SciPy2015

Proceedings of the 14th Python in Science Conference

July 6 - 12 • Austin, Texas

Kathryn Huff
James Bergstra

PROCEEDINGS OF THE 14TH PYTHON IN SCIENCE CONFERENCE

Edited by Kathryn Huff and James Bergstra.

SciPy 2015
Austin, Texas
July 6 - 12, 2015

ISSN:2575-9752
https://doi.org/10.25080/Majora-95ae3ab6-01e

ORGANIZATION

Conference Chair

KELSEY JORDAHL, Enthought, Inc.

Program

NELLE VAROQUAUX, Nelle Varoquaux, Mines ParisTech, Institut Curie, INSERM
SERGE RAY, Arizona State University

Communications

COURTENAY GODSHALL, Enthought Inc.
IRMA KRAMER, PyLadies Austin

Birds of a Feather

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

Proceedings

JAMES BERGSTRA, University of Waterloo
KATHRYN HUFF, University of California at Berkeley

Financial Aid

BEN ROOT, University of Oklahoma
JOHN WIGGINS, XFEL

Tutorials

KRISTEN THYNG, Texas A&M University
JUSTIN VINCENT, Google Inc.

Sprints

DHARHAS POTHINA, US Army Corps of Engineers, ERDC
JONATHAN ROCHER, Enthought Inc.

Sponsors

JILL COWAN, Enthought Inc.

Financial

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

Logistics

JILL COWAN, Enthought Inc.
LEAH JONES, Enthought Inc.

Program Committee

TOM ALDCROFT
FRANCESC ALTED
CHLOE AZENCOTT
CHRIS BARKER
MATTHIAS BUSSONNIER
HOWARD BUTLER
CHRIS CALLOWAY
GREG CAPORASO
ONDREJ CERTIK
AASHISH CHAUDHARY
JEAN CONNELLY
MATT DAVIS

Michael Droettboom
Juan Duque
Daniel Dye
Philip Elson
David Folch
Christopher Fonnesbeck
Emmanuelle Gouillart
Perry Greenfield
Matt Hall
Kathryn Huff
Kyle Kastner
Jackie Kazil
Thomas Kluyver
Karin Lagesen
Jay Laura
Nicholas Lederer
Daniel Lewis
Gilles Louppe
Matthew McCormick
Damon McDougall
Michael McKerns
Andreas Mueller
Ana Nelson
Juan Nunez-Iglesias
Patricia Francis-Lyon
Jeremy Price
Prabhu Ramachandran
Florian Rathgeber
Thomas Robitaille
Jonathan Rocher
Matthew Rocklin
Nicolas Rougier
Dan Schult
Skipper Seabold
John Stachurski
Philip Stephens
Erik Tollerud
Stefan van der Walt
Gael Varoquaux
Shaun Walbridge
Christopher Wilmer
Tony Yu
Xiwei Zhang
Tiziano Zito
Alex Zvoleff

Proceedings Reviewers

Tom Aldcroft
Ankur Ankan
Yoshiki Vazquez Baeza
Kyle Barbary
Eric Battenberg
James Bednar
Trevor Bekolay
Sebastian Benthall
James Bergstra
Matthew Brett
Matthias Bussonnier
Azalee Bostroem
Chris Calloway

AASHISH CHAUDHARY
CHRISTINE CHOIRAT
JEAN CONNELLY
ALLEN DOWNEY
MICHAEL DROETTBOOM
DAN ELLIS
ADINA CHUANG HOWE
ALMAR KLEIN
THOMAS KLUYVER
PASCAL LAMBLIN
HANS PETTER LANGTANGEN
NICHOLAS LEDERER
DAVID LIPPA
ANDREAS MUELLER
MICHAEL PACER
ABINASH PANDA
JOSEF PERKTOLD
COLLIN RAFFEL
MIN RAGAN-KELLEY
PRABHU RAMACHANDRAN
MATTHEW ROCKLIN
PHILIPP RUDIGER
SKIPPER SEABOLD
ASHTON SHORTRIDGE
NATHANIEL SMITH
WILLIAM SPOTZ
ERIK TOLLERUD
JAKE VANDERPLAS
STEFAN VAN DER WALT
ALEJANDRO WEINSTEIN
TIZIANO ZITO

Mini Symposium Committee

JAKE VANDERPLAS, Astronomy and Astrophysics
GREG CAPORASO, Computational Life and Medical Sciences
LORENA BARBA, Engineering
MATT HALL, Geophysics
CARSON FARMER, GIS
CHRIS BARKER, Oceanography and Meteorology
DAMON MCDOUGALL, Vision, Visualization, and Imaging

Tutorial Review Committee

SARA SAFAVI, Rackspace Inc.
JEREMY PRICE, Rackspace Inc.
DHARHAS POTHINA, US Army Engineer Research and Development Center
ARON AHMADIA, Continuum Analytics

Diversity Committee

PATRICIA FRANCIS-LYON, University of San Francisco
LEAH SILEN, NumFOCUS
CINDY SRIDHARAN,
MATT DAVIS, Autodesk
APRIL WRIGHT, University of Texas at Austin

SPONSORED ATTENDEES

AZALEE BOSTROEM, University of California - Davis
PILAR BRIST, University of Texas - Tyler
LUKE CAMPAGNOLA, VisPy
LESLEY CHAPMAN, University of Rochester Medical Center
ROBERTO COLISTETE JUNIOR, Federal University of Espirito Santo
YANNICK CONGO, EDSPI France
MELLISSA CROSS, University of Minnesota
FILIPE FERNANDES, SECOORA
GABRIEL GRANT, RepsWith.us
HARSH GUPTA, SymPy
IAN HENRIKSEN, Brigham Young University
JAIME HUERTA-CEPAS, European Molecular Biology Laboratory
KYLE KASTNER, Universite de Montreal
JULIANA LEONEL, Universidade Federal de Bahia
ERIC MA, Massachusetts Institute of Technology
SHAYLYN SCOTT, George Mason University
LIZ STRECKERT, PyLadies
JORDI TORRENTS, NetworkX
DAVID URBINA, University of Texas at Dallas
ALEXANDER VOSTRIKOV, University of Chicago
DAVID WARDE-FARLEY, Universite de Montreal
JOSHUA WARNER, Mayo Clinic
AVANI WILDANI, The Salk Institute for Biological Studies
LEVI WOLF, Arizona State University
CHUAN YANG, Shengjing Hospital of China Medical University
AMY PRAGER, Girl Scouts of Eastern Mass.
DANA ENGEBRETSON, University of Minnesota
LEENA P, PyLadies
NIKHIL HAAS, University of San Francisco

CONTENTS

Will Millennials Ever Get Married?

Allen B. Downey[‡*]

https://www.youtube.com/watch?v=XHYFNraQEEo

Abstract—Using data from the National Survey of Family Growth (NSFG), we investigate marriage patterns among women in the United States We describe and predict age at first marriage for successive generations based on decade of birth. The fraction of women married by age 22 has dropped by 11 percentage points per decade, from 69% for women born in the 1940s to 13% for women born in the 90s. The fraction of women married by age 42 fell more slowly, from 93% for women born in the 40s to 82% for women born in the 70s. Projections suggest that this fraction will be substantially lower for later generations, between 68% and 72%. Along with these results, this paper presents an introduction to survival analysis methods and an implementation in Python.

