to each step, and then moving on to the next step. However, if this is done

in a synchronous fashion for the whole lab, the stronger students will need to wait at each step for the weaker students to catch up.

An alternative approach is to automate this process to allow students make incremental progress. As the Notebook interface proved to be extremely popular with the students, an experimental *progressively fillable* version of notebooks was recently implemented in the GraphTerm server. A notebook code cell is assigned to each step of a multi-step task, with associated Markdown cells for explanatory text. Initially, only the first code cell is visible, and the remaining code cells are hidden. The code cell contains a "skeleton" program, with missing lines (Fig. 6). The expected textual or graphical output of the code is also shown. Students can enter the missing lines and repeatedly execute the code using *Control-Enter* to reproduce the expected results. If the code runs successfully, or if they are ready to give up, they press *Shift-Enter* to move on. The last version of the code executed by the student, whether right or wrong, is saved in the notebook (as Markdown), and the correct version of the code is then displayed in the cell and executed to produce the desired result (Fig. 7). The next code cell becomes visible and the whole process is repeated for the next step of the task.

The user interface for creating progressively fillable notebooks in this experimental version is very simple. The instructor creates a regular notebook, with each code cell corresponding to a specific step of a complex task. The comment string `## ANSWER` is appended to all code lines that are to be hidden (Fig. 7). The code in each successive step can depend on the previous step being completed correctly. Each code cell is executed in sequence to produce output for the step. The notebook is then saved with the suffix `-fill` appended to the base filename to indicate that it is fillable. The saving step creates new Markdown content from the output of each code cell to display the expected output in the progressive version of the notebook. Once filled by the students, the notebooks can be submitted for grading, as they contain a record of the last attempt at completing each step, even if unsuccessful.

One can think of progressively fillable notebooks as providing "training wheels" for the inexperienced programmer trying to juggle different algorithmic concepts at the same time. They can work on assignments that require getting several pieces of code right for the the whole program to work, without being stymied by a pesky error in a single piece. (This approach is also somewhat analogous to simple unit testing using the `doctest` Python module, which runs functions with specified input and compares the results to the expected output.)

Some shortcomings

Cost is an issue for virtual computer labs, because running a remote server using a cloud service vendor does not come free. For example, an AWS general purpose `m3.medium` server, which may be able to support 20 students, costs \$0.07 per hour, which works out to \$50 per month, if running full time. This would be much cheaper than the total cost of maintaining a lab with 20 computers, even if it can be used for 10 different courses simultaneously. However, this is a real upfront cost whereas the cost of computer labs is usually hidden in the institutional overheads. Of course, on-campus servers could be used to host the virtual computer labs, instead of commercial providers. Also, dedicated commercial servers may be considerably cheaper than cloud-based servers for sustained long-term use.

```

~$ gadmin -a sessions ".*quiz.*"
jgarcia/quiz:Untitled8#3
mefries/quiz4a:Untitled7#1
mmckeown/quiz4:Untitled4#3
~$ gadmin -t -a sessions ".*quiz.*"
jgarcia/quiz
mefries/quiz4a
mmckeown/quiz4
~$ gframe -r 300 -b -c 3 -t /jgarcia/quiz /mefries/quiz4a /mmckeown/quiz4

```

Standard Atmosphere Plot

Altitude

Pressure

```

def vertsection(month,
lon_index):
    m = int(month)
    l = int(lon_index)

```

```

defined
>>> month_data = month[:]
Traceback (most recent call
last):
  File "<stdin>", line 1, in
<module>
NameError: name 'month' is
not defined
>>> print type(month_data)
Traceback (most recent call
last):
  File "<stdin>", line 1, in
<module>
NameError: name 'month_data'
is not defined
>>>

```

vertsection(2,11)

10000

8000

6000

4000

2000

0

50 0 -50

-60 -40 -20 0 20 40

>>>

Fig. 5: The instructor "dashboard" in the virtual computer lab, with embedded views of student terminals generated using the `gframe` command.

view terminal command notebook help share local/tty1: 1 run steal new detach

NB: Sample-fill.py

Part I

Write a function `abs_add` that returns the sum of the absolute values of two numbers and another function `abs_sub` that computes the difference of absolute values. Test the two functions.

```

# Part 1a: Define the function abs_add
##... (fill in code here)

# Testing function abs_add
print abs_add(3, -4)

# Part 1b: Define the function abs_sub
##... (fill in code here)

# Testing function abs_sub
print abs_sub(3, -4)
>>>

```

Expected output:

```

7
-1

```

Fig. 6: View of progressively fillable notebook before user completes Step 1. Note two comment line where it says (fill in code here). The user can replace these lines with code and execute it. The resulting output should be compared to the expected output, shown below the code cell.

Depending upon whether the remote server is located on campus or off campus, a good internet connection may be essential for the performance a virtual computer lab during work hours. For a small number of students, server capacity should not be an issue, because classroom assignments are rarely compute-intensive. For large class sizes, more expensive servers may be needed.

When compared to using a physical computer lab, typically managed by professional system administrators, instructors planning to set up their own virtual computer lab would need some minimal command line skills. The GraphTerm server runs only on Linux/Mac systems, as it requires access to the Unix terminal interface. (The browser-based GraphTerm client can be used on Windows computers, as well as iPads and Android tablets.)

GraphTerm supports a basic notebook interface that is closely

view terminal command notebook help share local/tty1: 1 run steal new detach

NB: Sample-fill.py

Your Input:

```

# Part 1a: Define the function abs_add
def abs_add(a,b):
    return a+b

# Testing function abs_add
print abs_add(3, -4)

# Part 1b: Define the function abs_sub
def abs_sub(a,b):
    return a+b

# Testing function abs_sub
print abs_sub(3, -4)

```

Your Output:

```

-1
-1

```

Expected Input:

```

# Part 1a: Define the function abs_add
def abs_add(a, b):
    return abs(a) + abs(b) ## ANSWER

# Testing function abs_add
print abs_add(3, -4)

# Part 1b: Define the function abs_sub
def abs_sub(a, b):
    return abs(a) - abs(b) ## ANSWER

# Testing function abs_sub
print abs_sub(3, -4)

7
-1
>>>

```

Fig. 7: View of progressively fillable notebook after user has completed Step 1. The last version of code entered and executed by the user is included the markup, and the code cell now displays the "correct" version of the code. Note the comment suffix `## ANSWER` on selected lines of code. These lines were hidden in the unfilled view.

integrated with the command line, and supports the collaborative/administrative features of the virtual computer lab. However, this interface will never be as full-featured as the IPython Notebook interface, which is a more comprehensive and mature product. For this reason, the virtual computer lab also provides the ability for users who need more advanced notebook features

to run their own IPython Notebook server and access it remotely. The compatibility of the `.ipynb` notebook file format and the shared user directory should make it fairly easy to switch between the two interfaces.

Although the notebook interface has been a boon for teaching students, it is not without its disadvantages. It has led to decreased awareness of the file and directory structure, as compared to the traditional command line interface. For example, as students download data, they often have no idea where the files are being saved. The concept of a modular project spread across functions in multiple files also becomes more difficult to grasp in the context of a sequential notebook interface. The all-inclusive `pylab` environment, although very convenient, can lead to reduced awareness of the modular nature of Python packages.

Conclusions

Students would like to break free of the physical limitations of a computer lab, and to be able to work on their assignments anywhere, anytime. However, the human interactions in a physical computer lab have considerable pedagogical value, and any virtual environment would need to support collaborative features to make up for that. With further development of the IPython Notebook, and other projects like SMC, one can expect to see increased support for collaboration through browser-based graphical interfaces.

The collaborative features of the GraphTerm server enable it to be used as a virtual computer lab, with automatic user creation, password-less authentication, and terminal sharing features. Developing a GUI for the complex set of tasks involved in managing a virtual lab can be daunting. Administering the lab using just command line applications would also be tedious, as some actions like viewing other users' terminals are inherently graphical operations. The hybrid CLI-GUI approach of GraphTerm gets around this problem by using a couple of tricks to implement the virtual "dashboard":

- (i) Commands that produce hyperlinked (clickable) listings, to easily select terminals for opening etc.
- (ii) A single GraphTerm window can embed multiple nested GraphTerm terminals for viewing

The IPython Notebook interface, with its blending of explanatory text, code, and graphics, has evolved into a powerful tool for teaching Python as well as other courses involving computation and data analysis. The notebook format can provide the "scaffolding" for structured instruction [[AeroPython](#)]. One of the dilemmas encountered when using notebooks for interactive assignments is when and how to reveal the answers. Progressively fillable notebooks address this issue by extending the notebook interface to support assignments where students are required to complete tasks in a sequential fashion, while being able to view the correct solutions to completed tasks immediately.

Appendix 1: GraphTerm server setup

The GraphTerm server is implemented purely in Python, with HTML+Javascript for the browser. Its only dependency is the Tornado web server. GraphTerm can be installed using the following shell command:

```
sudo pip install graphterm
```

To start up a multi-user server on a Linux/Mac computer, a variation of the following command may be executed (as root):

Fig. 8: Automatic form display for the `ec2launch` command, used to configure and launch a new virtual lab using the AWS cloud. The form elements are automatically generated from the command line options for `ec2launch`

```
gtermserver --daemon=start --auth_type=multiuser
--user_setup=manual --users_dir=/home
--port=80 --host=server_domain_or_ip
```

If a physical server is not readily available for multi-user access, a virtual server can be created on demand using Amazon Web Services (AWS). The GraphTerm distribution includes the convenience scripts `ec2launch`, `ec2list`, `ec2scp`, and `ec2ssh` to launch and monitor AWS Elastic Computing Cloud (EC2) instances running a GraphTerm server. (An AWS account is required to use these scripts, and the `botocore` Python module needs to be installed.)

To launch a GraphTerm server in the cloud using AWS, first start up the single-user version of GraphTerm:

```
gtermserver --terminal --auth_type=none
```

The above command should automatically open up a GraphTerm window in your browser. You can also open one using the URL <http://localhost:8900> Within the GraphTerm window, run the following command to create a virtual machine on AWS:

```
ec2launch
```

The above command will display a web form within the GraphTerm window (Fig. 8). This is an example of the hybrid CLI-GUI interface supported by GraphTerm that avoids having to develop a new web GUI for each additional task. Filling out the form and submitting it will automatically generate and execute a command line which looks like:

```
ec2launch --type=m3.medium --key_name=ec2key
--ami=ami-2f8f9246 --gmail_addr=user@gmail.com
--auth_type=multiuser --pylab --netcdf testlab
```



```
ec2list
Public DNS
ec2-54-197-139-116.compute-1.amazonaws.com
ec2-54-204-51-129.compute-1.amazonaws.com
ec2-54-198-135-149.compute-1.amazonaws.com
Key
ec2key
ec2key
ec2key
Tags
test48.gterm.net
test49.gterm.net
lab.gterm.net
State
shutting-down
running
running
Action
Kill
Kill
Kill
```

Fig. 9: Output of the `ec2list` command, listing currently active AWS cloud instances running the virtual computer lab. Clickable links are displayed for terminating each instance

GraphTerm Login

Please specify username (letters/digits/hyphens, starting with letter).
If new user, enter your group code to create account.

User:
Code: OR

Fig. 10: Login page for GraphTerm server in multiuser mode. The user needs to enter the group access code, and may choose to use Google Authentication

The above command can be saved, modified, and re-used as needed. After the new AWS Linux server has launched and completed configuration, which can take several minutes, its IP address and domain name will be displayed. The following command can then be used to list, access or terminate all running cloud instances associated with your AWS account (Fig. 9):

```
ec2list
```

Detailed instructions for accessing the newly launched server are provided on the GraphTerm website [GraphTerm].

Appendix 2: Multiple user authentication and remote access

Assuring network security is a real headache for *roll your own* approaches to creating multi-user servers. Institutional or commercial support is essential for keeping passwords secure and software patched. Often, the only sensitive information in a remotely-accessed academic computer lab account is the student's password, which may be the same as one used for a more confidential account. It is therefore best to avoid passwords altogether for virtual computer labs, and remove a big burden of responsibility from the instructor.

The GraphTerm server uses two approaches for password-less authentication: (i) A randomly-generated user access code, or (ii) Google authentication. The secret user access code is stored in a protected file on the students' local computers and a hash-digest scheme is used for authentication without actually transmitting the secret code. Students create an account using a browser URL provided by the instructor, selecting a new user name and entering a group access code (Fig. 10). A new Unix account is created for each user and the user-specific access code is displayed (Fig. 11). Instead of using this access code, students can choose to use password-less Google Authentication.

After logging in, users connect to an existing terminal session or create a new terminal session. A specific name can be used for a new terminal session, or the special name `new` can be used to automatically choose names like `tty1`, `tty2` etc. When sharing terminals with others, it is often useful to choose a meaningful name for the terminal session.

Users can detach from a terminal session any time and connect to it at a later time, without losing any state information. For example, a terminal created at work can be later accessed from

GraphTerm Hosts

User: `jsmith`

Created new user with authentication code:
74ac-7a60-6760-6ec7
Copy this information for future use or [email it to yourself](#).
[Click here](#) to open a terminal.

Optionally, you can enter an email address below. It will be

E-mail:

[Sign out](#)

Fig. 11: New user welcome page, with access code displayed.

home, without interrupting program execution. The students found the ability to access their terminal sessions from anywhere to be perhaps the most desirable feature of the virtual computer lab.

REFERENCES

- [AeroPython] *AeroPython* <http://lorenabarba.com/blog/announcing-aeropython/>
- [Docker] *Docker* sandboxed linux containers <http://www.docker.com/whatisdocker/>
- [GraphTerm] *GraphTerm* home page <http://code.mindmeldr.com/graphterm>
- [IPython] *IPython* Notebook public server http://ipython.org/ipython-doc/stable/notebook/public_server.html
- [JiffyLab] *JiffyLab* multiuser IPython notebooks <https://github.com/ptone/jiffylab>
- [Lin12] J. Lin. *A Hands-On Introduction to Using Python in the Atmospheric and Oceanic Sciences* [Chapter 9, Exercise 29, p. 162] <http://www.johnny-lin.com/pyintro>
- [Perez12] F. Perez. *The IPython notebook: a historical retrospective*. Jan 2012 <http://blog.fperez.org/2012/01/ipython-notebook-historical.html>
- [QtConsole] *A Qt Console for IPython*. <http://ipython.org/ipython-doc/2/interactive/qtconsole.html>
- [Sage] *Sage Math Cloud* <https://cloud.sagemath.com/>
- [Saravanan13] R. Saravanan. *GraphTerm: A notebook-like graphical terminal interface for collaboration and inline data visualization*, Proceedings of the 12th Python in Science Conference, 90-94, July 2013. <http://conference.scipy.org/proceedings/scipy2013/pdfs/saravanan.pdf>
- [Stein13] W. Stein. *IPython Notebooks in the Cloud with Realtime Synchronization and Support for Collaborators*. Sep 2013 <http://sagemath.blogspot.com/2013/09/ipython-notebooks-in-cloud-with.html>
- [Thompson11] A. Thompson. *The Perfect Educational Computer Lab*. Nov 2011 <http://blogs.msdn.com/b/alfredth/archive/2011/11/30/the-perfect-educational-computer-lab.aspx>
- [Wakari] *Wakari* collaborative data analytics platform <http://continuum.io/wakari>

TracPy: Wrapping the Fortran Lagrangian trajectory model TRACMASS

Kristen M. Thyng^{‡*}, Robert D. Hetland[‡]

<https://www.youtube.com/watch?v=8poLWacun50>

Abstract—Numerical Lagrangian trajectory modeling is a natural method of investigating transport in a circulation system and understanding the physics on the wide range of length scales that are actually experienced by a drifter. A previously-developed tool, TRACMASS, written in Fortran, accomplishes this modeling with a clever algorithm that operates natively on the commonly used staggered Arakawa C grid. TracPy is a Python wrapper written to ease running batches of simulations. Some improvements in TracPy include updating to netCDF4-CLASSIC from netCDF3 for saving drifter trajectories, providing an iPython notebook as a usermanual for using the system, and adding unit tests for stable continued development.

Index Terms—Lagrangian tracking, numerical drifters, Python wrapper

Introduction

Drifters are used in oceanography and atmospheric *in situ* in order to demonstrate flow patterns created by individual fluid parcels. For example, in the ocean, drifters will often be released on the sea surface, and allowed to be passively transported with the flow, reporting their location via GPS at regular intervals. In this way, drifters are gathering data in a Lagrangian perspective. For example, [LaCasce2003] analyzes a set of over 700 surface drifters released in the northern Gulf of Mexico, using the tracks to better understand the dynamics of the underlying circulation fields.

Lagrangian trajectory modeling is a method of moving parcels through a fluid based on numerically modeled circulation fields. This approach enables analysis of many different drifter experiments for a much lower cost than is required to gather one relatively small set of drifters. Additionally, the inherent limits to the number of drifters that can reasonably be used *in situ* can lead to biased statistics [LaCasce2008]. In one study, numerical drifters were used to understand where radio-nuclides from a storage facility would travel if accidentally released [Döös2007]. Drifters are also used in on-going response work by the Office of Response and Restoration in the National Oceanic and Atmospheric Administration (NOAA). Using model output made available by various groups, responders apply their tool (General NOAA Oil Modeling Environment, GNOME) to simulate drifters and get

best estimates of predicted oil transport [Beegle-Krause1999], [Beegle-Krause2001].

Numerical drifters may be calculated online, while a circulation model is running, in order to use the highest resolution model-predicted velocity fields available in time (on the order of seconds to minutes). However, due to the high costs of the hydrodynamic computation, many repeated online simulations is not usually practical. In this case, Lagrangian trajectories can also be calculated offline, using the velocity fields at the stored temporal resolution (on the order of minutes to hours).

There are many sources of error in simulating offline Lagrangian trajectories. For example, the underlying circulation model must be capturing the dynamics to be investigated, and model output must be available often enough to represent the simulated flow conditions accurately. On top of that, the Lagrangian trajectory model must properly reproduce the transport pathways of the system. A given drifter's trajectory is calculated using velocity fields with a spatial resolution determined by the numerical model grid. To move the drifter, the velocity fields must be available at the drifter's location, which in general will not be co-located with all necessary velocity information. Many Lagrangian trajectory models use low- or high-order interpolation in space to extend the velocity information to the drifter location. The algorithm discussed in this work has a somewhat different approach.