Keywords—Survival analysis, marriage patterns, Python.

Introduction

A recent study from the Pew Research Center [Wan14] reports that the fraction of adults in the U.S. who have never married is increasing. Between 1960 and 2012, the fraction of men 25 and older who had never married increased from 10% to 23%. The corresponding fraction of women increased from 8% to 17%. The Pew study focuses on the causes of these trends, but does not address this question: is the fraction of people who never marry increasing, are people marrying later, or both? That is the subject of this paper.

To answer this question, we apply tools of survival analysis to data from the National Survey of Family Growth (NSFG). Since 1973 the U.S. Centers for Disease Control and Prevention (CDC) have conducted this survey, intended to gather "information on family life, marriage and divorce, pregnancy, infertility, use of contraception, and men's and women's health." See http://cdc.gov/nchs/nsfg.htm.

NSFG data is organized in cycles; during each cycle several thousand respondents were interviewed, including women ages 14–44. Men were included starting with Cycle 6 in 2002, but for this study we use only data from female respondents.

Table 1 shows the interview dates for each cycle, the number of respondents, and the birth years of the respondents. We did not use data from Cycles 1 and 2 because they included only married women. The total sample size for this study is 52 789.

* *Corresponding author: allen.downey@olin.edu*
‡ *Olin College of Engineering*

Cycle	Interview dates	Number of respondents	Birth years
3	1982–83	7 969	1937–68
4	1988–88	8 450	1943–73
5	1995	10 847	1950–80
6	2002–03	7 643	1957–88
7	2006–10	12 279	1961–95
8	2011–13	5 601	1966–98

TABLE 1: *NSFG Survey Cycles*

For each respondent we have date of birth (year and month), date of interview, and date of first marriage, if applicable. So we can compute with resolution of one month each respondent's age at interview, `age`, and age at first marriage, `agemarry`.

To study changes in marriage patterns over time, we group the respondents into cohorts by decade of birth. For each cohort, Table 2 reports the number of respondents, range of ages when they were interviewed, number who had been married at least once at time of interview, and the number of married respondents whose date of marriage was not ascertained.

Cohort 30 includes women born in the 1930s, and so on for the other cohorts. One goal of this paper is to describe and predict marriage patterns for the Millennial Generation, defined here to include women born in the 1980s and 90s.

Another goal of this paper is to present survival analysis and its implementation in Python to an audience that may not be familiar with it. We also describe the resampling methods we use to deal with the stratified sampling design of the NSFG.

The code and data for this project are available in a public Git repository at https://github.com/AllenDowney/MarriageNSFG.

Cohort	Number of respondents	Age at interview	Number married	Number with missing data
30	325	42–44	310	0
40	3 608	32–44	3275	0
50	10 631	22–44	8658	10
60	14 484	15–44	8421	27
70	12 083	14–43	5908	25
80	8 536	14–33	2203	8
90	3 122	15–23	93	0

TABLE 2: *NSFG Birth Cohorts*

Methodology

Survival analysis

Survival analysis is a powerful set of tools with applications in many domains, but it is often considered a specialized topic.

Survival analysis is used to study and predict the time until an event: in medicine, the event might be the death of a patient, hence "survival"; but more generally we might be interested in the time until failure of a mechanical part, the lifetimes of civilizations, species, or stars; or in this study the time from birth until first marriage.

The result of survival analysis is often a **survival function**, which shows the fraction of the population that survives after t, for any time, t. If T is a random variable that represents the time until an event, the survival function, $S(t)$, is the probability that T exceeds t:

$$S(t) \equiv \Pr(T > t)$$

If the distribution of T is known, or can be estimated from a representative sample, computing $S(t)$ is simple: it is the complement of the cumulative distribution function (CDF):

$$S(t) = 1 - \mathrm{CDF}_T(t)$$

In Python we can compute the survival function like this:

```
from collections import Counter
import numpy as np

def MakeSurvivalFunction(values):
    counter = Counter(values)
    ts, fs = zip(*sorted(counter.items()))
    ts = np.asarray(ts)
    ps = np.cumsum(fs, dtype=np.float)
    ps /= ps[-1]
    ss = 1 - ps
    return SurvivalFunction(ts, ss)
```

`values` is a sequence of observed lifetimes. Counter makes a map from each unique value to the number of times it appears, which we split into a sorted sequence of times, `ts`, and their frequencies, `fs`.

We convert ts to a NumPy array [Wal11]. Then `ps` is the cumulative sum of the frequencies, normalized to go from 0 to 1, so it represents the CDF of the observed values. `ss`, which is the complement of `ps`, is the survival function.

`SurvivalFunction` is defined in `marriage.py`, a Python module we wrote for this project.

Given a survival curve, we can compute the **hazard function**, which is the instantaneous death rate at time t; that is, the fraction of people who survive until time t and then die at time t. When t is continuous, the hazard function, $\lambda(t)$, is

$$\lambda(t) = -S'(t)/S(t)$$

Where $S'(t)$ is the derivative of $S(t)$. Since the survival function decreases monotonically, its derivative is nonpositive, so the hazard function is nonnegative.

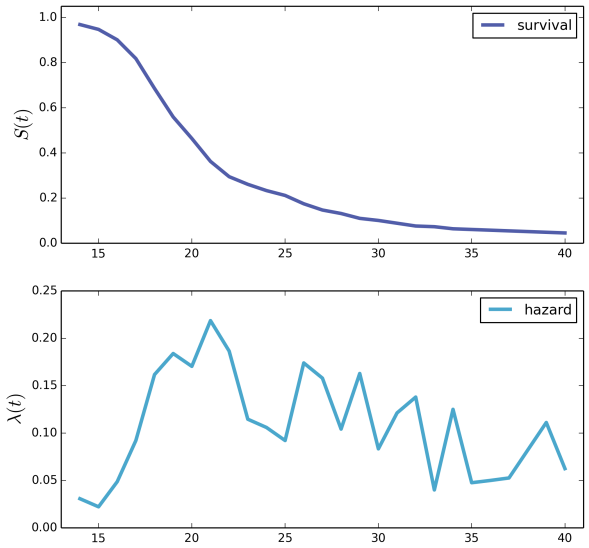


Fig. 1: *Survival and hazard functions for 1930s cohort.*

With a survival function represented by discrete `ts` and `ss`, we can compute the hazard function like this:

```
import pandas as pd

# class SurvivalFunction
def MakeHazardFunction(self):
    lams = pd.Series(index=self.ts)
    prev = 1.0
    for t, s in zip(self.ts, self.ss):
        lams[t] = (prev - s) / prev
        prev = s
    return HazardFunction(lams)
```

`MakeHazardFunction` is a method of `SurvivalFunction`, which provides attributes `ts` and `ss`. The result, `lams`, is a Pandas Series [McK10] object that maps from the same set of `ts` to the estimated hazard function, $\lambda(t)$.