TRACMASS is a Lagrangian trajectory model that runs natively on velocity fields that have been calculated on a staggered Arakawa C grid. Originally written about 2 decades ago, it has been used in many applications (*e.g.*, [Döös2007]). The core algorithm for TRACMASS is written in Fortran for speed, and has been wrapped in Python for increased usability. This code package together is called TracPy [Thyng2014b].

TRACMASS

The TRACMASS algorithm for stepping numerical drifters in space is distinct from many algorithms because it runs natively on a staggered Arakawa C grid, *i.e.*, it uses the velocity fields at the grid locations at which they are calculated. This grid is used in ocean modeling codes, including ROMS, MITgcm, and HyCOM. In the staggered Arakawa C grid, the west-east or zonal velocity, u , is located at the west and east walls of a grid cell; the north-south or meridional velocity, v , is located at the north and south walls; and the vertical velocity, w , is located at the vertically top and bottom cell walls (Figure 1). Note that the algorithm

* Corresponding author: kthyng@tamu.edu

‡ Texas A&M University

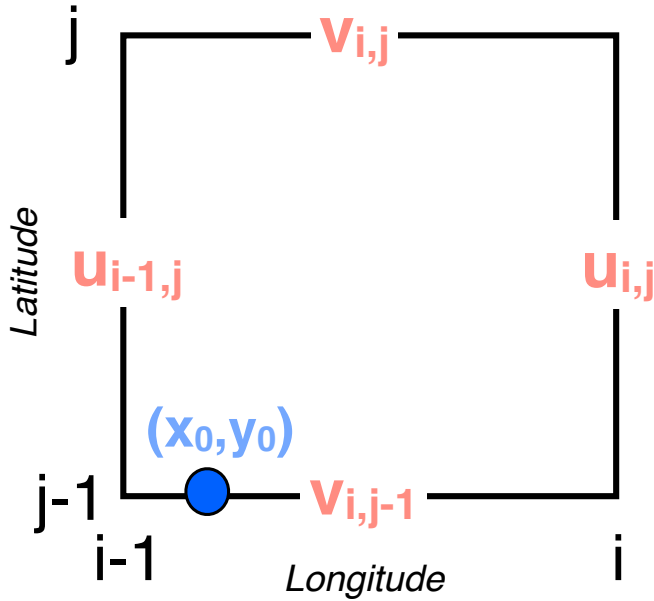


Fig. 1: A single rectangular grid cell is shown in the x - y plane. Zonal (meridional) u (v) velocities are calculated at the east/west (north/south) cell walls. In the vertical direction, w velocities are calculated at the top and bottom cell walls. After [Döös2013].

is calculated using fluxes through grid cell walls instead of the velocities themselves to account for differences in cell wall size due to a curvilinear horizontal grid or a σ coordinate vertical grid. The drifter is stepped as follows:

- 1) To calculate the time required for the drifter to exit the grid cell in the x direction:
 - a. Linearly interpolate the velocity across the cell in the zonal direction to find $u(x)$.
 - b. Solve the ordinary differential equation $u(x) = \frac{dx}{dt}$ for $x(t)$.
 - c. Back out the time required to exit the grid cell in the zonal direction, t_x .
- 2) Follow the same methodology in the meridional and vertical directions to find t_y and t_z .
- 3) The minimum time t_{min} ; the minimum of t_x, t_y, t_z ; is when the drifter would first exit the grid cell
- 4) The subsequent (x, y, z) position for the drifter is calculated using t_{min} .

This process occurs for each drifter each time it is moved forward from one grid cell edge to the next. If a drifter will not reach a grid cell edge, it stops in the grid cell. Calculations for the drifter trajectories are done in grid index space so that the grid is rectangular, which introduces a number of simplifications. The velocity fields are linearly interpolated in time for each subsequent stepping of each drifter. Because a drifter is moved according to its distinct time and location, each drifter is stepped separately, and the time step between each reinterpolation can be different. The location of all drifters is sampled at regular intervals between the available circulation model outputs for consistency. Because reading in the circulation model output is one of the more time-consuming parts of the process, all drifters are stepped between

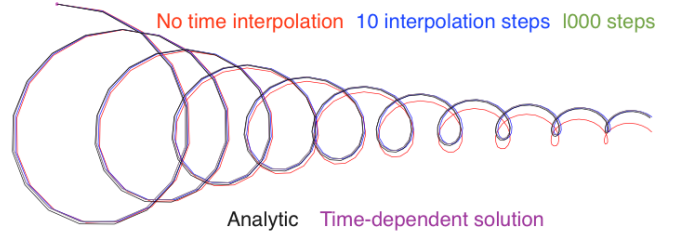


Fig. 2: A trajectory from a damped inertial oscillation is shown from several simulated cases with the analytic solution. Cases shown are trajectories calculated using TRACMASS with zero [red], 10 [blue], and 1000 [green] time interpolation steps between model outputs; the analytic solution [black]; and the time-dependent algorithm [purple]. The green, black, and purple curves are indistinguishable. From [Döös2013].

the velocity fields at two consecutive times, then the velocity fields from the next output time are read in to continue stepping.

Drifters can be stepped forward or backward in time; this is accomplished essentially by multiplying the velocity fields by -1 . Because of the analytic nature of the TRACMASS algorithm, the trajectories found forward and backward in time are the same.

Time is assumed to be steady while a drifter is being stepped through a grid cell—how much this will affect the resulting trajectory depends on the size of the grid cell relative to the speed of the drifter. When a drifter reaches another grid cell wall, the fields are re-interpolated. The user may choose to interpolate the velocity fields at shorter intervals if desired by setting a maximum time before reinterpolation. A time-dependent algorithm has been developed to extend the TRACMASS algorithm [DeVries2001], but previous researchers have found that the steady approximation is adequate in many cases [Döös2013] and it is not implemented in TracPy.

The capability of the TRACMASS algorithm has been demonstrated by creating synthetic model output, running numerical drifters, and comparing with known trajectory solutions (Figure 2). A damped inertial oscillation is used in the test, for which the analytic solutions for both the velocity fields and a particle's trajectory are known [Döös2013]. Cases of a drifter trajectory calculated with different levels of interpolation between model outputs are shown along with the analytic solution and a trajectory calculated using the time-dependent TRACMASS algorithm. All trajectories generally following the analytic solution, but the case with no time interpolation of the fields clearly deviates. The case with 10 interpolation steps in times performs well, and with 1000 interpolation steps, the curves are indistinguishable. Note that in this test case, the size of the grid cell relative to the motion of the trajectory emphasizes the effect of time interpolation.

Options are available to complement the basic algorithm of TRACMASS. For example, it can be important to consider whether or not to add additional explicit subgrid diffusion to drifters. Energy at scales below a few spatial grid cells is not included in an ocean circulation model except through a turbulence closure scheme or other means. This energy is included in the numerical scheme and implemented in the simulation, and in this regard is implicitly included in the saved velocity fields from the circulation model. From this perspective, adding any additional subgrid energy is duplicating the energy that is already included in the simulation. However, without including some small-scale energy to drifter tracks, drifters starting at the same

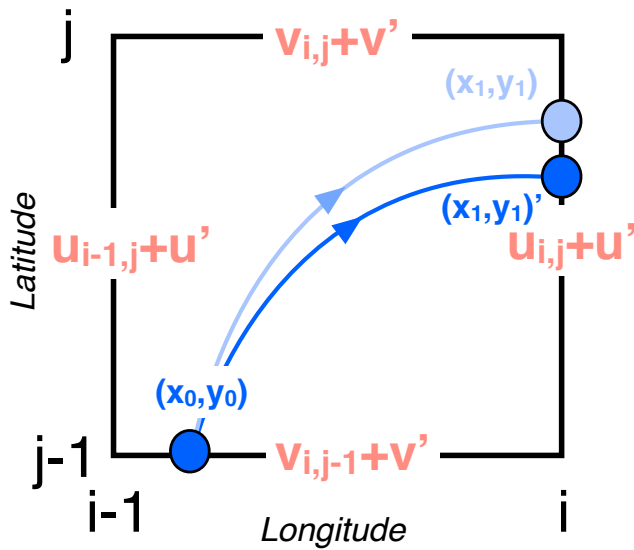


Fig. 3: Instead of being stepped forward to new location (x_1, y_1) by the base velocity field, a drifter can be instead stepped forward by the velocity field plus a random velocity fluctuation to include explicit subgrid diffusion, such that the drifter ends up instead at $(x_1, y_1)'$. After [Döös2013].

time and location will follow the same path, which is clearly not realistic—adding a small amount of energy to drifter tracks acts to stir drifters in a way that often looks more realistic than when explicit subgrid diffusion is not included. This added energy will also affect Lagrangian metrics that are calculated from drifter trajectories (e.g., [Döös2011]).

To address this issue, there are several optional means of including explicit subgrid diffusion in TRACMASS, all of which are low order schemes [LaCase2008]. Drifter trajectories may be stepped using not the basic velocity fields (u, v) but with the velocity fields plus some small random velocity fluctuation (u', v') (Figure 3). Alternatively, drifter trajectory locations can be given an added random walk—randomly moved a small distance away from their location each step within a circle whose radius is controlled by an input parameter (Figure 4). Note that when using additional subgrid diffusion, drifter tracks will not be the same forward and backward in time.

TracPy

The goal of TracPy is to take advantage of the speed and ingenuity of the TRACMASS algorithm, written in Fortran, but have access to the niceties of Python and for quickly and simply setting up and running batches of simulations. Being a scientific research code, TRACMASS has been developed by different researchers and with specific research purposes in mind, such that the complexity of the code grew over time. TracPy was written to include the important basic, computationally burdensome elements of calculating drifter trajectories from TRACMASS, and do the rest in Python.

TracPy uses a class for a given simulation of drifters. The TracPy class is initialized with all necessary parameters for the simulation itself, e.g., number of days to run the simulation, parameter for maximum time before reinterpolation between available circulation model outputs, whether to use subgrid diffusion, and whether to run in 2D or 3D. The class has methods for reading

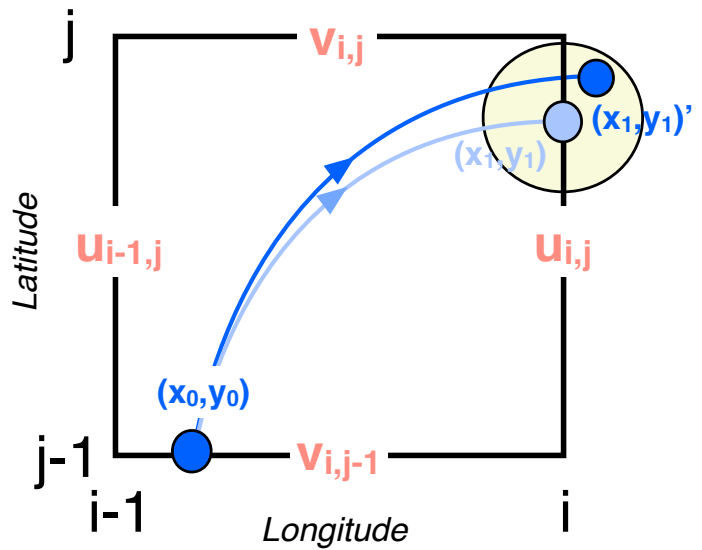


Fig. 4: A drifter's location can be randomly pushed within a circle from its calculated position to add explicit subgrid diffusion. After [Döös2013].

in the numerical grid, preparing for the simulation, preparing for each model step (e.g., reading in the velocity fields at the next time step), stepping the drifters forward between the two time steps of velocity fields stored in memory, wrapping up the time step, and wrapping up the simulation. Utilities are provided in TracPy for necessary computations, such as moving between grid spaces of the drifter locations. That is, drifter locations may, in general, be given in geographic space (i.e., longitude/latitude) or in projected space (e.g., universal traverse mercator or Lambert conformal conic), and positions are converted between the two using Python packages Basemap or Pyproj. Additionally, drifter locations will need to be transformed between grid index space, which is used in TRACMASS, and real space. Plotting functions and common calculations are also included in the suite of code making up TracPy.

Other improvements in the code system:

- Global variables have been removed in moving from the original set of TRACMASS code to the leaner TRACMASS algorithm that exists in TracPy, and have been replaced with variables that are passed directly between functions as needed.
- A user manual has been implemented in an iPython [notebook](#).
- A few simple test cases have been provided for users to experiment with and as a set of unit tests to improve stability during code development.

The parallelization of an offline Lagrangian trajectory model could be relatively straight-forward. Each drifter trajectory in any given simulation is independent of every other drifter. However, one of the slowest parts of drifter tracking is often reading in the velocity fields—separating out drifter trajectory calculations into different processes would most likely increase the input/output requirement. Still, an easy way to take advantage of the drifter calculations being inherently decoupled is to run different simulations on different processes. Many times, drifter simulations

	netCDF3	netCDF4C	% decrease
Simulation run time [s]	1038	1038	0
File save time [s]	3527	131	96
File size [GB]	3.6	2.1	42

TABLE 1: Comparisons between simulations run with netCDF3_64BIT and netCDF4-CLASSIC.

are run in large sets to gather meaningful statistics, in which case these separate simulations can all be distributed to different processes—as opposed to subdividing individual simulations to calculate different trajectories on different processes.

Drifter tracks are saved in netCDF files. The file format was recently changed from netCDF3 to netCDF4-CLASSIC. This change was made because netCDF4-CLASSIC combines many of the good parts of netCDF3 (e.g., file aggregation along a dimension) with some of the abilities of netCDF4 (compression). It does not allow for multiple unlimited dimensions (available in netCDF4), but that capability has not been necessary in this application. Changing to netCDF4-CLASSIC sped up the saving process, which had been slow with netCDF3 when a large number of drifters was used. The 64 bit format is used for saving the tracks for lossless compression of information.

We ran a two-dimensional test with about 270,000 surface drifters and over 100,000 grid cells for 30 days. A NaN is stored once a drifter exits the domain and forever after in time for that drifter (i.e., drifters do not reenter the numerical domain). This results in a large amount of output (much of which may contain NaNs), and saving such a large file can be really slow using netCDF3. Run time and space requirement results comparing simulations run with netCDF3 and netCDF4-CLASSIC show improved results with netCDF4-CLASSIC (Table 1). The simulation run time does not include time for saving the tracks, which is listed separately. The simulation run time was the same regardless of the file format used (since it only comes in when saving the file afterward), but the file save time was massively reduced by using netCDF4-CLASSIC (about 96%). Additionally, the file size was reduced by about 42%. Note that the file size is the same between netCDF4 and netCDF4-CLASSIC (not shown).

Suites of simulations were run using TracPy to test its time performance on both a Linux workstation (Figure 5) and a Macintosh laptop (not shown, but similar results). Changing the number of grid cells in a simulation (keeping the number of drifters constant at a moderate value) most affects the amount of time required to prepare the simulation, which is when the grid is read in. The grid will not be changing size in typical use cases so it may not be a significant problem, but the rapid increase in time required to run the code with an increasing number of grid cells may indicate an opportunity for improvement in the way the simulations are prepared. However, the time required to read in the grid increases exponentially with number of grid cells due to the increase in memory requirement for the grid arrays, so a change in approach to what information is necessary to have on hand for a simulation may be the only way to improve this. Changing the number of drifters (keeping the number of grid cells constant at a moderate value) affects the timing of several parts of the simulation. The base time spent preparing the simulation is mostly consistent since the grid size does not change between the cases. The time for stepping the drifters with TRACMASS, and processing after

stepping drifters and at the end of the simulation increase with an increasing number of drifters, as would be expected. The time required for increasing the number of drifters should scale linearly. Files used to run these tests are available on [GitHub](#).

The TracPy suite of code has been used to investigate several research problems so far. In one study, we sought to understand the effect of the temporal resolution of the circulation model output on the resulting drifter tracks (Figure 6). In another study, we initialized drifters uniformly throughout a numerical domain of the northwestern Gulf of Mexico and used the resulting tracks to examine the connectivity of water across the shelf break and the connectivity of surrounding waters with parts of the coastline (see e.g., Figure 7). Drifters have also been initialized at the inputs of the Mississippi and Atchafalaya rivers and tracked to illustrate the complex pathways of the fresh water (Figure 8).

Many improvements and extensions could be made to TracPy. It is intended to be integrated into NOAA’s GNOME oil tracking system in order to contribute another mover to their tracking system and take advantage of utilities in GNOME that are not in the TRACMASS algorithm, such as the ability to directly apply windage (this can be important for modeling material that directly feels wind stress, such as large oil slicks). Potential improvements include:

- The way the grid is read in and stored is taking too much time, as was seen in the TracPy performance tests.
- Placeholders for all locations for all drifters are currently stored for the entirety of a simulation run, which inflates the memory required for a simulation. Instead, drifter locations could be only temporarily stored and appended to the output file as calculated.
- A drifter location is set to NaN when the drifter exits the domain. This is currently somewhat accounted for by using netCDF4-CLASSIC compression. However, another way to minimize unnecessary NaN storage would be to alter how drifter tracks are stored. Instead of the current approach of storing tracks in a two-dimensional array of drifter versus location in time, all drifter locations for a given time step could be stored together on the same row. This makes retrieval more difficult and requires ragged rows, but eliminates the need to store a drifter that is inactive. Alternatively, a sparse matrix could be used to only store active drifters.
- Storage could be updated to full netCDF4 format.
- The modularity of the TracPy class should be improved.