Figure 1 shows the survival and hazard functions for women born in the 1930s. These women were interviewed when they were 42–44 years old. At that point more than 95% of them had been married; for the others we set age at marriage to infinity (`np.inf`). In this cohort, the hazard function is highest at ages 18–22, and lower as age increases.

This example demonstrates the simple case, where the respondents are the same age and most events are complete. But for most applications of survival analysis, the sample also includes incomplete events. For example, the 1960s cohort includes women from ages 14–44; for the ones that are not married, we don't know when they will marry, if ever. These missing data are said to be "censored".

It might be tempting to ignore unmarried women and compute the survival function for women whose ages at marriage

are known. But that would discard useful information and seriously bias the results.

For women who are not married yet, their age at interview is a lower bound on their age at marriage. We can use both groups to estimate the hazard function, then compute the survival function. One common way to do that is Kaplan-Meier estimation.

The fundamental idea is that at each time, t, we know the number of events that occurred and the number of respondents who were "at risk"; that is, known to to be unmarried. The ratio of these factors estimates the hazard function.

Initially, the entire sample is considered at risk. At each time step, we subtract people who got married at age t as well as people who were interviewed at age t (and therefore no longer in the observation pool at the next time step). The following function implements this algorithm:

```
def EstimateHazardFunction(complete, ongoing):
    hist_complete = Counter(complete)
    hist_ongoing = Counter(ongoing)

    ts = list(hist_complete | hist_ongoing)
    ts.sort()

    at_risk = len(complete) + len(ongoing)

    lams = pd.Series(index=ts)
    for t in ts:
        ended = hist_complete[t]
        censored = hist_ongoing[t]

        lams[t] = ended / at_risk
        at_risk -= ended + censored

    return HazardFunction(lams)
```

`complete` is a sequence of lifetimes for complete events, in this case age at marriage. `ongoing` is a sequence of lower bounds for incomplete observations, in this case age at interview.

`hist_complete` counts how many respondents were married at each age; `hist_ongoing` counts how many unmarried respondents were interviewed at each age.

`ts` is a sorted list of observation times, which is the union of unique values from complete and ongoing.

`at_risk` is the number of respondents at risk; initially it is the total number of respondents.

`lams` is a Pandas Series that maps from each observation time to the estimated hazard rate.

For each value of `t` we look up `ended`, which is the number of people married for the first time at `t`, and `censored`, which is the number of never married people interviewed at `t`. The estimated hazard function at `t` is the ratio of `ended` and `at_risk`.

At the end of each time step, we update `at_risk` by subtracting off `ended` and `censored`.

The result is a HazardFunction object that contains the Series `lams` and provides methods to access it.

With this estimated HazardFunction, we can compute the SurvivalFunction. The hazard function, $\lambda(t)$, is the probability of ending at time t conditioned on surviving until t. Therefore, the probability of surviving until t is the cumulative product of the complementary hazard function:

$$S(t) = \prod_{t_i<t} [1 - \lambda(t_i)]$$

Here's the Python implementation:

```
# class HazardFunction
def MakeSurvival(self):
    series = (1 - self.series).cumprod()
    ts = series.index.values
    ss = series.values
    return SurvivalFunction(ts, ss)
```

We wrote our own implementation of these methods in order to demonstrate the methodology, and also to make them work efficiently with the resampling methods described in the next section. But Kaplan-Meier estimation and other survival analysis algorithms are also available in a Python package called Lifelines [Dav15].

Resampling

The NSFG is intended to be representative of the adult U.S. population, but it uses stratified sampling to systematically oversample certain subpopulations, including teenagers and racial minorities. Our analysis takes this design into account to generate results that are representative of the population.

As an example of stratified sampling, suppose there are 10 000 people in the population you are studying, and you sample 100. Each person in the sample represents 100 people in the population, so each respondent has the same "sampling weight".

Now suppose there are two subgroups, a minority of 1 000 people and a majority of 9 000. A sample of 100 people will have 10 members of the minority group, on average, which might not be enough for reliable statistical inference.

In a stratified sample, you might survey 40 people from the minority group and only 60 from the majority group. This design improves some statistical properties of the sample, but it changes the weight associated with each respondent. Each of the 40 minorities represents $1000/40 = 25$ people in the population, while each of the 60 others represents $9000/60 = 150$ people. In general, respondents from oversampled groups have lower weights.

The NSFG includes a computed weight for each respondent, which indicates how many people in the U.S. population she represents. Some statistical methods, like regression, can be extended to take these weights into account, but in general it is not easy.

However, bootstrapping provides a simple and effective approach. The idea behind bootstrapping is to use the actual sample as a model of the population, then simulate the results of additional experiments by drawing new samples (with replacement) from the actual sample.

With stratified sampling, we can modify the bootstrap process to take sampling weights into account. The following function performs weighted resampling on the NSFG data:

```
import thinkstats2

def ResampleRowsWeighted(df):
    weights = df.finalwgt
    cdf = thinkstats2.Cdf(dict(weights))
    indices = cdf.Sample(len(weights))
    sample = df.loc[indices]
    return sample
```

`df` is a Pandas DataFrame with one row per respondent; it includes a column that contains sampling weights, called `finalwgt`.

`weights` is a Series that maps from respondent index to sampling weight. `cdf` represents a cumulative distribution function that maps from each index to its cumulative probability. The Cdf class is provided by `thinkstats2.py`, a module that accompanies the second edition of *Think Stats* [Dow14]. We use it here because it provides an efficient implementation of random sampling from an arbitrary distribution.

`Sample` generates a random sample of indices based on the sampling weights. The return value, `sample`, is a Pandas DataFrame that contains the selected rows. Since the sample is generated with replacement, some respondents might appear more than once; others might not appear at all.

After resampling, we jitter the data by adding Gaussian noise (mean 0, standard deviation 1 year) to each respondent's age at interview and age at marriage. Jittering contributes some smoothing, which makes the figures easier to interpret, and some robustness, making the results less prone to the effect of a small number of idiosyncratic data points.

Jittering also makes sense in the context of bootstrapping. Each respondent in the sample represents several thousand people in the population; it is reasonable to assume that there is variation within each represented subgroup.

Finally, we discretize age at interview and age at marriage, rounding down to integer values.

Results

Figure 2 shows the estimated survival curve for each cohort (we omit the 1930s cohort because it only includes people born after 1936, so it is not representative of the decade). The lines show the median of 101 resampling runs; the gray regions show 90% confidence intervals.

Two trends are apparent in this figure: women are getting married later, and the fraction of women who remain unmarried is increasing.

Table 3 shows the percentage of married women in each cohort at ages 22, 32, and 42 (which are the last observed ages for cohorts 90, 80, and 70).

Two features of this data are striking:

- By age 22, only 13% of the 90s cohort have been married, contrasted with 69% of the 40s cohort. Between these cohorts, the fraction of women married by age 22 dropped more than 11 percentage points per decade.
- By age 32, only 60% of the 80s cohort is married, and their survival curve seems to have gone flat. In this cohort, 259 were at risk at age 30, and only 9 were married that year; 155 were at risk at age 31, and none were married; 63 were are risk at age 32, and again none were married. These low hazard rates are strange, but they are based on sample sizes large enough that it is hard to dismiss them.

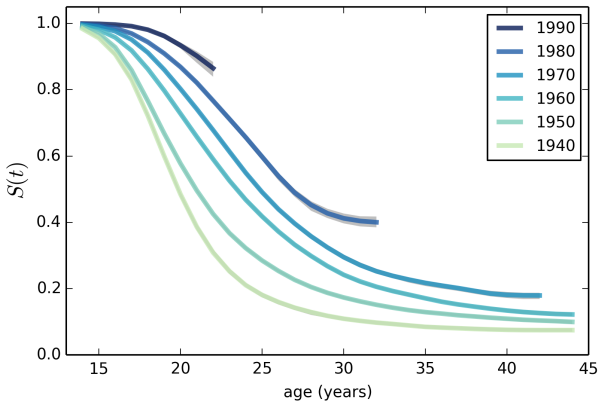


Fig. 2: *Survival functions by birth cohort.*

Cohort	% married by age		
	22	32	42
40	69	90	92
50	57	85	90
60	41	79	87
70	32	75	82
80	23	60	–
90	13	–	–

TABLE 3: *Marriage rates by birth cohort and age.*

Projection

Predicting these kinds of social trends is nearly futile. We can use current trends to generate projections, but in general there is no way to know which trends will continue and which will decrease or reverse.

As we saw in the previous section, the 80s cohort seems to be on strike, with unprecedented low marriage rates in their early thirties. Visual extrapolation of their survival curve suggests that 40% of them will remain unmarried, more than double the fraction of previous generations.

At the same time the number of women getting married at ages 35–45 has been increasing for several generations, so we might expect that trend to continue. In that case the gap between the 80s and 70s cohorts would close.

These prediction methods provide a rough upper and lower bound on what we might expect. A middle ground is to assume that the hazard function from the previous generation will apply to the next.

This method predicts higher marriage rates than extrapolating the survival curves because it takes into account the structure of the model: because fewer women married young, more are at risk at later ages, so we expect more late marriages.

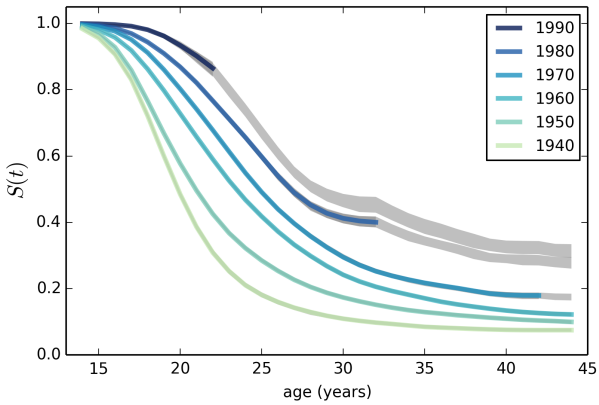


Fig. 3: *Survival functions with projections.*

To make these projections, we extend each HazardFunction using data from the previous cohort:

```
# class HazardFunction
def Extend(self, other):
    last_t = self.series.index[-1]
    other_ts = other.series.index
    hs = other.series[other_ts > last_t]
    self.series = pd.concat([self.series, hs])
```

Then we convert the extended hazard functions to survival functions using `HazardFunction.MakeSurvival`.

Figure 3 shows the results. Again, the gray regions show 90% confidence intervals. For the 80s cohort, the median projection is that 72% will marry by age 42, down from 82% in the previous cohort.

For the 90s cohort, the median projection is that only 68% will marry by age 42. This projection assumes that this cohort will also go on a "marriage strike" in their early thirties, but this event might not be repeated.

Discussion

The previous section addresses the title question of this paper, "Will Millennials Ever Get Married?" Our projections suggest that the fraction still unmarried at age 42 will be higher than in previous generations, by about 10 percentage points, unless there is a substantial increase in the hazard rate after age 30.

We also investigate how much of the change in marriage rates is driven by two factors: people getting married later, or never getting married at all. Up through the 70s cohort, people were getting married later, but the fraction who never married was increasing only slowly. Among Millennials (women born in the 80s and 90s), the fraction of people marrying young is continuing to fall, but we also see indications that the fraction of people who never marry is increasing more quickly.

Future work

This work is preliminary, and there are many avenues for future investigation:

- The NSFG includes data from male respondents, starting with Cycle 6 in 2002. We plan to repeat our analysis for these men.
- There are many subgroups in the U.S. that would be interesting to explore, including different regions, education and income levels, racial and religious groups.
- We have data from the Canadian General Social Survey, which will allow us to compare marriage patterns between countries (see http://tinyurl.com/canadagss).
- We are interested in finding similar data from other countries.

Acknowledgment

Many thanks to Lindsey Vanderlyn for help with data acquisition, preparation, and analysis. And thanks to the SciPy reviewers who made many helpful suggestions.

References

[Dow14] Allen Downey, *Think Stats: Exploratory Data Analysis*, 2nd edition, O'Reilly Media, October 2014. http://thinkstats2.com

[Dav15] Cameron Davidson-Pilon, Lifelines, (2015), Github repository, https://github.com/CamDavidsonPilon/lifelines

[McK10] Wes McKinney. "Data Structures for Statistical Computing in Python", *Proceedings of the 9th Python in Science Conference*, 51-56 (2010) http://pandas.pydata.org.

[Wal11] 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) http://www.numpy.org

[Wan14] Wendy Wang and Kim Parker, "Record Share of Americans Have Never Married", Washington D.C.: Pew Research Center's Social and Demographic Trends project, September 2014. http://tinyurl.com/wang14pew

pgmpy: Probabilistic Graphical Models using Python

Ankur Ankan*, Abinash Panda

https://www.youtube.com/watch?v=Vcmjqx7lht0

Abstract—Probabilistic Graphical Models (PGM) is a technique of compactly representing a joint distribution by exploiting dependencies between the random variables. It also allows us to do inference on joint distributions in a computationally cheaper way than the traditional methods. PGMs are widely used in the field of speech recognition, information extraction, image segmentation, modelling gene regulatory networks.

pgmpy [pgmpy] is a python library for working with graphical models. It allows the user to create their own graphical models and answer inference or map queries over them. pgmpy has implementation of many inference algorithms like VariableElimination, Belief Propagation etc.

This paper first gives a short introduction to PGMs and various other python packages available for working with PGMs. Then we discuss about creating and doing inference over Bayesian Networks and Markov Networks using pgmpy.

Index Terms—Graphical Models, Bayesian Networks, Markov Networks, Variable Elimination

Introduction

Probabilistic Graphical Model (PGM) is a technique of representing Joint Distributions over random variables in a compact way by exploiting the dependencies between them. PGMs use a network structure to encode the relationships between the random variables and some parameters to represent the joint distribution.