Conclusions

A Python wrapper, TracPy, to a Lagrangian trajectory model, TRACMASS, combines the speed of the Fortran core algorithm with the ease of using Python. TracPy uses netCDF4-CLASSIC for saving trajectory paths, which is an improvement over netCDF3 in both time required to save the file and disk space required for the file. It also includes several improvements such as including an iPython notebook user manual and eliminating the use of global variables. TracPy performance tests indicate expected behavior in simulation time increase when increasing the number of drifters being simulated. However, when increasing the number of grid cells in the underlying numerical circulation model, preparing for the run takes more additional time than it probably should. The TracPy suite of code has been used for

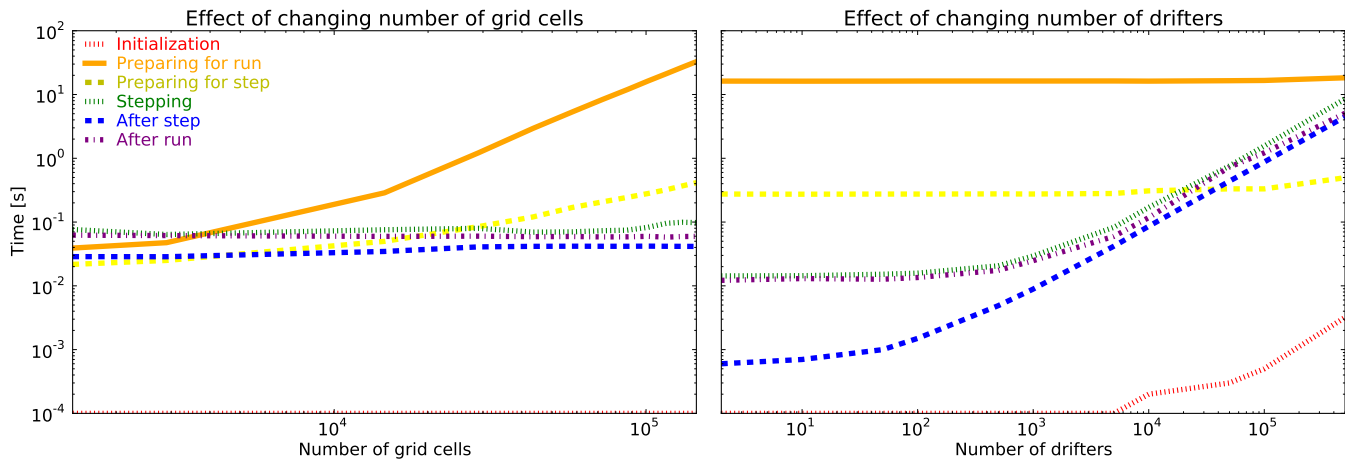


Fig. 5: Time required to run simulations with different numbers of grid cells (left) and drifters (right). A moderate number of drifters (5000) (left) and grid cells (100,000) (right) were used as the independent variable in the tests. For timing, the code is split into initialization [red], preparing for the run [orange], preparing for the model steps [yellow], stepping the drifters with TRACMASS [green], processing after the steps [blue], and processing at the end of the simulation [purple].

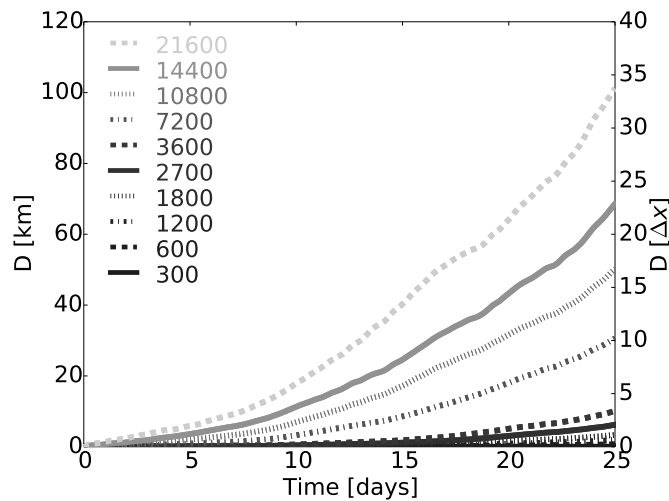


Fig. 6: Separation distance between pairs of drifters run with circulation model velocity fields output at different temporal resolutions (given in seconds), averaged over many pairs of drifters. From [Thyng2014a].

several applications so far, with more in the works for the future, along with continual code improvements.

Acknowledgements

Thanks to Chris Barker for help in improving TracPy modularity and unit tests, and for on-going work in integrating TracPy into NOAA's GNOME system. Thanks also to helpful review comments from Terry Letsche.

REFERENCES

- [Barker2000] C. H. Barker & J. A. Galt. *Analysis of methods used in spill response planning: Trajectory Analysis Planner TAP II*. Spill Science & Technology Bulletin, 6(2), 145-152, 2000.
- [Beegle-Krause1999] C. J. Beegle-Krause. *GNOME: NOAA's next-generation spill trajectory model*, Oceans '99 MTS/IEEE Proceedings. MTS/IEEE Conference Committee, Escondido, CA, vol. 3, pp. 1262-1266, 1999.

- [Beegle-Krause2001] C. J. Beegle-Krause. *General NOAA oil modeling environment (GNOME): a new spill trajectory model*, IOOSC 2001 Proceedings, Tampa, FL, March 26-29, 2001. Mira Digital Publishing, Inc., St. Louis, MO, vol. 2, pp. 865-871, 2001.
- [DeVries2001] P. de Vries, K. Döös. *Calculating Lagrangian trajectories using time-dependent velocity fields*, J Atmos Ocean Technol 18:1092-1101, 2001.
- [Döös2007] K. Döös, & A. Engqvist. *Assessment of water exchange between a discharge region and the open sea—A comparison of different methodological concepts*. Estuarine, Coastal and Shelf Science, 74(4), 709-721, 2007.
- [Döös2011] K. Döös, V. Rupolo, & L. Brodeau. *Dispersion of surface drifters and model-simulated trajectories*. Ocean Modelling, 39(3), 301-310, 2011.
- [Döös2013] K. Döös, J. Kjellsson, & B. Jönsson. *TRACMASS—A Lagrangian trajectory model*. In Preventive Methods for Coastal Protection (pp. 225-249). Springer International Publishing, 2013.
- [LaCasce2003] J. H. LaCasce & C. Ohlmann. *Relative dispersion at the surface of the Gulf of Mexico*, Journal of Marine Research, 61(3), 285-312, 2003.
- [LaCasce2008] J. H. LaCasce. *Statistics from Lagrangian observations*, Progress in Oceanography, 77(1), 1-29, 2008.
- [Thyng2014a] K. M. Thyng, R. D. Hetland, R. Montuoro, J. Kurian. *Lagrangian tracking errors due to temporal subsampling of numerical model output*. Submitted to Journal of Atmospheric and Oceanic Technology, 2014.
- [Thyng2014b] K. M. Thyng. TracPy. ZENODO. doi: 10.5281/zenodo.10433, 2014.

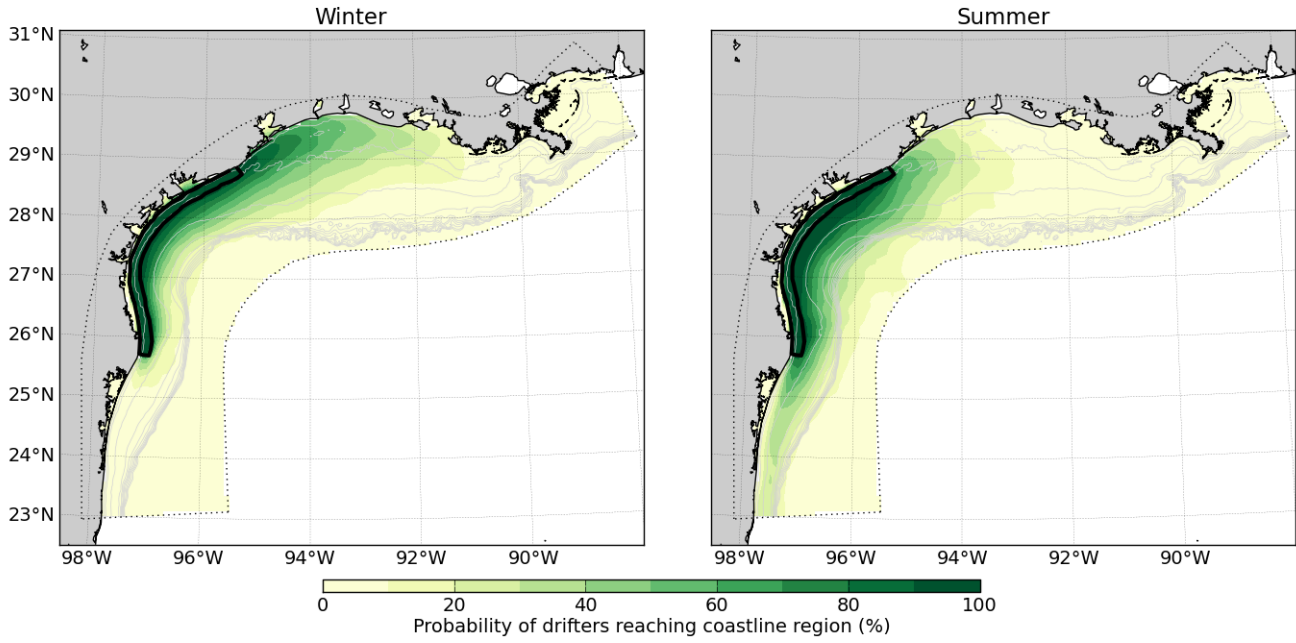


Fig. 7: Connectivity of waters with the southern Texas coastline over a 30 day time period, for the winter and summer months. Averaged over the years 2004-2010. Project [available on GitHub](#).

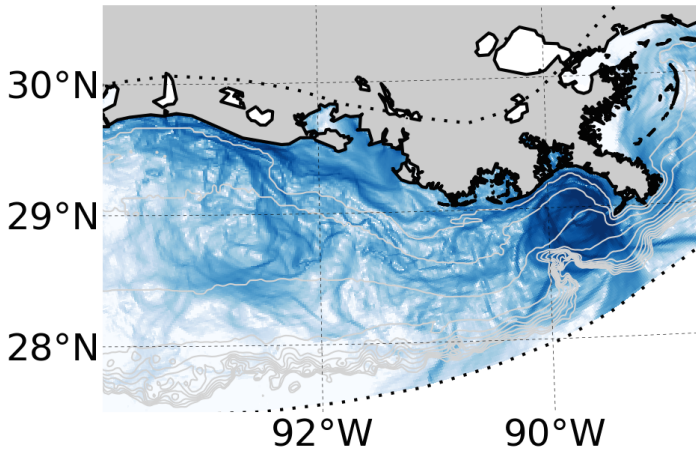


Fig. 8: Integrated pathways of drifters initialized in the Atchafalaya and Mississippi river inputs to the numerical domain.

Frequentism and Bayesianism: A Python-driven Primer

Jake VanderPlas^{‡*}

<http://www.youtube.com/watch?v=KhAUfqhLakw>



Abstract—This paper presents a brief, semi-technical comparison of the essential features of the frequentist and Bayesian approaches to statistical inference, with several illustrative examples implemented in Python. The differences between frequentism and Bayesianism fundamentally stem from differing definitions of probability, a philosophical divide which leads to distinct approaches to the solution of statistical problems as well as contrasting ways of asking and answering questions about unknown parameters. After an example-driven discussion of these differences, we briefly compare several leading Python statistical packages which implement frequentist inference using classical methods and Bayesian inference using Markov Chain Monte Carlo.¹

Index Terms—statistics, frequentism, Bayesian inference

Introduction

One of the first things a scientist in a data-intensive field hears about statistics is that there are two different approaches: frequentism and Bayesianism. Despite their importance, many researchers never have opportunity to learn the distinctions between them and the different practical approaches that result.

This paper seeks to synthesize the philosophical and pragmatic aspects of this debate, so that scientists who use these approaches might be better prepared to understand the tools available to them. Along the way we will explore the fundamental philosophical disagreement between frequentism and Bayesianism, explore the practical aspects of how this disagreement affects data analysis, and discuss the ways that these practices may affect the interpretation of scientific results.

This paper is written for scientists who have picked up some statistical knowledge along the way, but who may not fully appreciate the philosophical differences between frequentist and Bayesian approaches and the effect these differences have on both the computation and interpretation of statistical results. Because this passing statistics knowledge generally leans toward frequentist principles, this paper will go into more depth on the details of Bayesian rather than frequentist approaches. Still, it is not meant to be a full introduction to either class of methods. In particular, concepts such as the likelihood are assumed rather than

derived, and many advanced Bayesian and frequentist diagnostic tests are left out in favor of illustrating the most fundamental aspects of the approaches. For a more complete treatment, see, e.g. [Wasserman2004] or [Gelman2004].

The Disagreement: The Definition of Probability

Fundamentally, the disagreement between frequentists and Bayesians concerns the definition of probability.

For frequentists, probability only has meaning in terms of **a limiting case of repeated measurements**. That is, if an astronomer measures the photon flux F from a given non-variable star, then measures it again, then again, and so on, each time the result will be slightly different due to the statistical error of the measuring device. In the limit of many measurements, the *frequency* of any given value indicates the probability of measuring that value. For frequentists, **probabilities are fundamentally related to frequencies of events**. This means, for example, that in a strict frequentist view, it is meaningless to talk about the probability of the *true* flux of the star: the true flux is, by definition, a single fixed value, and to talk about an extended frequency distribution for a fixed value is nonsense.

For Bayesians, the concept of probability is extended to cover **degrees of certainty about statements**. A Bayesian might claim to know the flux F of a star with some probability $P(F)$: that probability can certainly be estimated from frequencies in the limit of a large number of repeated experiments, but this is not fundamental. The probability is a statement of the researcher's knowledge of what the true flux is. For Bayesians, **probabilities are fundamentally related to their own knowledge about an event**. This means, for example, that in a Bayesian view, we can meaningfully talk about the probability that the *true* flux of a star lies in a given range. That probability codifies our knowledge of the value based on prior information and available data.

The surprising thing is that this arguably subtle difference in philosophy can lead, in practice, to vastly different approaches to the statistical analysis of data. Below we will explore a few examples chosen to illustrate the differences in approach, along with associated Python code to demonstrate the practical aspects of the frequentist and Bayesian approaches.

A Simple Example: Photon Flux Measurements

First we will compare the frequentist and Bayesian approaches to the solution of an extremely simple problem. Imagine that we point a telescope to the sky, and observe the light coming from

* Corresponding author: jakevdp@cs.washington.edu

‡ eScience Institute, University of Washington

Copyright © 2014 Jake VanderPlas. This is an open-access article distributed under the terms of the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original author and source are credited.

1. This paper draws heavily from content originally published in a series of posts on the author's blog, [Pythonic Perambulations](#) [VanderPlas2014].

a single star. For simplicity, we will assume that the star's true photon flux is constant with time, i.e. that it has a fixed value F ; we will also ignore effects like sky background systematic errors. We will assume that a series of N measurements are performed, where the i^{th} measurement reports the observed flux F_i and error e_i .² The question is, given this set of measurements $D = \{F_i, e_i\}$, what is our best estimate of the true flux F ?

First we will use Python to generate some toy data to demonstrate the two approaches to the problem. We will draw 50 samples F_i with a mean of 1000 (in arbitrary units) and a (known) error e_i :

```
>>> np.random.seed(2) # for reproducibility
>>> e = np.random.normal(30, 3, 50)
>>> F = np.random.normal(1000, e)
```

In this toy example we already know the true flux F , but the question is this: given our measurements and errors, what is our best point estimate of the true flux? Let's look at a frequentist and a Bayesian approach to solving this.

Frequentist Approach to Flux Measurement

We will start with the classical frequentist maximum likelihood approach. Given a single observation $D_i = (F_i, e_i)$, we can compute the probability distribution of the measurement given the true flux F given our assumption of Gaussian errors:

$$P(D_i|F) = (2\pi e_i^2)^{-1/2} \exp\left(\frac{-(F_i - F)^2}{2e_i^2}\right).$$

This should be read "the probability of D_i given F equals ...". You should recognize this as a normal distribution with mean F and standard deviation e_i . We construct the *likelihood* by computing the product of the probabilities for each data point:

$$\mathcal{L}(D|F) = \prod_{i=1}^N P(D_i|F)$$

Here $D = \{D_i\}$ represents the entire set of measurements. For reasons of both analytic simplicity and numerical accuracy, it is often more convenient to instead consider the log-likelihood; combining the previous two equations gives

$$\log \mathcal{L}(D|F) = -\frac{1}{2} \sum_{i=1}^N \left[\log(2\pi e_i^2) + \frac{(F_i - F)^2}{e_i^2} \right].$$

We would like to determine the value of F which maximizes the likelihood. For this simple problem, the maximization can be computed analytically (e.g. by setting $d \log \mathcal{L} / dF|_{\hat{F}} = 0$), which results in the following point estimate of F :

$$\hat{F} = \frac{\sum w_i F_i}{\sum w_i}; \quad w_i = 1/e_i^2$$

The result is a simple weighted mean of the observed values. Notice that in the case of equal errors e_i , the weights cancel and \hat{F} is simply the mean of the observed data.

We can go further and ask what the uncertainty of our estimate is. One way this can be accomplished in the frequentist approach is to construct a Gaussian approximation to the peak likelihood; in this simple case the fit can be solved analytically to give:

$$\sigma_{\hat{F}} = \left(\sum_{i=1}^N w_i \right)^{-1/2}$$

2. We will make the reasonable assumption of normally-distributed measurement errors. In a Frequentist perspective, e_i is the standard deviation of the results of the single measurement event in the limit of (imaginary) repetitions of *that event*. In the Bayesian perspective, e_i describes the probability distribution which quantifies our knowledge of F given the measured value F_i .

This result can be evaluated this in Python as follows:

```
>>> w = 1. / e ** 2
>>> F_hat = np.sum(w * F) / np.sum(w)
>>> sigma_F = w.sum() ** -0.5
```

For our particular data, the result is $\hat{F} = 999 \pm 4$.

Bayesian Approach to Flux Measurement

The Bayesian approach, as you might expect, begins and ends with probabilities. The fundamental result of interest is our knowledge of the parameters in question, codified by the probability $P(F|D)$. To compute this result, we next apply Bayes' theorem, a fundamental law of probability:

$$P(F|D) = \frac{P(D|F) P(F)}{P(D)}$$

Though Bayes' theorem is where Bayesians get their name, it is important to note that it is not this theorem itself that is controversial, but the Bayesian *interpretation of probability* implied by the term $P(F|D)$. While the above formulation makes sense given the Bayesian view of probability, the setup is fundamentally contrary to the frequentist philosophy, which says that probabilities have no meaning for fixed model parameters like F . In the Bayesian conception of probability, however, this poses no problem.

Let's take a look at each of the terms in this expression:

- $P(F|D)$: The **posterior**, which is the probability of the model parameters given the data.
- $P(D|F)$: The **likelihood**, which is proportional to the $\mathcal{L}(D|F)$ used in the frequentist approach.
- $P(F)$: The **model prior**, which encodes what we knew about the model before considering the data D .
- $P(D)$: The **model evidence**, which in practice amounts to simply a normalization term.