There are two major types of Graphical Models: Bayesian Networks and Markov Networks.

Bayesian Network: A Bayesian Network consists of a directed graph and a conditional probability distribution associated with each of the random variables. A Bayesian network is used mostly when there is a causal relationship between the random variables. An example of a Bayesian Network representing a student [student] taking some course is shown in Fig 1.

Markov Network: A Markov Network consists of an undirected graph and a few Factors are associated with it. Unlike Conditional Probability Distributions, a Factor does not represent the probabilities of variables in the network; instead it represents the compatibility between random variables that is how much a particular state of a random variable likely to agree with the another state of some other random variable. An example of markov [markov] network over four friends A, B, C, D agreeing to some concept is shown in Fig 2.

There are numerous open source packages available in Python for working with graphical models. eBay's bayesian-belief-networks [bbn] mostly focuses on Bayesian Models and has implementation of a limited number of inference algorithms. Another package pymc [pymc] focuses mainly on Markov Chain Monte Carlo (MCMC) method. libpgm [libpgm] also mainly focuses on Bayesian Networks.

pgmpy tries to be a complete package for working with graphical models and gives the user full control on designing the model. The source code is very well documented with proper docstrings and doctests for each method so that users can quickly get upto speed. Furthermore, pgmpy also provides easy extensibility allowing users to write their own inference algorithms or elimination order algorithms without any additional effort to get familiar with the source code.

* *Corresponding author: ankurankan@gmail.com*

Getting Source Code and Installing

pgmpy is released under MIT Licence and is hosted on github. We can simply clone the repository and install it:

```
git clone https://github.com/pgmpy/pgmpy
cd pgmpy
[sudo] python3 setup.py install
```

Dependencies: pgmpy runs only on python3 and is dependent on networkx, numpy, pandas and scipy which can be installed using pip or conda as:

```
pip install -r requirements.txt
```

or:

```
conda install --file requirements.txt
```

Creating Bayesian Models using pgmpy

A Bayesian Network consists of a directed graph where nodes represents random variables and edges represent the the relation between them. It is parameterized using Conditional Probability Distributions(CPD). Each random variable in a Bayesian Network has a CPD associated with it. If a random varible has parents in the network then the CPD represents $P(var|Par_{var})$ i.e. the probability of that variable given its parents. In the case, when the random variable has no parents it simply represents $P(var)$ i.e. the probability of that variable.

For example, we can take the case of student model represented in Fig 1. A possible CPD for the random variable grade is shown in Table 1.

We can represent the CPD shown in Table 1 in pgmpy as follows:

```
from pgmpy.factors import TabularCPD
grade_cpd = TabularCPD(
```

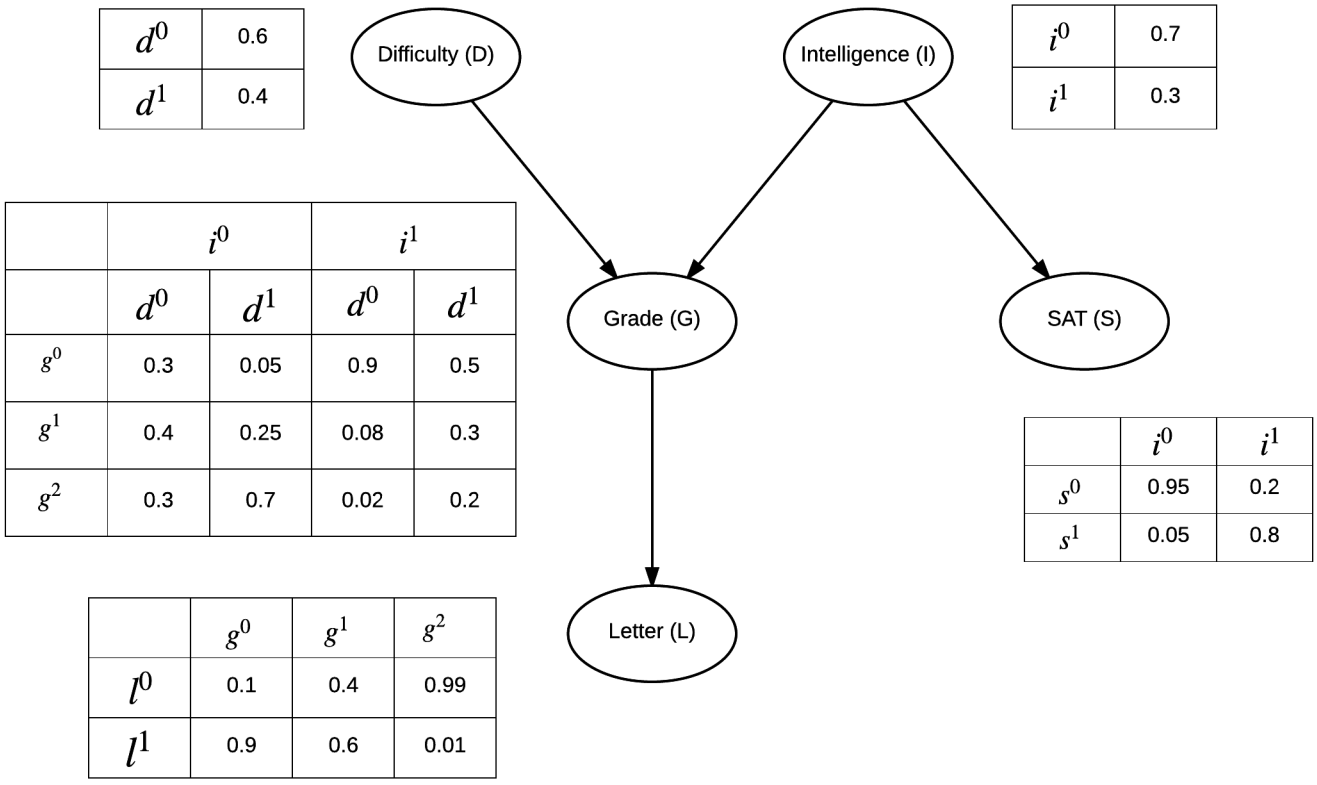


Fig. 1: *Student Model: A simple Bayesian Network.*

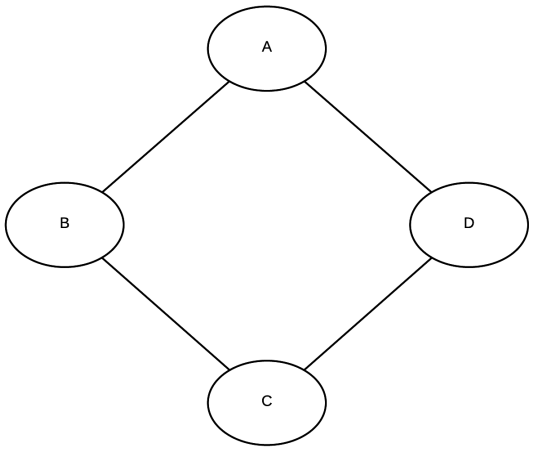


Fig. 2: *A simple Markov Model*

Intelligence (I) **Difficulty (D)**	i^0 d^0	i^0 d^1	i^1 d^0	i^1 d^1
g^0	0.3	0.05	0.9	0.5
g^1	0.4	0.25	0.08	0.3
g^2	0.3	0.7	0.02	0.2