If we set the prior $P(F) \propto 1$ (a *flat prior*), we find

$$P(F|D) \propto \mathcal{L}(D|F).$$

That is, with a flat prior on F , the Bayesian posterior is maximized at precisely the same value as the frequentist result! So despite the philosophical differences, we see that the Bayesian and frequentist point estimates are equivalent for this simple problem.

You might notice that we glossed over one important piece here: the prior, $P(F)$, which we assumed to be flat.³ The prior allows inclusion of other information into the computation, which becomes very useful in cases where multiple measurement strategies are being combined to constrain a single model (as is the case in, e.g. cosmological parameter estimation). The necessity to specify a prior, however, is one of the more controversial pieces of Bayesian analysis.

A frequentist will point out that the prior is problematic when no true prior information is available. Though it might seem straightforward to use an **uninformative prior** like the flat prior mentioned above, there are some surprising subtleties involved.⁴ It turns out that in many situations, a truly uninformative prior cannot exist! Frequentists point out that the subjective choice of a prior which necessarily biases the result should have no place in scientific data analysis.

3. A flat prior is an example of an improper prior: that is, it cannot be normalized. In practice, we can remedy this by imposing some bounds on possible values: say, $0 < F < F_{tot}$, the total flux of all sources in the sky. As this normalization term also appears in the denominator of Bayes' theorem, it does not affect the posterior.

A Bayesian would counter that frequentism doesn't solve this problem, but simply skirts the question. Frequentism can often be viewed as simply a special case of the Bayesian approach for some (implicit) choice of the prior: a Bayesian would say that it's better to make this implicit choice explicit, even if the choice might include some subjectivity. Furthermore, as we will see below, the question frequentism answers is not always the question the researcher wants to ask.

Where The Results Diverge

In the simple example above, the frequentist and Bayesian approaches give basically the same result. In light of this, arguments over the use of a prior and the philosophy of probability may seem frivolous. However, while it is easy to show that the two approaches are often equivalent for simple problems, it is also true that they can diverge greatly in other situations. In practice, this divergence most often makes itself most clear in two different ways:

- 1) The handling of nuisance parameters: i.e. parameters which affect the final result, but are not otherwise of interest.
- 2) The different handling of uncertainty: for example, the subtle (and often overlooked) difference between frequentist confidence intervals and Bayesian credible regions.

We will discuss examples of these below.

Nuisance Parameters: Bayes' Billiards Game

We will start by discussing the first point: nuisance parameters. A nuisance parameter is any quantity whose value is not directly relevant to the goal of an analysis, but is nevertheless required to determine the result which is of interest. For example, we might have a situation similar to the flux measurement above, but in which the errors e_i are unknown. One potential approach is to treat these errors as nuisance parameters.

Here we consider an example of nuisance parameters borrowed from [Eddy2004] that, in one form or another, dates all the way back to the posthumously-published 1763 paper written by Thomas Bayes himself [Bayes1763]. The setting is a gambling game in which Alice and Bob bet on the outcome of a process they can't directly observe.

Alice and Bob enter a room. Behind a curtain there is a billiard table, which they cannot see. Their friend Carol rolls a ball down the table, and marks where it lands. Once this mark is in place, Carol begins rolling new balls down the table. If the ball lands to the left of the mark, Alice gets a point; if it lands to the right of the mark, Bob gets a point. We can assume that Carol's rolls are unbiased: that is, the balls have an equal chance of ending up anywhere on the table. The first person to reach six points wins the game.

Here the location of the mark (determined by the first roll) can be considered a nuisance parameter: it is unknown and not of immediate interest, but it clearly must be accounted for when predicting the outcome of subsequent rolls. If this first roll settles

far to the right, then subsequent rolls will favor Alice. If it settles far to the left, Bob will be favored instead.

Given this setup, we seek to answer this question: *In a particular game, after eight rolls, Alice has five points and Bob has three points. What is the probability that Bob will get six points and win the game?*

Intuitively, we realize that because Alice received five of the eight points, the marker placement likely favors her. Given that she has three opportunities to get a sixth point before Bob can win, she seems to have clinched it. But quantitatively speaking, what is the probability that Bob will persist to win?

A Naïve Frequentist Approach

Someone following a classical frequentist approach might reason as follows:

To determine the result, we need to estimate the location of the marker. We will quantify this marker placement as a probability p that any given roll lands in Alice's favor. Because five balls out of eight fell on Alice's side of the marker, we compute the maximum likelihood estimate of p , given by:

$$\hat{p} = 5/8,$$

a result follows in a straightforward manner from the binomial likelihood. Assuming this maximum likelihood probability, we can compute the probability that Bob will win, which requires him to get a point in each of the next three rolls. This is given by:

$$P(B) = (1 - \hat{p})^3$$

Thus, we find that the probability of Bob winning is 0.053, or odds against Bob winning of 18 to 1.

A Bayesian Approach

A Bayesian approach to this problem involves *marginalizing* (i.e. integrating) over the unknown p so that, assuming the prior is accurate, our result is agnostic to its actual value. In this vein, we will consider the following quantities:

- B = Bob Wins
- D = observed data, i.e. $D = (n_A, n_B) = (5, 3)$
- p = unknown probability that a ball lands on Alice's side during the current game

We want to compute $P(B|D)$; that is, the probability that Bob wins given the observation that Alice currently has five points to Bob's three. A Bayesian would recognize that this expression is a *marginal probability* which can be computed by integrating over the joint distribution $P(B, p|D)$:

$$P(B|D) \equiv \int_{-\infty}^{\infty} P(B, p|D)dp$$

This identity follows from the definition of conditional probability, and the law of total probability: that is, it is a fundamental consequence of probability axioms and will always be true. Even a frequentist would recognize this; they would simply disagree with the interpretation of $P(p)$ as being a measure of uncertainty of knowledge of the parameter p .

To compute this result, we will manipulate the above expression for $P(B|D)$ until we can express it in terms of other quantities that we can compute.

We start by applying the definition of conditional probability to expand the term $P(B, p|D)$:

$$P(B|D) = \int P(B|p, D)P(p|D)dp$$

4. The flat prior in this case can be motivated by maximum entropy; see, e.g. [Jeffreys1946]. Still, the use of uninformative priors like this often raises eyebrows among frequentists: there are good arguments that even "uninformative" priors can add information; see e.g. [Evans2002].

Next we use Bayes' rule to rewrite $P(p|D)$:

$$P(B|D) = \int P(B|p, D) \frac{P(D|p)P(p)}{P(D)} dp$$

Finally, using the same probability identity we started with, we can expand $P(D)$ in the denominator to find:

$$P(B|D) = \frac{\int P(B|p, D)P(D|p)P(p)dp}{\int P(D|p)P(p)dp}$$

Now the desired probability is expressed in terms of three quantities that we can compute:

- $P(B|p, D)$: This term is proportional to the frequentist likelihood we used above. In words: given a marker placement p and Alice's 5 wins to Bob's 3, what is the probability that Bob will go on to six wins? Bob needs three wins in a row, i.e. $P(B|p, D) = (1-p)^3$.
- $P(D|p)$: this is another easy-to-compute term. In words: given a probability p , what is the likelihood of exactly 5 positive outcomes out of eight trials? The answer comes from the Binomial distribution: $P(D|p) \propto p^5(1-p)^3$
- $P(p)$: this is our prior on the probability p . By the problem definition, we can assume that p is evenly drawn between 0 and 1. That is, $P(p) \propto 1$ for $0 \leq p \leq 1$.

Putting this all together and simplifying gives

$$P(B|D) = \frac{\int_0^1 (1-p)^6 p^5 dp}{\int_0^1 (1-p)^3 p^5 dp}$$

These integrals are instances of the beta function, so we can quickly evaluate the result using scipy:

```
>>> from scipy.special import beta
>>> P_B_D = beta(6+1, 5+1) / beta(3+1, 5+1)
```

This gives $P(B|D) = 0.091$, or odds of 10 to 1 against Bob winning.

Discussion

The Bayesian approach gives odds of 10 to 1 against Bob, while the naive frequentist approach gives odds of 18 to 1 against Bob. So which one is correct?

For a simple problem like this, we can answer this question empirically by simulating a large number of games and count the fraction of suitable games which Bob goes on to win. This can be coded in a couple dozen lines of Python (see part II of [VanderPlas2014]). The result of such a simulation confirms the Bayesian result: 10 to 1 against Bob winning.

So what is the takeaway: is frequentism wrong? Not necessarily: in this case, the incorrect result is more a matter of the approach being "naïve" than it being "frequentist". The approach above does not consider how p may vary. There exist frequentist methods that can address this by, e.g. applying a transformation and conditioning of the data to isolate dependence on p , or by performing a Bayesian-like integral over the sampling distribution of the frequentist estimator \hat{p} .

Another potential frequentist response is that the question itself is posed in a way that does not lend itself to the classical, frequentist approach. A frequentist might instead hope to give the answer in terms of null tests or confidence intervals: that is, they might devise a procedure to construct limits which would provably bound the correct answer in $100 \times (1 - \alpha)$ percent of similar trials, for some value of α – say, 0.05. We will discuss the meaning of such confidence intervals below.

There is one clear common point of these two frequentist responses: both require some degree of effort and/or special expertise in classical methods; perhaps a suitable frequentist approach would be immediately obvious to an expert statistician, but is not particularly obvious to a statistical lay-person. In this sense, it could be argued that for a problem such as this (i.e. with a well-motivated prior), Bayesianism provides a more natural framework for handling nuisance parameters: by simple algebraic manipulation of a few well-known axioms of probability interpreted in a Bayesian sense, we straightforwardly arrive at the correct answer without need for other special statistical expertise.

Confidence vs. Credibility: Jaynes' Truncated Exponential

A second major consequence of the philosophical difference between frequentism and Bayesianism is in the handling of uncertainty, exemplified by the standard tools of each method: frequentist confidence intervals (CIs) and Bayesian credible regions (CRs). Despite their apparent similarity, the two approaches are fundamentally different. Both are statements of probability, but the probability refers to different aspects of the computed bounds. For example, when constructing a standard 95% bound about a parameter θ :

- A Bayesian would say: "Given our observed data, there is a 95% probability that the true value of θ lies within the credible region".
- A frequentist would say: "If this experiment is repeated many times, in 95% of these cases the computed confidence interval will contain the true θ ."⁵

Notice the subtle difference: the Bayesian makes a statement of probability about the *parameter value* given a *fixed credible region*. The frequentist makes a statement of probability about the *confidence interval itself* given a *fixed parameter value*. This distinction follows straightforwardly from the definition of probability discussed above: the Bayesian probability is a statement of degree of knowledge about a parameter; the frequentist probability is a statement of long-term limiting frequency of quantities (such as the CI) derived from the data.

This difference must necessarily affect our interpretation of results. For example, it is common in scientific literature to see it claimed that it is 95% certain that an unknown parameter lies within a given 95% CI, but this is not the case! This is erroneously applying the Bayesian interpretation to a frequentist construction. This frequentist oversight can perhaps be forgiven, as under most circumstances (such as the simple flux measurement example above), the Bayesian CR and frequentist CI will more-or-less overlap. But, as we will see below, this overlap cannot always be assumed, especially in the case of non-Gaussian distributions constrained by few data points. As a result, this common misinterpretation of the frequentist CI can lead to dangerously erroneous conclusions.

To demonstrate a situation in which the frequentist confidence interval and the Bayesian credibility region do not overlap, let us turn to an example given by E.T. Jaynes, a 20th century physicist who wrote extensively on statistical inference. In his words, consider a device that

5. [Wasserman2004], however, notes on p. 92 that we need not consider repetitions of the same experiment; it's sufficient to consider repetitions of any correctly-performed frequentist procedure.

"...will operate without failure for a time θ because of a protective chemical inhibitor injected into it; but at time θ the supply of the chemical is exhausted, and failures then commence, following the exponential failure law. It is not feasible to observe the depletion of this inhibitor directly; one can observe only the resulting failures. From data on actual failure times, estimate the time θ of guaranteed safe operation..." [Jaynes1976]

Essentially, we have data D drawn from the model:

$$P(x|\theta) = \begin{cases} \exp(\theta - x) & , x > \theta \\ 0 & , x < \theta \end{cases}$$

where $p(x|\theta)$ gives the probability of failure at time x , given an inhibitor which lasts for a time θ . We observe some failure times, say $D = \{10, 12, 15\}$, and ask for 95% uncertainty bounds on the value of θ .

First, let's think about what common-sense would tell us. Given the model, an event can only happen after a time θ . Turning this around tells us that the upper-bound for θ must be $\min(D)$. So, for our particular data, we would immediately write $\theta \leq 10$. With this in mind, let's explore how a frequentist and a Bayesian approach compare to this observation.

Truncated Exponential: A Frequentist Approach

In the frequentist paradigm, we'd like to compute a confidence interval on the value of θ . We might start by observing that the population mean is given by

$$E(x) = \int_0^\infty xp(x)dx = \theta + 1.$$

So, using the sample mean as the point estimate of $E(x)$, we have an unbiased estimator for θ given by

$$\hat{\theta} = \frac{1}{N} \sum_{i=1}^N x_i - 1.$$

In the large- N limit, the central limit theorem tells us that the sampling distribution is normal with standard deviation given by the standard error of the mean: $\sigma_{\hat{\theta}}^2 = 1/N$, and we can write the 95% (i.e. 2σ) confidence interval as

$$CI_{\text{large } N} = \left(\hat{\theta} - 2N^{-1/2}, \hat{\theta} + 2N^{-1/2} \right)$$

For our particular observed data, this gives a confidence interval around our unbiased estimator of $CI(\theta) = (10.2, 12.5)$, entirely above our common-sense bound of $\theta < 10$! We might hope that this discrepancy is due to our use of the large- N approximation with a paltry $N = 3$ samples. A more careful treatment of the problem (See [Jaynes1976] or part III of [VanderPlas2014]) gives the exact confidence interval (10.2, 12.2): the 95% confidence interval entirely excludes the sensible bound $\theta < 10$!

Truncated Exponential: A Bayesian Approach

A Bayesian approach to the problem starts with Bayes' rule:

$$P(\theta|D) = \frac{P(D|\theta)P(\theta)}{P(D)}.$$

We use the likelihood given by

$$P(D|\theta) \propto \prod_{i=1}^N P(x_i|\theta)$$

and, in the absence of other information, use an uninformative flat prior on θ to find

$$P(\theta|D) \propto \begin{cases} N \exp[N(\theta - \min(D))] & , \theta < \min(D) \\ 0 & , \theta > \min(D) \end{cases}$$

where $\min(D)$ is the smallest value in the data D , which enters because of the truncation of $P(x_i|\theta)$. Because $P(\theta|D)$ increases exponentially up to the cutoff, the shortest 95% credibility interval (θ_1, θ_2) will be given by $\theta_2 = \min(D)$, and θ_1 given by the solution to the equation

$$\int_{\theta_1}^{\theta_2} P(\theta|D)d\theta = f$$

which has the solution

$$\theta_1 = \theta_2 + \frac{1}{N} \ln \left[1 - f(1 - e^{-N\theta_2}) \right].$$

For our particular data, the Bayesian credible region is

$$CR(\theta) = (9.0, 10.0)$$

which agrees with our common-sense bound.

Discussion

Why do the frequentist CI and Bayesian CR give such different results? The reason goes back to the definitions of the CI and CR, and to the fact that *the two approaches are answering different questions*. The Bayesian CR answers a question about the value of θ itself (the probability that the parameter is in the fixed CR), while the frequentist CI answers a question about the procedure used to construct the CI (the probability that any potential CI will contain the fixed parameter).

Using Monte Carlo simulations, it is possible to confirm that both the above results correctly answer their respective questions (see [VanderPlas2014], III). In particular, 95% of frequentist CIs constructed using data drawn from this model in fact contain the true θ . Our particular data are simply among the unhappy 5% which the confidence interval misses. But this makes clear the danger of misapplying the Bayesian interpretation to a CI: this particular CI is not 95% likely to contain the true value of θ ; it is in fact 0% likely!

This shows that when using frequentist methods on fixed data, we must carefully keep in mind what question frequentism is answering. Frequentism does not seek a *probabilistic statement about a fixed interval* as the Bayesian approach does; it instead seeks probabilistic statements about an *ensemble of constructed intervals*, with the particular computed interval just a single draw from among them. Despite this, it is common to see a 95% confidence interval interpreted in the Bayesian sense: as a fixed interval that the parameter is expected to be found in 95% of the time. As seen clearly here, this interpretation is flawed, and should be carefully avoided.

Though we used a correct unbiased frequentist estimator above, it should be emphasized that the unbiased estimator is not always optimal for any given problem: especially one with small N and/or censored models; see, e.g. [Hardy2003]. Other frequentist estimators are available: for example, if the (biased) maximum likelihood estimator were used here instead, the confidence interval would be very similar to the Bayesian credible region derived above. Regardless of the choice of frequentist estimator, however, the correct interpretation of the CI is the same: it gives probabilities concerning the *recipe for constructing limits*, not for the *parameter values given the observed data*. For sensible

parameter constraints from a single dataset, Bayesianism may be preferred, especially if the difficulties of uninformative priors can be avoided through the use of true prior information.

Bayesianism in Practice: Markov Chain Monte Carlo

Though Bayesianism has some nice features in theory, in practice it can be extremely computationally intensive: while simple problems like those examined above lend themselves to relatively easy analytic integration, real-life Bayesian computations often require numerical integration of high-dimensional parameter spaces.

A turning-point in practical Bayesian computation was the development and application of sampling methods such as Markov Chain Monte Carlo (MCMC). MCMC is a class of algorithms which can efficiently characterize even high-dimensional posterior distributions through drawing of randomized samples such that the points are distributed according to the posterior. A detailed discussion of MCMC is well beyond the scope of this paper; an excellent introduction can be found in [Gelman2004]. Below, we will propose a straightforward model and compare a standard frequentist approach with three MCMC implementations available in Python.

Application: A Simple Linear Model

As an example of a more realistic data-driven analysis, let's consider a simple three-parameter linear model which fits a straight-line to data with unknown errors. The parameters will be the y -intercept α , the slope β , and the (unknown) normal scatter σ about the line.