TABLE 1: *Conditional Probability Table.*

A	**B**	$\phi(A,B)$
a^0	b^0	30
a^0	b^1	5
a^1	b^0	1
a^1	b^1	10

TABLE 2: *Factor over variables A and B.*

```
                variable='G',
                variable_card=3,
                values=[[0.3, 0.05, 0.9, 0.5],
                        [0.4, 0.25, 0.08, 0.3],
                        [0.3, 0.7, 0.02, 0.2]],
                evidence=['I', 'D'],
                evidence_card=[2, 2])
```

Now, coming back to defining a model using pgmpy. The general workflow for defining a model in pgmpy is to first define the network structure and then add the parameters to it. We can create the student model shown in Fig 1 in pgmpy as follows:

```
from pgmpy.models import BayesianModel
from pgmpy.factors import TabularCPD
student_model = BayesianModel([('D', 'G'),
                               ('I', 'G'),
                               ('G', 'L'),
                               ('I', 'S')])
grade_cpd = TabularCPD(
                variable='G',
                variable_card=3,
                values=[[0.3, 0.05, 0.9, 0.5],
                        [0.4, 0.25, 0.08, 0.3],
                        [0.3, 0.7, 0.02, 0.2]],
                evidence=['I', 'D'],
                evidence_card=[2, 2])
difficulty_cpd = TabularCPD(
                variable='D',
                variable_card=2,
                values=[[0.6, 0.4]])
intel_cpd = TabularCPD(
                variable='I',
                variable_card=2,
                values=[[0.7, 0.3]])
letter_cpd = TabularCPD(
                variable='L',
                variable_card=2,
                values=[[0.1, 0.4, 0.99],
                        [0.9, 0.6, 0.01]],
                evidence=['G'],
                evidence_card=[3])
sat_cpd = TabularCPD(
                variable='S',
                variable_card=2,
                values=[[0.95, 0.2],
                        [0.05, 0.8]],
                evidence=['I'],
                evidence_card=[2])
student_model.add_cpds(grade_cpd, difficulty_cpd,
                       intel_cpd, letter_cpd,
                       sat_cpd)
```

The network structure of a Graphical Model encodes the independence conditions between the random variables. pgmpy also has methods to determine the local independencies, D-Separation, converting to a markov model etc. A few example are shown below:

```
student_model.get_cpds()
[<TabularCPD representing P(G:3 | I:2, D:2)
                              at 0x7f196c0b27b8>,
 <TabularCPD representing P(D:2) at 0x7f196c0b2828>,
 <TabularCPD representing P(I:2) at 0x7f196c0b2908>,
 <TabularCPD representing P(L:2 | G:3)
                              at 0x7f196c0b2978>,
 <TabularCPD representing P(S:2 | I:2)
                              at 0x7f196c0b27f0>]

student_model.active_trail_nodes('D')
{'D', 'G', 'L'}

student_model.local_independencies('G')
(G _|_ S | D, I)

student_model.get_independencies()
(S _|_ I, G, L | D)
(S _|_ D, I | G)
(S _|_ D, I, G | L)
(D _|_ G, L | S)
(D _|_ I, S | G)
(D _|_ G, L | I)
(D _|_ G, I, S | L)
(G _|_ D, I, L | S)
(G _|_ I, L, S | D)
(G _|_ D, L | I)
(G _|_ D, I, S | L)
(I _|_ G, L | S)
(I _|_ G, S, L | D)
(I _|_ D, S | G)
(I _|_ D, G, S | L)
(L _|_ D, G, I | S)
(L _|_ G, I, S | D)
(L _|_ D, G | I)

student_model.to_markov_model()
<pgmpy.models.MarkovModel.MarkovModel
                              at 0x7f196c0b2470>
```

Creating Markov Models in pgmpy

A Markov Network consists of an undirected graph which connects the random variables according to the relation between them. A markov network is parameterized by factors which represent the likelihood of a state of one variable to agree with some state of other variable.

We can take the example of a Factor over variables A and B in the network shown in Fig 2. A possible Factor over variables A and B is shown in Table 2.

We can represent this Factor in pgmpy as follows:

```
from pgmpy.factors import Factor
phi_a_b = Factor(varibales=['A', 'B'],
                 cardinality=[2, 2],
                 value=[100, 5, 5, 100])
```

Assuming some other possible factors as in Table 3, 4 and 5, we can define the complete markov model as:

```
from pgmpy.models import MarkovModel
from pgmpy.factors import Factor
model = MarkovModel([('A', 'B'), ('B', 'C'),
                     ('C', 'D'), ('D', 'A')])
factor_a_b = Factor(variables=['A', 'B'],
```

C	$\phi(B,C)$	
b^0	c^0	100
b^0	c^1	1
b^1	c^0	1
b^1	c^1	100

TABLE 3: *Factor over variables B and C.*

C	D	$\phi(C,D)$
c^0	d^0	1
c^0	d^1	100
c^1	d^0	100
c^1	d^1	1

TABLE 4: *Factor over variables C and D.*

```
                    cardinality=[2, 2],
                    value=[100, 5, 5, 100])
factor_b_c = Factor(variables=['B', 'C'],
                    cardinaity=[2, 2],
                    value=[100, 3, 2, 4])
factor_c_d = Factor(variables=['C', 'D'],
                    cardinality=[2, 2],
                    value=[3, 5, 1, 6])
factor_d_a = Factor(variables=['D', 'A'],
                    cardinality=[2, 2],
                    value=[6, 2, 56, 2])
model.add_factors(factor_a_b, factor_b_c,
                  factor_c_d, factor_d_a)
```

Similar to Bayesian Networks, pgmpy also has the feature for computing independencies, converting to Bayesian Network etc in the case of Markov Networks.

```
model.get_local_independencies()
(D _|_ B | C, A)
(C _|_ A | D, B)
(A _|_ C | D, B)
(B _|_ D | C, A)

model.to_bayesian_model()
<pgmpy.models.BayesianModel.BayesianModel
                            at 0x7f196c084320>

model.get_partition_function()
10000
```

Doing Inference over models

pgmpy support various Exact and Approximate inference algorithms. Generally, to perform inference over models, we need to first create an inference object by passing the model to the inference class. Once an inference object is instantiated, we can call either query method to find the probability of some variable given evidence, or else map_query method to know the state of the variable having maximum probability. Let's perform inference on the student model (Fig 1) using variable elimination :