For data $D = \{x_i, y_i\}$, the model is

$$\hat{y}(x_i|\alpha, \beta) = \alpha + \beta x_i,$$

and the likelihood is the product of the Gaussian distribution for each point:

$$\mathcal{L}(D|\alpha, \beta, \sigma) = (2\pi\sigma^2)^{-N/2} \prod_{i=1}^N \exp\left[-\frac{[y_i - \hat{y}(x_i|\alpha, \beta)]^2}{2\sigma^2}\right].$$

We will evaluate this model on the following data set:

```
import numpy as np
np.random.seed(42) # for repeatability
theta_true = (25, 0.5)
xdata = 100 * np.random.random(20)
ydata = theta_true[0] + theta_true[1] * xdata
ydata = np.random.normal(ydata, 10) # add error
```

Below we will consider a frequentist solution to this problem computed with the statsmodels package⁶, as well as a Bayesian solution computed with several MCMC implementations in Python: emcee⁷, PyMC⁸, and PyStan⁹. A full discussion of the strengths and weaknesses of the various MCMC algorithms used by the packages is out of scope for this paper, as is a full discussion of performance benchmarks for the packages. Rather, the purpose of this section is to show side-by-side examples of the Python APIs of the packages. First, though, we will consider a frequentist solution.

6. statsmodels: Statistics in Python <http://statsmodels.sourceforge.net/>

7. emcee: The MCMC Hammer <http://dan.iel.fm/emcee/>

8. PyMC: Bayesian Inference in Python <http://pymc-devs.github.io/pymc/>

9. PyStan: The Python Interface to Stan <https://pystan.readthedocs.org/>

Frequentist Solution

A frequentist solution can be found by computing the maximum likelihood point estimate. For standard linear problems such as this, the result can be computed using efficient linear algebra. If we define the *parameter vector*, $\theta = [\alpha \ \beta]^T$; the *response vector*, $Y = [y_1 \ y_2 \ y_3 \ \dots \ y_N]^T$; and the *design matrix*,

$$X = \begin{bmatrix} 1 & 1 & 1 & \dots & 1 \\ x_1 & x_2 & x_3 & \dots & x_N \end{bmatrix}^T,$$

it can be shown that the maximum likelihood solution is

$$\hat{\theta} = (X^T X)^{-1} (X^T Y).$$

The confidence interval around this value is an ellipse in parameter space defined by the following matrix:

$$\Sigma_{\hat{\theta}} \equiv \begin{bmatrix} \sigma_{\alpha}^2 & \sigma_{\alpha\beta} \\ \sigma_{\alpha\beta} & \sigma_{\beta}^2 \end{bmatrix} = \sigma^2 (M^T M)^{-1}.$$

Here σ is our unknown error term; it can be estimated based on the variance of the residuals about the fit. The off-diagonal elements of $\Sigma_{\hat{\theta}}$ are the correlated uncertainty between the estimates. In code, the computation looks like this:

```
>>> X = np.vstack([np.ones_like(xdata), xdata]).T
>>> theta_hat = np.linalg.solve(np.dot(X.T, X),
...                             np.dot(X.T, ydata))
>>> y_hat = np.dot(X, theta_hat)
>>> sigma_hat = np.std(ydata - y_hat)
>>> Sigma = sigma_hat ** 2 * \
...         np.linalg.inv(np.dot(X.T, X))
```

The 1σ and 2σ results are shown by the black ellipses in Figure 1.

In practice, the frequentist approach often relies on many more statistical diagnostics beyond the maximum likelihood and confidence interval. These can be computed quickly using convenience routines built-in to the statsmodels package [Seabold2010]. For this problem, it can be used as follows:

```
>>> import statsmodels.api as sm # version 0.5
>>> X = sm.add_constant(xdata)
>>> result = sm.OLS(ydata, X).fit()
>>> sigma_hat = result.params
>>> Sigma = result.cov_params()
>>> print(result.summary2())
```

```
=====
Model:                OLS      AIC:                147.773
Dependent Variable:   y        BIC:                149.765
No. Observations:    20        Log-Likelihood:     -71.887
Df Model:             1         F-statistic:        41.97
Df Residuals:        18        Prob (F-statistic): 4.3e-06
R-squared:            0.70      Scale:              86.157
Adj. R-squared:      0.68

-----
                Coef.  Std.Err.  t      P>|t|  [0.025  0.975]
-----
const    24.6361    3.7871    6.5053  0.0000  16.6797  32.592
x1        0.4483    0.0692    6.4782  0.0000   0.3029   0.593
-----
Omnibus:                1.996    Durbin-Watson:      2.75
Prob(Omnibus):          0.369    Jarque-Bera (JB):   1.63
Skew:                   0.651    Prob(JB):           0.44
Kurtosis:               2.486    Condition No.:      100
=====
```

The summary output includes many advanced statistics which we don't have space to fully discuss here. For a trained practitioner these diagnostics are very useful for evaluating and comparing fits,

especially for more complicated models; see [Wasserman2004] and the statsmodels project documentation for more details.

Bayesian Solution: Overview

The Bayesian result is encapsulated in the posterior, which is proportional to the product of the likelihood and the prior; in this case we must be aware that a flat prior is not uninformative. Because of the nature of the slope, a flat prior leads to a much higher probability for steeper slopes. One might imagine addressing this by transforming variables, e.g. using a flat prior on the angle the line makes with the x-axis rather than the slope. It turns out that the appropriate change of variables can be determined much more rigorously by following arguments first developed by [Jeffreys1946].

Our model is given by $y = \alpha + \beta x$ with probability element $P(\alpha, \beta) d\alpha d\beta$. By symmetry, we could just as well have written $x = \alpha' + \beta' y$ with probability element $Q(\alpha', \beta') d\alpha' d\beta'$. It then follows that $(\alpha', \beta') = (-\beta^{-1}\alpha, \beta^{-1})$. Computing the determinant of the Jacobian of this transformation, we can then show that $Q(\alpha', \beta') = \beta^3 P(\alpha, \beta)$. The symmetry of the problem requires equivalence of P and Q , or $\beta^3 P(\alpha, \beta) = P(-\beta^{-1}\alpha, \beta^{-1})$, which is a functional equation satisfied by

$$P(\alpha, \beta) \propto (1 + \beta^2)^{-3/2}.$$

This turns out to be equivalent to choosing flat priors on the alternate variables $(\theta, \alpha_{\perp}) = (\tan^{-1}\beta, \alpha \cos \theta)$.

Through similar arguments based on the invariance of σ under a change of units, we can show that

$$P(\sigma) \propto 1/\sigma,$$

which is most commonly known as the *Jeffreys Prior* for scale factors after [Jeffreys1946], and is equivalent to flat prior on $\log \sigma$. Putting these together, we find the following uninformative prior for our linear regression problem:

$$P(\alpha, \beta, \sigma) \propto \frac{1}{\sigma} (1 + \beta^2)^{-3/2}.$$

With this prior and the above likelihood, we are prepared to numerically evaluate the posterior via MCMC.

Solution with emcee

The emcee package [ForemanMackey2013] is a lightweight pure-Python package which implements Affine Invariant Ensemble MCMC [Goodman2010], a sophisticated version of MCMC sampling. To use emcee, all that is required is to define a Python function representing the logarithm of the posterior. For clarity, we will factor this definition into two functions, the log-prior and the log-likelihood:

```
import emcee # version 2.0

def log_prior(theta):
    alpha, beta, sigma = theta
    if sigma < 0:
        return -np.inf # log(0)
    else:
        return (-1.5 * np.log(1 + beta**2)
               - np.log(sigma))

def log_like(theta, x, y):
    alpha, beta, sigma = theta
    y_model = alpha + beta * x
    return -0.5 * np.sum(np.log(2*np.pi*sigma**2) +
                        (y-y_model)**2 / sigma**2)
```

```
def log_posterior(theta, x, y):
    return log_prior(theta) + log_like(theta, x, y)
```

Next we set up the computation. emcee combines multiple interacting "walkers", each of which results in its own Markov chain. We will also specify a burn-in period, to allow the chains to stabilize prior to drawing our final traces:

```
ndim = 3 # number of parameters in the model
nwalkers = 50 # number of MCMC walkers
nburn = 1000 # "burn-in" to stabilize chains
nsteps = 2000 # number of MCMC steps to take
starting_guesses = np.random.rand(nwalkers, ndim)
```

Now we call the sampler and extract the trace:

```
sampler = emcee.EnsembleSampler(nwalkers, ndim,
                               log_posterior,
                               args=[xdata, ydata])
sampler.run_mcmc(starting_guesses, nsteps)

# chain is of shape (nwalkers, nsteps, ndim):
# discard burn-in points and reshape:
trace = sampler.chain[:, nburn:, :].T
trace = trace.reshape(-1, ndim).T
```

The result is shown by the blue curve in Figure 1.

Solution with PyMC

The PyMC package [Patil2010] is an MCMC implementation written in Python and Fortran. It makes use of the classic Metropolis-Hastings MCMC sampler [Gelman2004], and includes many built-in features, such as support for efficient sampling of common prior distributions. Because of this, it requires more specialized boilerplate than does emcee, but the result is a very powerful tool for flexible Bayesian inference.

The example below uses PyMC version 2.3; as of this writing, there exists an early release of version 3.0, which is a complete rewrite of the package with a more streamlined API and more efficient computational backend. To use PyMC, we first we define all the variables using its classes and decorators:

```
import pymc # version 2.3

alpha = pymc.Uniform('alpha', -100, 100)

@pymc.stochastic(observed=False)
def beta(value=0):
    return -1.5 * np.log(1 + value**2)

@pymc.stochastic(observed=False)
def sigma(value=1):
    return -np.log(abs(value))

# Define the form of the model and likelihood
@pymc.deterministic
def y_model(x=xdata, alpha=alpha, beta=beta):
    return alpha + beta * x

y = pymc.Normal('y', mu=y_model, tau=1./sigma**2,
               observed=True, value=ydata)

# package the full model in a dictionary
model = dict(alpha=alpha, beta=beta, sigma=sigma,
            y_model=y_model, y=y)
```

Next we run the chain and extract the trace:

```
S = pymc.MCMC(model)
S.sample(iter=100000, burn=50000)
trace = [S.trace('alpha')[:, :], S.trace('beta')[:, :],
        S.trace('sigma')[:, :]]
```

The result is shown by the red curve in Figure 1.

Solution with PyStan

PyStan is the official Python interface to Stan, a probabilistic programming language implemented in C++ and making use of a Hamiltonian MCMC using a No U-Turn Sampler [Hoffman2014]. The Stan language is specifically designed for the expression of probabilistic models; PyStan lets Stan models specified in the form of Python strings be parsed, compiled, and executed by the Stan library. Because of this, PyStan is the least "Pythonic" of the three frameworks:

```
import pystan # version 2.2

model_code = """
data {
  int<lower=0> N; // number of points
  real x[N]; // x values
  real y[N]; // y values
}
parameters {
  real alpha_perp;
  real<lower=-pi()/2, upper=pi()/2> theta;
  real log_sigma;
}
transformed parameters {
  real alpha;
  real beta;
  real sigma;
  real ymodel[N];
  alpha <- alpha_perp / cos(theta);
  beta <- sin(theta);
  sigma <- exp(log_sigma);
  for (j in 1:N)
    ymodel[j] <- alpha + beta * x[j];
}
model {
  y ~ normal(ymodel, sigma);
}
"""

# perform the fit & extract traces
data = {'N': len(xdata), 'x': xdata, 'y': ydata}
fit = pystan.stan(model_code=model_code, data=data,
                  iter=25000, chains=4)
tr = fit.extract()
trace = [tr['alpha'], tr['beta'], tr['sigma']]
```

The result is shown by the green curve in Figure 1.

Comparison

The 1σ and 2σ posterior credible regions computed with these three packages are shown beside the corresponding frequentist confidence intervals in Figure 1. The frequentist result gives slightly tighter bounds; this is primarily due to the confidence interval being computed assuming a single maximum likelihood estimate of the unknown scatter, σ (this is analogous to the use of the single point estimate for the nuisance parameter p in the billiard game, above). This interpretation can be confirmed by plotting the Bayesian posterior conditioned on the maximum likelihood estimate $\hat{\sigma}$; this gives a credible region much closer to the frequentist confidence interval.

The similarity of the three MCMC results belie the differences in algorithms used to compute them: by default, PyMC uses a Metropolis-Hastings sampler, PyStan uses a No U-Turn Sampler (NUTS), while emcee uses an affine-invariant ensemble sampler. These approaches are known to have differing performance characteristics depending on the features of the posterior being

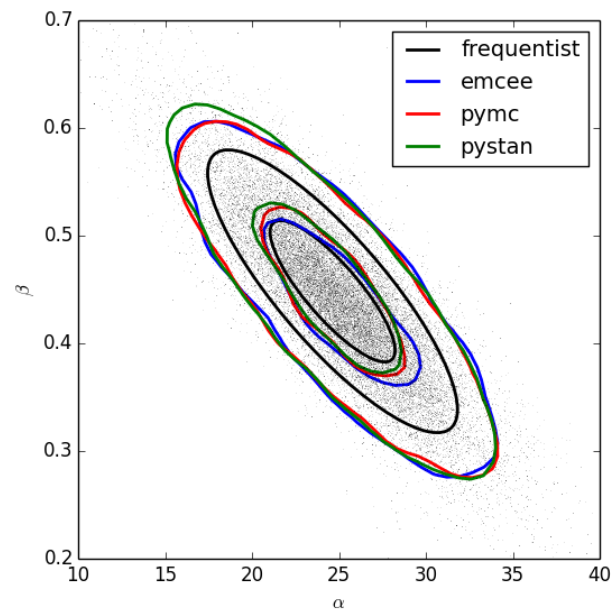


Fig. 1: Comparison of model fits using frequentist maximum likelihood, and Bayesian MCMC using three Python packages: emcee, PyMC, and PyStan.

explored. As expected for the near-Gaussian posterior used here, the three approaches give very similar results.

A main apparent difference between the packages is the Python interface. Emcee is perhaps the simplest, while PyMC requires more package-specific boilerplate code. PyStan is the most complicated, as the model specification requires directly writing a string of Stan code.

Conclusion

This paper has offered a brief philosophical and practical glimpse at the differences between frequentist and Bayesian approaches to statistical analysis. These differences have their root in differing conceptions of probability: frequentists define probability as related to *frequencies of repeated events*, while Bayesians define probability as a *measure of uncertainty*. In practice, this means that frequentists generally quantify the properties of *data-derived quantities* in light of *fixed model parameters*, while Bayesians generally quantify the properties of *unknown models parameters* in light of *observed data*. This philosophical distinction often makes little difference in simple problems, but becomes important within more sophisticated analysis.

We first considered the case of nuisance parameters, and showed that Bayesianism offers more natural machinery to deal with nuisance parameters through *marginalization*. Of course, this marginalization depends on having an accurate prior probability for the parameter being marginalized.

Next we considered the difference in the handling of uncertainty, comparing frequentist confidence intervals with Bayesian credible regions. We showed that when attempting to find a single, fixed interval bounding the true value of a parameter, the Bayesian solution answers the question that researchers most often ask. The frequentist solution can be informative; we just must be careful to correctly interpret the frequentist confidence interval.

Finally, we combined these ideas and showed several examples of the use of frequentism and Bayesianism on a more realistic linear regression problem, using several mature packages available in the Python language ecosystem. Together, these packages offer a set of tools for statistical analysis in both the frequentist and Bayesian frameworks.

So which approach is best? That is somewhat a matter of personal ideology, but also depends on the nature of the problem at hand. Frequentist approaches are often easily computed and are well-suited to truly repeatable processes and measurements, but can hit snags with small sets of data and models which depart strongly from Gaussian. Frequentist tools for these situations do exist, but often require subtle considerations and specialized expertise. Bayesian approaches require specification of a potentially subjective prior, and often involve intensive computation via MCMC. However, they are often conceptually more straightforward, and pose results in a way that is much closer to the questions a scientist wishes to answer: i.e. how do *these particular data* constrain the unknowns in a certain model? When used with correct understanding of their application, both sets of statistical tools can be used to effectively interpret of a wide variety of scientific and technical results.

REFERENCES

- [Bayes1763] T. Bayes. *An essay towards solving a problem in the doctrine of chances*. Philosophical Transactions of the Royal Society of London 53(0):370-418, 1763
- [Eddy2004] S.R. Eddy. *What is Bayesian statistics?*. Nature Biotechnology 22:1177-1178, 2004
- [Evans2002] S.N. Evans & P.B. Stark. *Inverse Problems as Statistics*. Mathematics Statistics Library, 609, 2002.
- [ForemanMackey2013] D. Foreman-Mackey, D.W. Hogg, D. Lang, J.Goodman. *emcee: the MCMC Hammer*. PASP 125(925):306-312, 2014
- [Gelman2004] A. Gelman, J.B. Carlin, H.S. Stern, and D.B. Rubin. *Bayesian Data Analysis, Second Edition*. Chapman and Hall/CRC, Boca Raton, FL, 2004.
- [Goodman2010] J. Goodman & J. Weare. *Ensemble Samplers with Affine Invariance*. Comm. in Applied Mathematics and Computational Science 5(1):65-80, 2010.
- [Hardy2003] M. Hardy. *An illuminating counterexample*. Am. Math. Monthly 110:234–238, 2003.
- [Hoffman2014] M.C. Hoffman & A. Gelman. *The No-U-Turn Sampler: Adaptively Setting Path Lengths in Hamiltonian Monte Carlo*. JMLR, submitted, 2014.
- [Jaynes1976] E.T. Jaynes. *Confidence Intervals vs Bayesian Intervals (1976)* Papers on Probability, Statistics and Statistical Physics Synthese Library 158:149, 1989
- [Jeffreys1946] H. Jeffreys *An Invariant Form for the Prior Probability in Estimation Problems*. Proc. of the Royal Society of London. Series A 186(1007): 453, 1946
- [Patil2010] A. Patil, D. Huard, C.J. Fonnesbeck. *PyMC: Bayesian Stochastic Modelling in Python* Journal of Statistical Software, 35(4):1-81, 2010.
- [Seabold2010] J.S. Seabold and J. Perktold. *Statsmodels: Econometric and Statistical Modeling with Python* Proceedings of the 9th Python in Science Conference, 2010
- [VanderPlas2014] J. VanderPlas. *Frequentism and Bayesianism*. Four-part series (I, II, III, IV) on *Pythonic Perambulations* <http://jakevdp.github.io/>, 2014.
- [Wasserman2004] L. Wasserman. *All of statistics: a concise course in statistical inference*. Springer, 2004.

Blaze: Building A Foundation for Array-Oriented Computing in Python

Mark Wiebe^{‡*}, Matthew Rocklin[‡], TJ Alumbaugh[‡], Andy Terrel[‡]

<http://www.youtube.com/watch?v=9HPR-1PdZUk>



Abstract—We present the motivation and architecture of Blaze, a library for cross-backend data-oriented computation. Blaze provides a standard interface to connect users familiar with NumPy and Pandas to other data analytics libraries like SQLAlchemy and Spark. We motivate the use of these projects through Blaze and discuss the benefits of standard interfaces on top of an increasingly varied software ecosystem. We give an overview of the Blaze architecture and then demonstrate its use on a typical problem. We use the abstract nature of Blaze to quickly benchmark and compare the performance of a variety of backends on a standard problem.

Index Terms—array programming, big data, numpy, scipy, pandas

Introduction

Standard Interfaces

Software and user communities around data analysis have changed remarkably in the last few years. The growth in this ecosystem come both from new computational systems and also from an increasing breadth of users. On the software side we see activity in different languages like Python [Pyt14], R [RLa14], and Julia [Jul12], and also in distributed systems like the projects surrounding the Hadoop File System (HDFS) [Bor07]. On the user side we see increased adoption both from physical scientists, with a strong tradition of computation, and also from social scientists and policy makers with less rigorous training. While these upward trends are encouraging, they also place significant strain on the programming ecosystem. Keeping novice users adapted to quickly changing programming paradigms and operational systems is challenging.

Standard interfaces facilitate interactions between layers of complex and changing systems. For example, NumPy fancy indexing syntax has become a standard interface among array programming systems within the Python ecosystem. Projects with very different implementations (e.g. NumPy [Van11], SciPy.sparse [Jon01], Theano [Ber10]), SciDB [Bro10]) all provide the same indexing interface despite operating very differently.

Standard interfaces help users to adapt to changing technologies without learning new programming paradigms. Standard interfaces help project developers by bootstrapping a well trained community of users. Uniformity smoothes adoption and allows the

ecosystem to evolve rapidly without the drag of everyone having to constantly learn new technologies.

Interactive Arrays and Tables

Analysis libraries like NumPy and Pandas demonstrate the value of interactive array and table objects. Projects such as these connect a broad base of users to efficient low-level operations through a high-level interface. This approach has given rise to large and productive software ecosystems within numeric Python (e.g. SciPy, Scikits, etc.) However, both NumPy and Pandas are largely restricted to an in-memory computational model, limiting problem sizes to a certain scale.

Concurrently developed data analytic ecosystems in other languages like R and Julia provide similar styles of functionality with different application foci. The Hadoop File System (HDFS) has accrued a menagerie of powerful distributed computing systems such as Hadoop, Spark, and Impala. The broader scientific computing community has produced projects like Elemental and SciDB for distributed array computing in various contexts. Finally, traditional SQL databases such as MySQL and Postgres remain both popular and very powerful.

As problem sizes increase and applications become more interdisciplinary, analysts increasingly require interaction with projects outside of the NumPy/Pandas ecosystem. Unfortunately, these foreign projects rarely feel as comfortable or as usable as the Pandas DataFrame.

What is Blaze

Blaze provides a familiar interface around computation on a diverse set of computational systems, or backends. It provides extensible mechanisms to connect this interface to new computational backends. Backends which the Blaze project explicitly provides hooks to include Python, Pandas, SQLAlchemy, and Spark.

This abstract connection to a variety of projects has the following virtues:

- Novice users gain access to relatively exotic technologies
- Users can trivially shift computational backends within a single workflow
- Projects can trivially shift backends as technologies change
- New technologies are provided with a stable interface and a trained set of users

Blaze doesn't do any computation itself. Instead it depends heavily on existing projects to perform computations. Currently

* Corresponding author: mwiebe@continuum.io

‡ Continuum Analytics

Blaze covers tabular computations as might fit into the SQL or Pandas model of computation. We intend to extend this model to arrays and other highly-regular computational models in the future.

Related Work

We separate related work into two categories:

- 1) Computational backends useful to Blaze
- 2) Similar efforts in uniform interfaces

Computational backends on which Blaze currently relies include Pandas, SQLAlchemy, PyToolz, Spark, PyTables, NumPy, and DyND. Pandas [McK10] provides an efficient in-memory table object. SQLAlchemy [sqlal] handles connection to a variety of SQL dialects like SQLite and Postgres. PyToolz [Roc13] provides tuned functions for streaming computation on core Python data structures. NumPy [Van11] and DyND [Wie13] serve as in-memory arrays and common data interchange formats. PyTables [Alt03] provides efficient sequential computations on out-of-core HDF5 files.

Uniform symbolic interfaces on varied computational resources are also common. SQLAlchemy provides a uniform interface onto various SQL implementations. Theano [Ber10] maps array operations onto Python/NumPy, C, or CUDA code generation. While computer algebra projects like SymPy [Sym08] often have expression trees they also commonly include some form of code generation to low-level languages like C, Fortran but also to languages like LaTeX and DOT for visualization.

Blaze Architecture

Blaze separates data analytics into three isolated components:

- Data access: *access* data efficiently across different storage systems, e.g. CSV, HDF5, HDFS,
- Symbolic Expression: *reason* symbolically about the desired result, e.g. Join, Sum, Split-Apply-Combine,
- Backend Computation: *execute* computations on a variety of backends, e.g. SQL, Pandas, Spark,

We isolate these elements to enable experts to create well crafted solutions in each domain without needing to understand the others, e.g., a Pandas expert can contribute without knowing Spark and vice versa. Blaze provides abstraction layers between these components to enable them to work together cleanly.

The assembly of these components creates in a multi-format, multi-backend computational engine capable of common data analytics operations in a variety of contexts.

Blaze Data

Blaze Data Descriptors are a family of Python objects that provide uniform access to a variety of common data formats. They provide standard iteration, insertion, and NumPy-like fancy indexing over on-disk files in common formats like CSV, JSON, and HDF5 in memory data structures like core Python data structures and NumPy arrays as well as more sophisticated data stores like SQL databases. The data descriptor interface is analogous to the Python buffer interface described in PEP 3118 [Oli06], but with a more flexible API.

Over the course of this article we'll refer to the following simple `accounts.csv` file:

```
id, name, balance
1, Alice, 100
2, Bob, -200
3, Charlie, 300
4, Denis, 400
5, Edith, -500
```

```
>>> from blaze import *
>>> csv = CSV('accounts.csv') # Create data object
```

Iteration: Data descriptors expose the `__iter__` method, which provides an iterator over the outermost dimension of the data. This iterator yields vanilla Python objects by default.

```
>>> list(csv)
[(1L, u'Alice', 100L),
 (2L, u'Bob', -200L),
 (3L, u'Charlie', 300L),
 (4L, u'Denis', 400L),
 (5L, u'Edith', -500L)]
```

Data descriptors also expose a `chunks` method, which also iterates over the outermost dimension but instead of yielding single rows of Python objects instead yields larger chunks of compactly stored data. These chunks emerge as DyND arrays that are more efficient for bulk processing and data transfer. DyND arrays support the `__array__` interface and so can be easily converted to NumPy arrays.

```
>>> next(csv.chunks())
nd.array([[1, "Alice", 100],
          [2, "Bob", -200],
          [3, "Charlie", 300],
          [4, "Denis", 400],
          [5, "Edith", -500]],
         type="5 * {id : int64, name : string, balance : int64}")
```

Insertion: Analogously to `__iter__` and `chunks`, the methods `extend` and `extend_chunks` allow for insertion of data into the data descriptor. These methods take iterators of Python objects and DyND arrays respectively. The data is coerced into whatever form is native for the storage medium, e.g. text for CSV, or INSERT statements for SQL.

```
>>> csv = CSV('accounts.csv', mode='a')
>>> csv.extend([(6, 'Frank', 600),
...           (7, 'Georgina', 700)])
```

Migration: The combination of uniform iteration and insertion along with robust type coercion enables trivial data migration between storage systems.

```
>>> sql = SQL('postgresql://user:pass@host/',
             'accounts', schema=csv.schema)
>>> sql.extend(iter(csv)) # Migrate csv file to DB
```

Indexing: Data descriptors also support fancy indexing. As with iteration, this supports either Python objects or DyND arrays through the `.py[...]` and `.dynd[...]` interfaces.

```
>>> list(csv.py[:,2, ['name', 'balance']])
[(u'Alice', 100L),
 (u'Charlie', 300L),
 (u'Edith', -500L),
 (u'Georgina', 700L)]

>>> csv.dynd[:,2, ['name', 'balance']]
nd.array([[ "Alice", 100],
```

```

["Charlie", 300],
["Edith", -500],
["Georgina", 700]],
type="var * {name : string, balance : int64}")

```

Performance of this approach varies depending on the underlying storage system. For file-based storage systems like CSV and JSON, it is necessary to seek through the file to find the right line (see [iopro]), but don't incur needless deserialization costs (i.e. converting text into floats, ints, etc.) which tend to dominate ingest times. Some storage systems, like HDF5, support random access natively.

Cohesion: Different storage techniques manage data differently. Cohesion between these disparate systems is accomplished with the two projects `dashape`, which specifies the intended meaning of the data, and `DyND`, which manages efficient type coercions and serves as an efficient intermediate representation.

Blaze Expr

To be able to run analytics on a wide variety of computational backends, it's important to have a way to represent them independent of any particular backend. Blaze uses abstract expression trees for this, including convenient syntax for creating them and a pluggable multiple dispatch mechanism for lowering them to a computation backend. Once an analytics computation is represented in this form, there is an opportunity to do analysis and transformation on it prior to handing it off to a backend, both for optimization purposes and to give heuristic feedback to the user about the expected performance.

To illustrate how Blaze expression trees work, we will build up an expression on a table from the bottom, showing the structure of the trees along the way. Let's start with a single table, for which we'll create an expression node

```

>>> accts = TableSymbol('accounts',
...                       '{id: int, name: string, balance: int}')

```

to represent a abstract table of accounts. By defining operations on expression nodes which construct new abstract expression trees, we can provide a familiar interface closely matching that of NumPy and of Pandas. For example, in structured arrays and dataframes you can access fields as `accts['name']`.

Extracting fields from the table gives us `Column` objects, to which we can now apply operations. For example, we can select all accounts with a negative balance.

```

>>> deadbeats = accts[accts['balance'] < 0]['name']

```

or apply the split-apply-combine pattern to get the highest grade in each class

```

>>> By(accts, accts['name'], accts['balance'].sum())

```

In each of these cases we get an abstract expression tree representing the analytics operation we have performed, in a form independent of any particular backend.

```

-----By-----
 /      |      \
accts  Column  Sum
 /      / \      |
accts  /   \   Column
      /     \   /   \
      accts  'name' accts  'balance'

```

Blaze Compute

Once an analytics expression is represented as a Blaze expression tree, it needs to be mapped onto a backend. This is done by walking the tree using the multiple dispatch `compute` function, which defines how an abstract Blaze operation maps to an operation in the target backend.

To see how this works, let's consider how to map the `By` node from the previous section into a Pandas backend. The code that handles this is an overload of `compute` which takes a `By` node and a `DataFrame` object. First, each of the child nodes must be computed, so `compute` gets called on the three child nodes. This validates the provided dataframe against the `accts` schema and extracts the 'name' and 'balance' columns from it. Then, the `pandas.groupby` call is used to group the 'balance' column according to the 'name' column, and apply the `sum` operation.

Each backend can map the common analytics patterns supported by Blaze to its way of dealing with it, either by computing it on the fly as the Pandas backend does, or by building up an expression in the target system such as an SQL statement or an RDD map and `groupByKey` in Spark.

Multiple dispatch provides a pluggable mechanism to connect new back ends, and handle interactions between different backends.

Example

We demonstrate the pieces of Blaze in a small toy example.

Recall our accounts dataset

```

>>> L = [(1, 'Alice', 100),
         (2, 'Bob', -200),
         (3, 'Charlie', 300),
         (4, 'Denis', 400),
         (5, 'Edith', -500)]

```

And our computation for names of account holders with negative balances

```

>>> deadbeats = accts[accts['balance'] < 0]['name']

```

We compose the abstract expression, `deadbeats` with the data `L` using the function `compute`.

```

>>> list(compute(deadbeats, L))
['Bob', 'Edith']

```

Note that the correct answer was returned as a list.

If we now store our same data `L` into a Pandas `DataFrame` and then run the exact same `deadbeats` computation against it, we find the same semantic answer.

```

>>> df=DataFrame(L, columns=['id', 'name', 'balance'])
>>> compute(deadbeats, df)
1      Bob
4      Edith
Name: name, dtype: object

```

Similarly against Spark

```

>>> sc = pyspark.SparkContext('local', 'Spark-app')
>>> rdd = sc.parallelize(L) # Distributed DataStructure

>>> compute(deadbeats, rdd)
PythonRDD[1] at RDD at PythonRDD.scala:37

>>> _.collect()
['Bob', 'Edith']

```

In each case of calling `compute(deadbeats, ...)` against a different data source, Blaze orchestrates the right computational backend to execute the desired query. The result is given in the form received and computation is done either with streaming Python, in memory Pandas, or distributed memory Spark. The user experience is identical in all cases.

Blaze Interface

The separation of expressions and backend computation provides a powerful multi-backend experience. Unfortunately, this separation may also be confusing for a novice programmer. To this end we provide an interactive object that feels much like a Pandas DataFrame, but in fact can be driving any of our backends.

```
>>> sql = SQL('postgres://postgres@localhost',
...           'accounts')
>>> t = Table(sql)
>>> t
   id  name  balance
0   1  Alice    100
1   2   Bob   -200
2   3 Charlie    300
3   4  Denis    400
4   5  Edith   -500

>>> t[t['balance'] < 0]['name']
   name
0   Bob
1  Edith
```

The astute reader will note the use of Pandas-like user experience and output. Note however, that these outputs are the result of computations on a Postgres database.

Discussion

Blaze provides both the ability to migrate data between data formats and to rapidly prototype common analytics operations against a wide variety of computational backends. It allows one to easily compare options and choose the best for a particular setting. As that setting changes, for example when data size grows considerably, our implementation can transition easily to a more suitable backend.

This paper gave an introduction to the benefits of separating expression of a computation from its computation. We expect future work to focus on integrating new backends, extending to array computations, and composing Blaze operations to transform existing in-memory backends like Pandas and DyND into an out-of-core and distributed setting.

REFERENCES

- [Zah10] Zaharia, Matei, et al. "Spark: cluster computing with working sets." Proceedings of the 2nd USENIX conference on Hot topics in cloud computing, 2010.
- [McK10] Wes McKinney. *Data Structures for Statistical Computing in Python*, Proceedings of the 9th Python in Science Conference, 51-56 (2010)
- [sqlal] <http://www.sqlalchemy.org/>
- [ioprop] <http://docs.continuum.io/iopro/index.html>
- [Roc13] Rocklin, Matthew and Welch, Erik and Jacobsen, John. *Toolz Documentation*, 2014 <http://toolz.readthedocs.org/>
- [Wie13] Wiebe, Mark. *LibDyND* <https://github.com/ContinuumIO/libdynd>
- [Sym08] SymPy Development Team. "SymPy: Python library for symbolic mathematics." (2008).
- [Ber10] Bergstra, James, et al. "Theano: a CPU and GPU math compiler in Python." Proc. 9th Python in Science Conf. 2010.
- [Bor07] Borthakur, Dhruba. "The hadoop distributed file system: Architecture and design." Hadoop Project Website 11 (2007): 21.
- [Alt03] Alted, Francesc, and Mercedes Fernández-Alonso. "PyTables: processing and analyzing extremely large amounts of data in Python." PyCon 2003 (2003).
- [Van11] Stéfan van der Walt, S. Chris Colbert and Gaël Varoquaux. *The NumPy Array: A Structure for Efficient Numerical Computation*, Computing in Science & Engineering, 13, 22-30 (2011),
- [Oli06] Oliphant, Travis and Banks, Carl. <http://legacy.python.org/dev/peps/pep-3118/>
- [Pyt14] G. Van Rossum. The Python Language Reference Manual. Network Theory Ltd., September 2003.
- [RLa14] R Core Team (2014). R: A language and environment for statistical computing. R Foundation for Statistical Computing, Vienna, Austria. URL <http://www.R-project.org/>.
- [Jul12] J. Bezanson, S. Karpinski, V. B. Shah, and A. Edelman. Julia: A fast dynamic language for technical computing. CoRR, abs/1209.5145, 2012.
- [Jon01] Jones E, Oliphant E, Peterson P, et al. SciPy: Open Source Scientific Tools for Python, 2001-, <http://www.scipy.org/> [Online; accessed 2014-09-25].
- [Bro10] Paul G. Brown, Overview of sciDB: large scale array storage, processing and analysis, Proceedings of the 2010 ACM SIGMOD International Conference on Management of data, June 06-10, 2010, Indianapolis, Indiana, USA

Simulating X-ray Observations with Python

John A. ZuHone^{||*}, Veronica Biffi[¶], Eric J. Hallman[§], Scott W. Randall[‡], Adam R. Foster[‡], Christian Schmid^{**}

<http://www.youtube.com/watch?v=fUMq6rmNshc>



Abstract—X-ray astronomy is an important tool in the astrophysicist's toolkit to investigate high-energy astrophysical phenomena. Theoretical numerical simulations of astrophysical sources are fully three-dimensional representations of physical quantities such as density, temperature, and pressure, whereas astronomical observations are two-dimensional projections of the emission generated via mechanisms dependent on these quantities. To bridge the gap between simulations and observations, algorithms for generating synthetic observations of simulated data have been developed. We present an implementation of such an algorithm in the `yt` analysis software package. We describe the underlying model for generating the X-ray photons, the important role that `yt` and other Python packages play in its implementation, and present a detailed workable example of the creation of simulated X-ray observations.

Index Terms—astronomical observations, astrophysics simulations, visualization

Introduction

In the early 21st century, astronomy is truly a multi-wavelength enterprise. Ground and space-based instruments across the electromagnetic spectrum, from radio waves to gamma rays, provide the most complete picture of the various physical processes governing the evolution of astrophysical sources. In particular, X-ray astronomy probes high-energy processes in astrophysics, including high-temperature thermal plasmas (e.g., the solar wind, the intracluster medium) and relativistic cosmic rays (e.g., from active galactic nuclei). X-ray astronomy has a long and successful pedigree, with a number of observatories. These include *Einstein*, *ROSAT*, *Chandra*, *XMM-Newton*, *Suzaku*, and *NuSTAR*, as well as upcoming missions such as *Astro-H* and *Athena*.