D	A	$\phi(D,A)$
d^0	a^0	100
d^0	a^1	1
d^1	a^0	1
d^1	a^1	100

TABLE 5: *Factor over variables D and A.*

```
from pgmpy.inference import VariableElimination
student_infer = VariableElimination(student_model)
prob_G = student_infer.query(variables='G')
print(prob_G['G'])
G       phi(G)
G_0     0.4470
G_1     0.2714
G_2     0.2816

prob_G = student_infer.query(
                variables='G',
                evidence=[('I', 1), ('D', 0)])
print(prob_G['G'])
G       phi(G)
G_0     0.0500
G_1     0.2500
G_2     0.7000

student_infer.map_query(variables='G')
{'G': 0}

student_infer.map_query(
                variables='G',
                evidence=[('I', 1), ('D', 0)])
{'G': 2}
```

Fit and Predict Methods

In a general machine learning task we are given some data from which we want to compute the parameters of the model. pgmpy simplifies working on these problems by providing fit and predict methods in the models. fit method accepts the given data as a pandas DataFrame object and learns all the parameters from it. The predict method also accepts a pandas DataFrame object and predicts values of all the missing variables using the model. An example of fit and predict over the student model using some randomly generated data:

```
from pgmpy.models import BayesianModel
import pandas as pd
import numpy as np

# Considering that each variable have only 2 states,
# we can generate some random data.
raw_data = np.random.randint(low=0,
                             high=2,
                             size=(1000, 5))
data = pd.DataFrame(raw_data,
                    columns=['D', 'I', 'G',
                             'L', 'S'])
data_train = data[: int(data.shape[0] * 0.75)]

student_model = BayesianModel([('D', 'G'),
                               ('I', 'G'),
                               ('I', 'S'),
                               ('G', 'L')])
student_model.fit(data_train)
student_model.get_cpds()
[<TabularCPD representing P(C:2) at 0x7f195ee5e400>,
 <TabularCPD representing P(A:2) at 0x7f195ee5e518>,
 <TabularCPD representing P(D:2) at 0x7f195ee5e2b0>,
 <TabularCPD representing P(F:2) at 0x7f195ee5e320>,
 <TabularCPD representing P(P:2 | F:2, A:2, L:2)
                                 at 0x7f195ed620f0>,
 <TabularCPD representing P(L:2 | C:2, D:2)
                                 at 0x7f195ed62048>]

data_test = data[0.75 * data.shape[0] : data.shape[0]]
```

```
data_test.drop('P', axis=1, inplace=True)
student_model.predict(data_test)
     P
750  0
751  0
752  1
753  0
..  ..
996  0
997  0
998  0
999  0

[250 rows x 1 columns]
```

Extending pgmpy

One of the main features of pgmpy is its extensibility. It has been built in a way so that new algorithms can be directly written without needing to get familiar with the code base.

For example, for writing any new inference algorithm we can simply inherit the Inference class. Inheriting this base inference class exposes three variables to the class: self.variables, self.cardinalities and self.factors; using these variables we can write our own inference algorithm. An example is shown:

```
from pgmpy.inference import Inference
class MyNewInferenceAlgo(Inference):
    def print_variables(self):
        print('variables: ', self.variables)
        print('cardinality: ', self.cardinalities)
        print('factors: ', self.factors)

infer = MyNewInferenceAlgo(
            student_model).print_variables()
variables: ['S', 'D', 'G', 'I', 'L']
cardianlity: {'D': 2, 'G': 3, 'I': 2,
              'S': 2, 'L': 2}
factors: defaultdict(<class 'list'>,
{'D': [<Factor representing phi(D:2)
                    at 0x7f195ed61c18>,
       <Factor representing phi(G:3, D:2, I:2)
                    at 0x7f195ed61cf8>],
 'I': [<Factor representing phi(S:2, I:2)
                    at 0x7f195ed61a58>,
       <Factor representing phi(G:3, D:2, I:2)
                    at 0x7f195ed61cf8>,
       <Factor representing phi(I:2)
                    at 0x7f195ed61e10>],
 'G': [<Factor representing phi(G:3, D:2, I:2)
                    at 0x7f195ed61cf8>,
       <Factor representing phi(L:2, G:3)
                    at 0x7f195ed61e48>],
 'S': [<Factor representing phi(S:2, I:2)
                    at 0x7f195ed61a58>],
 'L': [<Factor representing phi(L:2, G:3)
                    at 0x7f195ed61e48>]})
```

Similarly, for adding any new variable elimination order algorithm we can simply inherit from BaseEliminationOrder and define a method named cost(self, variable) which returns the cost of eliminating that variable. Inheriting this class also exposes two variables: self.bayesian_model and self.moralized_graph. We can then call the get_elimination_order method to get the elimination order. Below is an example for returning an elimination order in which the variables are sorted alphabetically.

```
from pgmpy.inference import BaseEliminationOrder
class MyEliminationAlgo(EliminationOrder):
    def cost(self, variable):
        return variable

order = MyEliminationAlgo(
            student_model).get_elimination_order()
['D', 'G', 'I', 'L', 'S']
```

Comparing pgmpy to other libraries

Starting with defining the model, pgmpy provides a very simple to use API. A model can be instantiated simply by using the __init__ method and the structure can be modified using add_node, add_edge etc methods. After the model is created, we can simply add the CPDs using the add_cpds method. In the case of eBay's bayesian belief network, we have to create a separate function for each CPD. And each of these function has a dict of CPD values and logic to return the value when the states are passed as arguments [example_bbn]. Similarly in case of libpgm we have the option to read the data from files defined in a specific format [example_libpgm] but doesn't provide any methods for making changes to the network. For changing the structure we will need to modify the internal variables storing the network information. We have tried to keep pgmpy as modular as possible. We can take the example of creating a model. We define a network structure and separately define different CPDs and then simply associate the CPDs to the structure. At any time we can modify these CPDs, unassociate or associate another CPD to the network.

Other than providing the features to easily create models, pgmpy also supports 4 standard file formats: pomdpX [pomdpX], ProbModelXML [ProbModel], XMLBeliefNetwork [XMLBelief] and XMLBIF [XMLBIF]. Using pgmpy we can read as well as write networks in these formats. Also there's an ongoing GSoC project for adding support for more file formats so hopefully we will be having support for many more formats soon.

There are many more benefits of using networkx to represent the graph structure. For example we can directly run various graph related algorihtms implemented in networkX on our networks. Also we can use networkX's plotting functionality to visualize our networks.

pgmpy also implements methods for getting independencies, D-Separation etc which would help a lot to people who are still new to Graphical Models. These features are not available in most of the other libraries.

We have tried to keep pgmpy as uniform as possible. For example we have fit and predict methods with each of the models which can automatically learn the parameters and structure and you can control the learning by simply passing arguments to these methods. Whereas in the case of libpgm, it has multiple methods for learning like lg_mle_estimateparams, lg_constraint_estimatesstruct, discrete_estimatebn etc. Similarly for each inference algorithm pgmpy prodives query and map_query methods.

Another area in which pgmpy excels is its extensibility. As we have discussed earlier, we can easily add new algorithms to pgmpy without even getting familiar with the code base. We have to tried to build pgmpy in such a way that new components can be easily added which will really help researchers working on new ideas to quickly prototype. Also, since pgmpy is documented very well it is very easy to understand the code base.

Performance wise pgmpy is a bit slower than a few libraries but we are currently actively working on improving the performance so hopefully we will be seeing a major improvement in the coming months.