An important distinguishing feature of X-ray astronomy from that of studies at longer wavelengths is that it is inherently *discrete*, e.g., the numbers of photons per second that reach the detectors are small enough that the continuum approximation, valid for longer-wavelength photons such as those in the visible light,

infrared, microwave, and radio bands, is invalid. Instead of images, the fundamental data products of X-ray astronomy are tables of individual photon positions, energies, and arrival times.

Due to modeling uncertainties, projection effects, and contaminating backgrounds, combining the insights from observations and numerical simulations is not necessarily straightforward. In contrast to simulations, where all of the physical quantities in 3 dimensions are completely known to the precision of the simulation algorithm, astronomical observations are by definition 2-D projections of 3-D sources along a given sight line, and the observed spectrum of emission is a complicated function of the fundamental physical properties (e.g., density, temperature, composition) of the source.

Such difficulties in bridging these two worlds have given rise to efforts to close the gap in the direction of the creation of synthetic observations from simulated data (see, e.g., [Gardini04], [Nagai06], [ZuHone09], and [Heinz11] for recent examples). This involves the determination of the spectrum of the emission from the properties of the source, the projection of this emission along the chosen line of sight, and, in the case of X-ray (and γ -ray) astronomy, the generation of synthetic photon samples. These photons are then convolved with the instrumental responses and (if necessary) background effects are added. One implementation of such a procedure, `PHOX`, was described in [Biffi12] and [Biffi13], and used for the analysis of simulated galaxy clusters from smoothed-particle hydrodynamics (SPH) cosmological simulations. `PHOX` was originally implemented in C using outputs from the `Gadget` SPH code. `PHOX` takes the inputs of density, temperature, velocity, and metallicity from a 3D `Gadget` simulation, using them as inputs to create synthetic spectra (using spectral models from the X-ray spectral fitting package `XSPEC`). Finally, `PHOX` uses these synthetic spectra convolved with instrument response functions to simulate samples of observed photons.

In this work, we describe an extension of this algorithm to outputs from other simulation codes. We developed the module `photon_simulator`, an implementation of `PHOX` within the Python-based `yt` simulation analysis package. We outline the design of the `PHOX` algorithm, the specific advantages to implementing it in Python and `yt`, and provide a workable example of the generation of a synthetic X-ray observation from a simulation dataset.

Model

The overall model that underlies the `PHOX` algorithm may be split up into roughly three steps: first, constructing an original large sample of simulated photons for a given source, second, choosing

* Corresponding author: jzuhone@milkyway.gsfc.nasa.gov

|| Astrophysics Science Division, Laboratory for High Energy Astrophysics, Code 662, NASA/Goddard Space Flight Center, Greenbelt, MD 20771

¶ SISSA - Scuola Internazionale Superiore di Studi Avanzati, Via Bonomea 265, 34136 Trieste, Italy

§ Center for Astrophysics and Space Astronomy, Department of Astrophysical & Planetary Science, University of Colorado, Boulder, CO 80309

‡ Harvard-Smithsonian Center for Astrophysics, 60 Garden Street, Cambridge, MA 02138

** Dr. Karl Remeis-Sternwarte & ECAP, Sternwartstr. 7, 96049 Bamberg, Germany

a subset of these photons corresponding to parameters appropriate to an actual observation and projecting them onto the sky plane, and finally, applying instrumental responses for a given detector. We briefly describe each of these in turn.

Step 1: Generating the Original Photon Sample

In the first step of the PHOX algorithm, we generate a large sample of photons in three dimensions, with energies in the rest frame of the source. These photons will serve as a "Monte-Carlo" sample from which we may draw subsets to construct realistic observations.

First, to determine the energies of the photons, a spectral model for the photon emissivity must be specified. In general, the normalization of the photon emissivity for a given volume element will be set by the number density of emitting particles, and the shape of the spectrum will be set by the energetics of the same particles.

As a specific and highly relevant example, one of the most common sources of X-ray emission is that from a low-density, high-temperature, thermal plasma, such as that found in the solar corona, supernova remnants, "early-type" galaxies, galaxy groups, and galaxy clusters. The specific photon count emissivity associated with a given density, temperature T , and metallicity Z of such a plasma is given by

$$\varepsilon_E^\gamma = n_e n_H \Lambda_E(T, Z) \text{ photons s}^{-1} \text{ cm}^{-3} \text{ keV}^{-1} \quad (1)$$

where the superscript γ refers to the fact that this is a photon count emissivity, E is the photon energy in keV, n_e and n_H are the electron and proton number densities in cm^{-3} , and $\Lambda_E(T, Z)$ is the spectral model in units of $\text{photons s}^{-1} \text{ cm}^3 \text{ keV}^{-1}$. The dominant contributions to Λ_E for an optically-thin, fully-ionized plasma are bremsstrahlung ("free-free") emission and collisional line excitation. A number of models for the emissivity of such a plasma have been developed, including Raymond-Smith [Raymond77], MeKaL [Mewe95], and APEC [Smith01]. These models (and others) are all built into the XSPEC package, which includes a Python interface, PyXspec, which is a package we will use to supply the input spectral models to generate the photon energies.

The original PHOX algorithm only allowed for emission from variants of the APEC model for a thermal plasma. However, astrophysical X-ray emission arises from a variety of physical processes and sources, and in some cases multiple sources may be emitting from within the same volume. For example, cosmic-ray electrons in galaxy clusters produce a power-law spectrum of X-ray emission at high energies via inverse-Compton scattering of the cosmic microwave background. Recently, the detection of previously unidentified line emission, potentially from decaying sterile neutrinos, was made in stacked spectra of galaxy clusters [Bulbul14]. The flexibility of our approach allows us to implement one or several models for the X-ray emission arising from a variety of physical processes as the situation requires.

Given a spectral model, for a given volume element i with volume ΔV_i (which may be grid cells or Lagrangian particles), a spectrum of photons may be generated. The *total number* of photons that are generated in our initial sample per volume element i is determined by other factors. We determine the number of photons for each volume element by artificially inflating the parameters that determine the number of photons received by an observer to values that are large compared to more realistic values. The inflated Monte-Carlo sample should be large enough that realistic sized subsets from it are statistically representative. In

the description that follows, parameters with subscript "0" indicate those with "inflated" values, whereas we will drop the subscripts in the second step when choosing more realistic values.

To begin with, the bolometric flux of photons received by the observer from the volume element i is

$$F_i^\gamma = \frac{n_e n_H \Lambda(T_i, Z_i) \Delta V_i}{4\pi D_{A,0}^2 (1+z_0)^2} \text{ photons s}^{-1} \text{ cm}^{-2} \quad (2)$$

where z_0 is the cosmological redshift and $D_{A,0}$ is the angular diameter distance to the source (if the source is nearby, $z_0 \approx 0$ and $D_{A,0}$ is simply the distance to the source). The physical quantities of interest are constant across the volume element. The total number of photons associated with this flux for an instrument with a collecting area $A_{\text{det},0}$ and an observation with exposure time $t_{\text{exp},0}$ is given by

$$N_{\text{phot}} = t_{\text{exp},0} A_{\text{det},0} \sum_i F_i^\gamma \quad (3)$$

By setting $t_{\text{exp},0}$ and $A_{\text{det},0}$ to values that are much larger than those associated with typical exposure times and actual detector areas, and setting z_0 to a value that corresponds to a nearby source (thus ensuring $D_{A,0}$ is similarly small), we ensure that we create suitably large Monte-Carlo sample to draw subsets of photons for more realistic observational parameters. Figure 1 shows a schematic representation of this model for a roughly spherical source of X-ray photons, such as a galaxy cluster.

Step 2: Projecting Photons to Create Specific Observations

The second step in the PHOX algorithm involves using this large 3-D sample of photons to create 2-D projections of simulated events, where a subsample of photons from the original Monte-Carlo sample is selected.

First, we choose a line-of-sight vector $\hat{\mathbf{n}}$ to define the primed coordinate system from which the photon sky positions (x', y') in the observer's coordinate system \mathcal{O}' are determined (c.f. Figure 1). The total emission from any extended object as a function of position on the sky is a projection of the total emission along the line of sight, minus the emission that has been either absorbed or scattered out of the sight-line along the way. In the current state of our implementation, we assume that the source is optically thin to the photons, so they pass essentially unimpeded from the source to the observer (with the caveat that some photons are absorbed by Galactic foreground gas). This is appropriate for most X-ray sources of interest.

Next, we must take into account processes that affect on the photon energies. The first, occurring at the source itself, is Doppler shifting and broadening of spectral lines, which arises from bulk motion of the gas and turbulence. Each volume element has a velocity \mathbf{v}_i in \mathcal{O} , and the component $v_{i,z'}$ of this velocity along the line of sight results in a Doppler shift of each photon's energy of

$$E_1 = E_0 \sqrt{\frac{c + v_{z'}}{c - v_{z'}}} \quad (4)$$

where E_1 and E_0 are the Doppler-shifted and rest-frame energies of the photon, respectively, and c is the speed of light in vacuum. Second, since many X-ray sources are at cosmological distances, each photon is cosmologically redshifted, reducing its energy further by a factor of $1/(1+z)$ before being received in the observer's frame.

Since we are now simulating an actual observation, we choose more realistic values for the exposure time t_{exp} and detector area

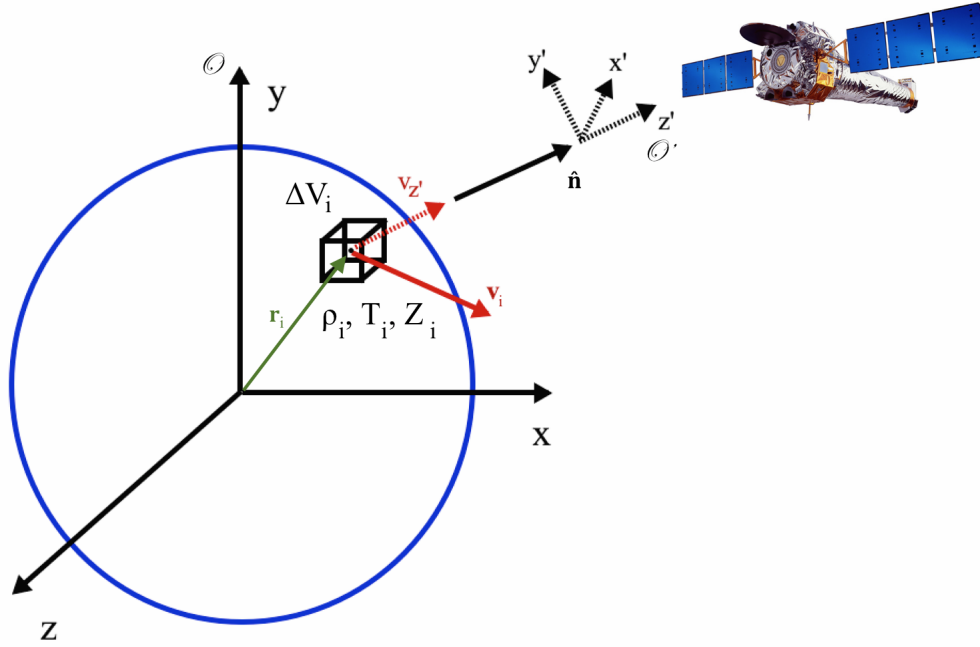


Fig. 1: Schematic representation of a roughly spherical X-ray emitting object, such as a galaxy cluster. The volume element ΔV_i at position \mathbf{r}_i in the coordinate system \mathcal{O} of the source has a velocity \mathbf{v}_i . Photons emitted along the direction given by $\hat{\mathbf{n}}$ will be received in the observer's frame in the coordinate system \mathcal{O}' , and will be Doppler-shifted by the line-of-sight velocity component $v_{i,z'}$. Chandra telescope image credit: NASA/CXC.

A_{det} than we did in the first step to determine the number of photons to use from the original Monte-Carlo sample. Similarly, we may also specify alternative values for the angular diameter distance D_A and the cosmological redshift z , if desired. The fraction f of the photons that will be used in the actual observation is then given by

$$f = \frac{t_{\text{exp}}}{t_{\text{exp},0}} \frac{A_{\text{det}}}{A_{\text{det},0}} \frac{D_{A,0}^2}{D_A^2} \frac{(1+z_0)^3}{(1+z)^3} \quad (5)$$

where $f \leq 1$.

Before being received by the observer, a number of the photons, particularly on the softer end of the spectrum, are absorbed by foregrounds of hydrogen gas in the Milky Way Galaxy. The last operation that is applied in our implementation of the PHOX algorithm is to use a tabulated model for the absorption cross-section as a function of energy (examples include `wabs` [Morrison83], `phabs` [Balucinska-Church92], `tbabs` [Wilms00], all included in XSPEC) as an acceptance-rejection criterion for which photons will be retained in the final sample, e.g., which of them are actually received by the observer.

The advantage of the PHOX algorithm is that the two steps of generating the photons in the source frame and projecting them along a given line of sight are separated, so that the first step, which is the most computationally expensive, need only be done once for a given source, whereas the typically cheaper second step may be repeated many times for many different lines of sight, different instruments, and different exposure times.

Step 3: Modeling Instrumental Effects

Unfortunately, the data products of X-ray observations do not simply consist of the original sky positions and energies of the received photons. Spatially, the positions of the received photons

on the detector are affected by a number of instrumental factors. These include vignetting, the layout of the CCD chips, and a typically spatially dependent point-spread function. Similarly, the photon energies are binned up by the detectors into a set of discrete energy channels, and there is typically not a simple one-to-one mapping between which channel a given photon ends up in and its original energy, but is instead represented by a non-diagonal response matrix. Finally, the "effective" collecting area of the telescope is also energy-dependent, and also varies with position on the detector. When performing analysis of X-ray data, the mapping between the detector channel and the photon energy is generally encapsulated in a [redistribution matrix file \(RMF\)](#), and the effective area curve as a function of energy is encapsulated in an [ancillary response file \(ARF\)](#).

In our framework, we provide two ways of convolving the detected photons with instrumental responses, depending on the level of sophistication required. The first is a "bare-bones" approach, where the photon positions are convolved with a user-specified point-spread function, and the photon energies are convolved with a user-input energy response functions. This will result in photon distributions that are similar enough to the final data products of real observations to be sufficient for most purposes.

However, some users may require a full simulation of a given telescope or may wish to compare observations of the same simulated system by multiple instruments. Several software packages exist for this purpose. The venerable `MARX` software performs detailed ray-trace simulations of how *Chandra* responds to a variety of astrophysical sources, and produces standard event data files in the same FITS formats as standard *Chandra* data products. `SIMX` and `Sixte` are similar packages that simulate most of the effects of the instrumental responses for a variety of current and planned X-ray missions. We provide convenient output

formats for the synthetic photons in order that they may be easily imported into these packages.

Implementation

The model described here has been implemented as the analysis module `photon_simulator` in `yt` [Turk11], a Python-based visualization and analysis toolkit for volumetric data. `yt` has a number of strengths that make it an ideal package for implementing our algorithm.

The first is that `yt` has support for analyzing data from a large number of astrophysical simulation codes (e.g., `FLASH`, `Enzo`, `Gadget`, `Athena`), which simulate the formation and evolution of astrophysical systems using models for the relevant physics, including magnetohydrodynamics, gravity, dark matter, plasmas, etc. The simulation-specific code is contained within various "frontend" implementations, and the user-facing API to perform the analysis on the data is the same regardless of the type of simulation being analyzed. This enables the same function calls to easily generate photons from models produced by any of these simulation codes making it possible to use the `PHOX` algorithm beyond the original application to `Gadget` simulations only. In fact, most previous approaches to simulating X-ray observations were limited to datasets from particular simulation codes.

The second strength is related, in that by largely abstracting out the simulation-specific concepts of "cells", "grids", "particles", "smoothing lengths", etc., `yt` provides a window on to the data defined primarily in terms of physically motivated volumetric region objects. These include spheres, disks, rectangular regions, regions defined on particular cuts on fields, etc. Arbitrary combinations of these region types are also possible. These volumetric region objects serve as natural starting points for generating X-ray photons from not only physically relevant regions within a complex hydrodynamical simulation, but also from simple "toy" models which have been constructed from scratch, when complex, expensive simulations are not necessary.

The third major strength is that implementing our model in `yt` makes it possible to easily make use of the wide variety of useful libraries available within the scientific Python ecosystem. Our implementation uses `SciPy` for integration, `AstroPy` for handling celestial coordinate systems and FITS I/O, and `PyXspec` for generating X-ray spectral models. Tools for analyzing astrophysical X-ray data are also implemented in Python (e.g., `CIAO`'s `Sherpa` package, [Refsdal09]), enabling an easy comparison between models and observations.

Example

Here we present a workable example of creating simulated X-ray events using `yt`'s `photon_simulator` analysis module. We implemented the module in `yt` v. 3.0 as `yt.analysis_modules.photon_simulator`. `yt` v. 3.0 can be downloaded from <http://yt-project.org>. The example code here is available as an [IPython notebook](#). This is not meant to be an exhaustive explanation of all of the `photon_simulator`'s features and options--for these the reader is encouraged to visit the [yt documentation](#).

As our input dataset, we will use an `Athena` simulation of a galaxy cluster core, which can be downloaded from the `yt` website at <http://yt-project.org/data/MHDSloshing.tar.gz>. You will also need to download a version of APEC from <http://www.atomdb.org>. Finally, the absorption cross section table used

here and the `Chandra` response files may be downloaded from http://yt-project.org/data/xray_data.tar.gz.

First, we must import the necessary modules:

```
import yt
from yt.analysis_modules.photon_simulator.api \
    import TableApecModel, ThermalPhotonModel, \
        PhotonList, TableAbsorbModel
from yt.utilities.cosmology import Cosmology
```

Next, we load the dataset `ds`, which comes from a set of simulations presented in [ZuHone14]. `Athena` datasets require a `parameters` dictionary to be supplied to provide unit conversions to Gaussian units; for most datasets generated by other simulation codes that can be read by `yt`, this is not necessary.

```
parameters={"time_unit":(1.0, "Myr"),
            "length_unit":(1.0, "Mpc"),
            "mass_unit":(1.0e14, "Msun")}
```

```
ds = yt.load("MHDSloshing/virgo_low_res.0054.vtk",
            parameters=parameters)
```

Slices through the density and temperature of the simulation dataset are shown in Figure 2. The luminosity and temperature of our model galaxy cluster roughly match that of Virgo. The photons will be created from a spherical region centered on the domain center, with a radius of 250 kpc:

```
sp = ds.sphere("c", (250., "kpc"))
```

This will serve as our `data_source` that we will use later. Now, we are ready to use the `photon_simulator` analysis module to create synthetic X-ray photons from this dataset.

Step 1: Generating the Original Photon Sample

First, we need to create the `SpectralModel` instance that will determine how the data in the grid cells will generate photons. A number of options are available, but we will use the `TableApecModel`, which allows one to use the APEC data tables:

```
atomdb_path = "/Users/jzuhone/Data/atomdb"
apec_model = TableApecModel(atomdb_path,
                            0.01, 10.0, 2000,
                            apec_ver="2.0.2",
                            thermal_broad=False)
```

where the first argument specifies the path to the APEC files, the next three specify the bounds in keV of the energy spectrum and the number of bins in the table, and the remaining arguments specify the APEC version to use and whether or not to apply thermal broadening to the spectral lines.

Now that we have our `SpectralModel`, we need to connect this model to a `PhotonModel` that will connect the field data in the `data_source` to the spectral model to and generate the photons which will serve as the sample distribution for observations. For thermal spectra, we have a special `PhotonModel` called `ThermalPhotonModel`:

```
thermal_model = ThermalPhotonModel(apec_model,
                                   X_H=0.75,
                                   Zmet=0.3)
```

Where we pass in the `SpectralModel`, and can optionally set values for the hydrogen mass fraction `X_H` and metallicity `Z_met`, the latter of which may be a single floating-point value or the name of the `yt` field representing the spatially-dependent metallicity.

Next, we need to specify "fiducial" values for the telescope collecting area in cm^2 , exposure time in seconds, and cosmological

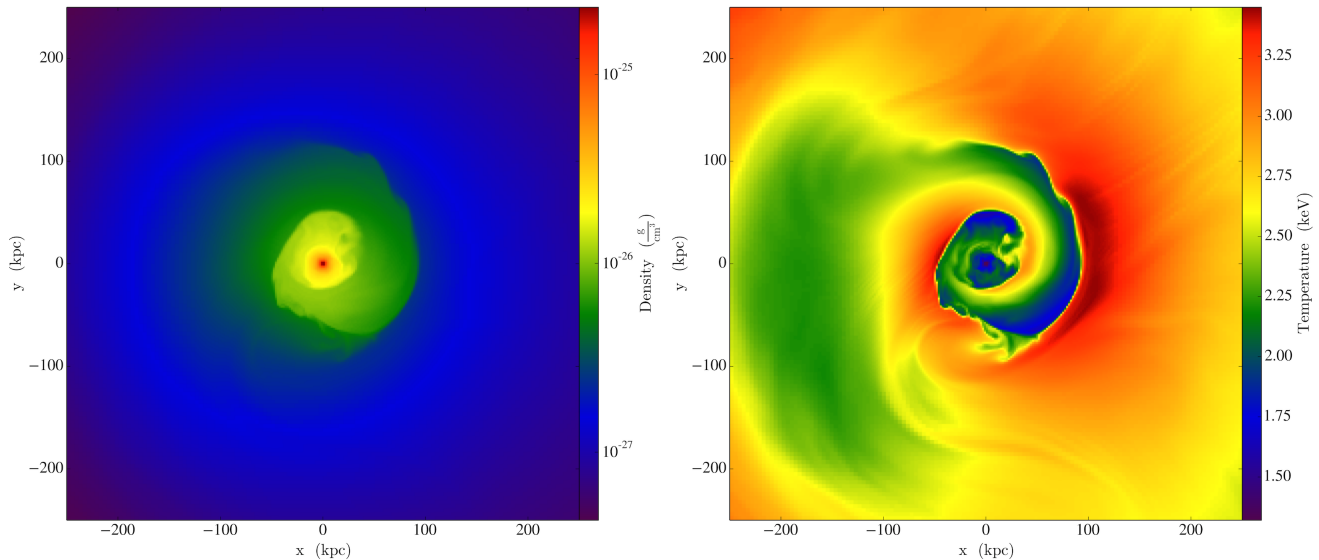


Fig. 2: Slices of density (left) and temperature (right) of an Athena dataset of a galaxy cluster core.

redshift, choosing generous values so that there will be a large number of photons in the Monte-Carlo sample. We also construct a `Cosmology` object, which will be used to determine the source distance from its redshift.

```
A = 6000. # must be in cm**2!
exp_time = 4.0e5 # must be in seconds!
redshift = 0.05
cosmo = Cosmology()
```

By default the `Cosmology` object uses the WMAP7 cosmological parameters from [Komatsu11], but others may be supplied, such as the [Planck13] parameters:

```
cosmo = Cosmology(hubble_constant = 0.67,
                 omega_matter = 0.32,
                 omega_lambda = 0.68)
```

Now, we finally combine everything together and create a `PhotonList` instance, which contains the photon samples:

```
photons = PhotonList.from_scratch(sp, redshift, A,
                                 exp_time,
                                 thermal_model,
                                 center="c",
                                 cosmology=cosmo)
```

where we have used all of the parameters defined above, and `center` defines the reference coordinate which will become the origin of the photon coordinates, which in this case is "c", the center of the simulation domain. This object contains the positions and velocities of the originating volume elements of the photons, as well as their rest-frame energies.

Generating this Monte-Carlo sample is the most computationally intensive part of the PHOX algorithm. Once a sample has been generated it can be saved to disk and loaded as needed rather than needing to be regenerated for different observational scenarios (instruments, redshifts, etc). The photons object can be saved to disk in the HDF5 format with the following method:

```
photons.write_h5_file("my_photons.h5")
```

To load these photons at a later time, we use the `from_file` method:

```
photons = PhotonList.from_file("my_photons.h5")
```

Step 2: Projecting Photons to Create Specific Observations

At this point the photons can be projected along a line of sight to create a specific synthetic observation. First, it is necessary to set up a spectral model for the Galactic absorption cross-section, similar to the spectral model for the emitted photons set up previously. Here again, there are multiple options, but for the current example we use `TableAbsorbModel`, which allows one to use an absorption cross section vs. energy table written in HDF5 format (available in the `xray_data.tar.gz` file mentioned previously). This method also takes the column density `N_H` in units of 10^{22} cm^{-2} as an additional argument.

```
N_H = 0.1
a_mod = TableAbsorbModel("tbabs_table.h5", N_H)
```

We next set a line-of-sight vector `L`:

```
L = [0.0, 0.0, 1.0]
```

which corresponds to the direction within the simulation domain along which the photons will be projected. The exposure time, telescope area, and source redshift may also be optionally set to more appropriate values for a particular observation:

```
texp = 1.0e5
z = 0.07
```

If any of them are not set, those parameters will be set to the original values used when creating the photons object.

Finally, an `events` object is created using the line-of-sight vector, modified observation parameters, and the absorption model:

```
events = photons.project_photons(L,
                                exp_time_new=texp,
                                redshift_new=z,
                                absorb_model=a_mod)
```

`project_photons` draws events uniformly from the `photons` sample, orients their positions in the coordinate frame defined by `L`, and applies the Doppler and cosmological energy shifts, and removes a number of events corresponding to the supplied Galactic absorption model.

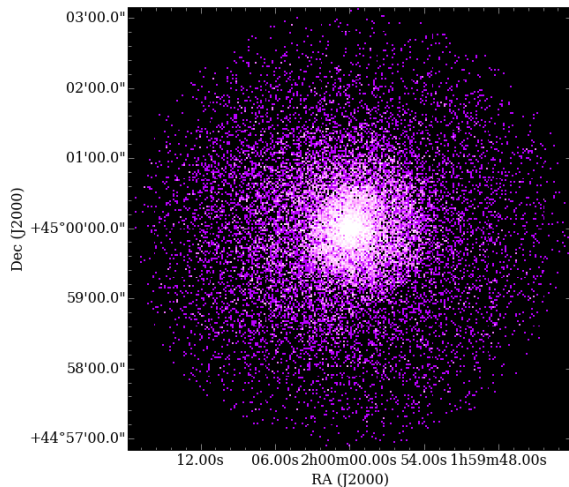


Fig. 3: 100 ks exposure of our simulated galaxy cluster, from a FITS image plotted with `APLpy`.

Step 3: Modeling Instrumental Effects

If desired, instrumental response functions may be supplied to convolve the photons with a particular instrumental model. The files containing these functions are defined and put in a single list `resp`:

```
ARF = "chandra_ACIS-S3_onaxis_arf.fits"
RMF = "chandra_ACIS-S3_onaxis_rmf.fits"
resp = [ARF, RMF]
```

In this case, we would replace our previous call to `project_photons` with one that supplies `resp` as the `responses` argument:

```
events = photons.project_photons(L,
                                exp_time_new=texp,
                                redshift_new=z,
                                absorb_model=a_mod,
                                responses=resp)
```

Supplying instrumental responses is optional. If they are provided, `project_photons` performs 2 additional calculations. If an ARF is provided, the maximum value of the effective area curve will serve as the `area_new` parameter, and after the absorption step a number of events are further removed using the effective area curve as the acceptance/rejection criterion. If an RMF is provided, it will be convolved with the event energies to produce a new array with the resulting spectral channels.

However, if a more accurate simulation of a particular X-ray instrument is needed, or if one wishes to simulate multiple instruments, there are a couple of options for outputting our simulated events to be used by other software that performs such simulations. Since these external packages apply instrument response functions to the events list, the original `events` object generated from the `project_photons` method must not be convolved with instrument responses (e.g., the ARF and RMF) in that step. For input to MARX, we provide an implementation of a MARX "user source" at http://bitbucket.org/jzuhone/yt_marx_source, which takes as input an HDF5 file. The events list can be written in the HDF5 file format with the following method:

```
events.write_h5_file("my_events.h5")
```

Input to SIMX and `Sixte` is handled via `SIMPUP`, a file format designed specifically for the output of simulated X-ray data. The

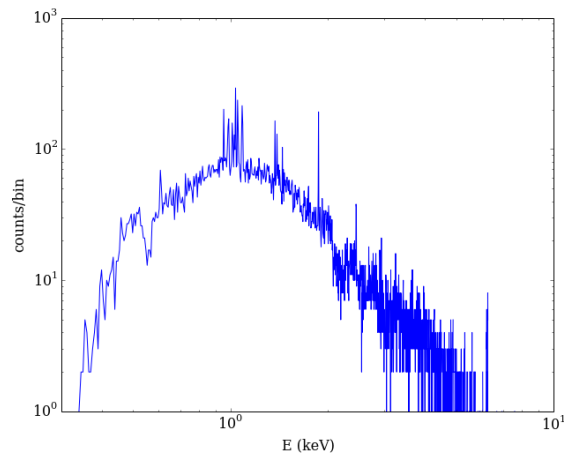


Fig. 4: Spectral energy distribution of our simulated observation.

events list can be written in SIMPUT file format with the following method:

```
events.write_simput_file("my_events",
                        clobber=True,
                        emin=0.1, emax=10.0)
```

where `emin` and `emax` are the energy range in keV of the outputted events. Figure 5 shows several examples of the generated photons passed through various instrument simulations. SIMX and MARX produce FITS event files that are the same format as the data products of the individual telescope pipelines, so they can be analyzed by the same tools as real observations (e.g., XSPEC, CIAO).

Examining the Data

The events may be binned into an image and written to a FITS file:

```
events.write_fits_image("my_image.fits",
                       clobber=True,
                       emin=0.5, emax=7.0)
```

where `emin` and `emax` specify the energy range for the image. Figure 3 shows the resulting FITS image plotted using `APLpy`.

We can also create a spectral energy distribution (SED) by binning the spectrum into energy bins. The resulting SED can be saved as a FITS binary table using the `write_spectrum` method. In this example we bin up the spectrum according to the original photon energy, before it was convolved with the instrumental responses:

```
events.write_spectrum("my_spec.fits",
                     energy_bins=True,
                     emin=0.1, emax=10.0,
                     nchan=2000, clobber=True)
```

here `energy_bins` specifies whether we want to bin the events in unconvolved photon energy or convolved photon channel. Figure 4 shows the resulting spectrum.

Summary

We have developed an analysis module within the Python-based volumetric data analysis toolkit `yt` to construct synthetic X-ray observations of astrophysical sources from simulation datasets, based on the PHOX algorithm. This algorithm generates a large

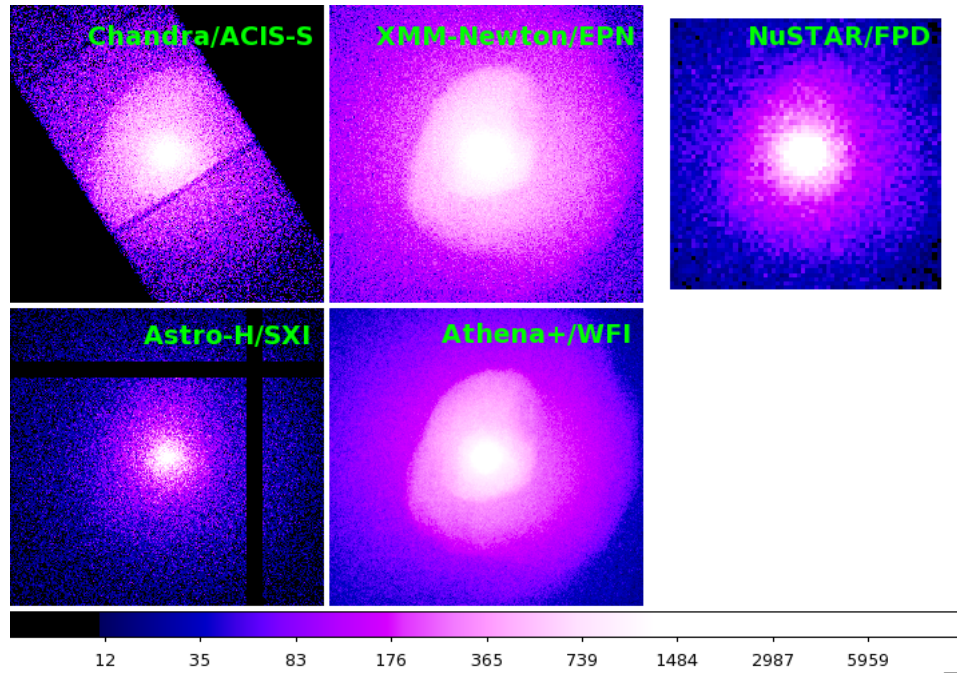


Fig. 5: 100 ks exposures of our simulated galaxy cluster, observed with several different existing and planned X-ray detectors. The Chandra image was made with *MARX*, while the others were made with *SIMX*. All images have the same angular scale.

sample of X-ray photons in the rest frame of the source from the physical quantities of the simulation dataset, and uses these as a sample from which a smaller number of photons are drawn and projected onto the sky plane, to simulate observations with a real detector. The utility of this algorithm lies in the fact that the most expensive step, namely that of generating the photons from the source, need only be done once, and these may be used as a Monte Carlo sample from which to generate as many simulated observations along as many projections and with as many instrument models as desired.

We implement PHOX in Python, using *yt* as an interface to the underlying simulation dataset. Our implementation takes advantage of the full range of capabilities of *yt*, especially its focus on physically motivated representations of simulation data and its support for a wide variety of simulation codes as well as generic *NumPy* array data generated on-the-fly. We also benefit from the object-oriented capabilities of Python as well as the ability to interface with existing astronomical and scientific Python packages.

Our module provides a crucial link between observations of astronomical sources and the simulations designed to represent the objects that are detected via their electromagnetic radiation, enabling some of the most direct testing of these simulations. Also, it is useful as a proposer's tool, allowing observers to generate simulated observations of astrophysical systems, to precisely quantify and motivate the needs of a proposal for observing time on a particular instrument. Our software also serves as a model for how similar modules in other wavebands may be designed, particularly in its use of several important Python packages for astronomy.

REFERENCES

- [Balucinska-Church92] Balucinska-Church, M., & McCammon, D. 1992, *ApJ*, 400, 699
- [Biffi12] Biffi, V., Dolag, K., Böhringer, H., & Lemson, G. 2012, *MNRAS*, 420, 3545
- [Biffi13] Biffi, V., Dolag, K., Böhringer, H. 2013, *MNRAS*, 428, 1395
- [Bulbul14] Bulbul, E., Markevitch, M., Foster, A., et al. 2014, *ApJ*, 789, 13
- [Gardini04] Gardini, A., Rasia, E., Mazzotta, P., Tormen, G., De Grandi, S., & Moscardini, L. 2004, *MNRAS*, 351, 505
- [Heinz11] Heinz, S., Brüggem, M., & Friedman, S. 2011, *ApJS*, 194, 21
- [Komatsu11] Komatsu, E., Smith, K. M., Dunkley, J., et al. 2011, *ApJS*, 192, 18
- [Mewe95] Mewe, R., Kaastra, J. S., & Liedahl, D. A. 1995, *Legacy*, 6, 16
- [Morrison83] Morrison, R. & McCammon, D. 1983, *ApJ*, 270, 119
- [Nagai06] Nagai, D., Vikhlinin, A., & Kravtsov, A. V. 2007, *ApJ*, 655, 98
- [Planck13] Planck Collaboration, Ade, P. A. R., Aghanim, N., et al. 2013, arXiv:1303.5076
- [Raymond77] Raymond, J. C., & Smith, B. W. 1977, *ApJS*, 35, 419
- [Refsdal09] Refsdal et al. *Sherpa: 1D/2D modeling and fitting in Python*. Proceedings of the 8th Python in Science conference (SciPy 2009), G Varoquaux, S van der Walt, J Millman (Eds.), pp. 51-57
- [Smith01] Smith, R. K., Brickhouse, N. S., Liedahl, D. A., & Raymond, J. C. 2001, *ApJL*, 556, L91
- [Turk11] Turk, M. J., Smith, B. D., Oishi, J. S., Skory, S., Skillman, S. W., Abel, T., & Norman, M. L. 2011, *ApJS*, 192, 9
- [Wilms00] Wilms, J., Allen, A., & McCray, R. 2000, *ApJ*, 542, 914
- [ZuHone09] ZuHone, J. A., Ricker, P. M., Lamb, D. Q., & Karen Yang, H.-Y. 2009, *ApJ*, 699, 1004
- [ZuHone14] ZuHone, J. A., Kunz, M. W., Markevitch, M., Stone, J. M., & Biffi, V. 2014, arXiv:1406.4031