Conclusion and future work

The pgmpy library provides an easy to use API for working with Graphical Models. It is also modular enough to provide separate classes for most commonly used graphical models like Naive Bayes, Hidden Markov Model etc. so that the user can directly use these special cases instead of contructing them from the base models. For machine learning problems the fit method can be used to learn parameters and predict can be used to predict values for newer data points. pgmpy's easy extensibility allows users to quickly prototype and test their ideas.

pgmpy is in a state of rapid development and some soon to come features are:

- Sampling Algorithms
- Dynamic Bayesian Networks
- Hidden Markov Models
- Support for more file formats
- Structure Learning

REFERENCES

[pgmpy] pgmpy github page https://github.com/pgmpy/pgmpy

[student] Koller, D.; Friedman, N. Probabilistic Graphical Models. Massachusetts: MIT Press, 2009, pp. 103-106.

[markov] Koller, D.; Friedman, N. Probabilistic Graphical Models. Massachusetts: MIT Press, 2009, pp. 53-54.

[bbn] bayesian-belief-networks github page https://github.com/eBay/bayesian-belief-networks

[pymc] pymc home page https://pymc-devs.github.io/pymc/

[libpgm] libpgm github page https://github.com/CyberPoint/libpgm

[pomdpX] http://bigbird.comp.nus.edu.sg/pmwiki/farm/appl/index.php?n=Main.PomdpXDocumentation

[ProbModel] http://www.probmodelxml.org/

[XMLBelief] http://xml.coverpages.org/xbn-MSdefault19990414.html

[XMLBIF] http://www.cs.cmu.edu/~fgcozman/Research/InterchangeFormat/

[example_bbn] bayesian belief network examples for creating models https://github.com/eBay/bayesian-belief-networks/tree/master/bayesian/examples/bbns

[example_libpgm] https://github.com/CyberPoint/libpgm/tree/master/examples

Python as a First Programming Language for Biomedical Scientists

Brian E. Chapman, Ph.D.[§*], Jeannie Irwin, Ph.D.[‡]

https://www.youtube.com/watch?v=kP_glnbesJ4

Abstract—We have been involved with teaching Python to biomedical scientists since 2005. In all, seven courses have been taught: 5 at the University of Pittsburgh, as a required course for biomedical informatics graduate students. Students have primarily been biomedical informatics graduate students with other students coming from human genetics, molecular biology, statistics, and similar fields. The range of prior computing experience has been wide: the majority of students had little or no prior programming experiences while a few students were experienced in other languages such as C/C++ and wanted to learn a scripting language for increased productivity. The semester-long courses have followed a procedural first approach then an introduction to object-oriented programming. By the end of the course students produce an independent programming project on a topic of their own choosing.

The course has evolved as biomedical questions have evolved, as the Python language has evolved, and as online resources have evolved. Topics of primary interest now focus on biomedical data science with analysis and visualization using tools such as Pandas, scikit-learn, and Bokeh. Class format has evolved from traditional slide-based lectures supplemented with IDLE programming demonstrations to flipped-classrooms with IPython notebooks with an interactive learning emphasis. Student evaluations indicate that students tend to find the class challenging but also empowering. The most difficult challenge for most students has been working with their computers (installing software, setting environment variables, etc.) Tools such as Canopy, Anaconda, and the IPython notebook have significantly reduced the extraneous cognitive burden on the students as they learn programming.

In addition to reviewing the nature of the course, we will review the long-term impact the course has had on the students, in terms of their retrospective evaluation of the course and the current nature of their computational toolbox. We will also discuss how our experience with these courses has been leveraged in designing a Python-centric summer school for biomedical data science.

Index Terms—education, biomedical informatics, biomedical sciences

Introduction

Python has become the most popular language for majors at the top computer science departments (Philip Guo, "Python is Now the Most Popular Introductory Teaching Language at Top U.S. Universities"). Motivations for Python as the first language are its simple semantics and syntax [Stefik2013], leading to students making fewer mistakes, feeling more confidence, and having a better grasp of programming concepts relative to peers taught with more traditional, lower-level languages such as C or Java [Koulouri2014]. Since Python is a multi-paradigm programming language, it offer great pedagogical flexibility. Python is also an active language with many open source projects and employers looking for Python programmers ("Is Python Becoming the King of the Data Science Forest?", "The RedMonk Programming Language Rankings: January 2015").

These same characteristics make Python well-suited for teaching programming to students without a computational background. The biomedical sciences are increasingly becoming computationally oriented. The installation of electronic medical records, digital public health registries, and the rise of -omics (e.g., genomics, proteomics, biomics) means biological discovery and healthcare delivery increasingly require the storage and analysis of digital data. However, students in biomedical sciences largely arrive in graduate school without computational skills. For example, biomedical informatics graduate students have diverse backgrounds including medicine, nursing, library science, psychology, and linguistics. In order to be successful in their graduate studies, the students must quickly pick up programming skills relevant to the biomedical problems they are addressing in their graduate studies.

Rather than asking students to take multiple undergraduate computer science courses, we have designed a one-semester Python programming course that allows the students to develop programming skills for their graduate studies. In this paper we first provide a brief summary of the course. Using both our personal observations and surveys of past teaching assistants and students, we then summarize our experiences with this course over the past ten years. Finally, we provide suggestions for future teaching of biomedical graduate students.

Course Objectives

The course we describe here was originally created as a required programming course for biomedical informatics students at the University of Pittsburgh. Most recently it has been offered at the University of Utah as a required course for an applied genomics certificate and as an elective for a variety of biomedical science graduate programs, including biomedical informatics, biology, human genetics, and oncological science. One of us (BEC) has seven years' experience as the course designer and instructor, the

* Corresponding author: brian.chapman@utah.edu
§ Department of Radiology, University of Utah
‡ Unaffiliated

other (JI) has one year's experience as a student (with a prior instructor) and four years' experience as a TA with BEC at the University of Pittsburgh.

As we conceive the course, it has two intended purposes. First, the course is intended to provide the students sufficient programming experience that they can use programming in their graduate studies, meaning they should be able to

a. continue to learn and improve their Python programming skills on their own,
b. successfully use Python in other programming-oriented courses during their graduate studies,
c. use Python in their thesis or dissertation work,
d. use their experience with the Python course to teach themselves another programming language, as needed.

Second, the course is intended to introduce students to the nature of biomedical data: what it looks like, what some of the standards associated with it are, and how to represent and model the data. For example, with clinical lab values, students would be asked to address whether integers, floating point numbers, or strings would be most appropriate for representing the depicted values, what type of meta-data should be associated with the value (e.g., units, method of measurements), and what sort of data structure would be most appropriate to store the data and meta-data (e.g., list, tuple, or dictionary).

Simultaneously, the course tries to illustrate biomedical problems that researchers are currently addressing so that students are not learning programming in a vacuum or purely abstractly but in the context of problems in their fields.

The course is described to students as a "boot camp" to get students with little or no programming experience up to speed for starting their graduate work. Consequently, as a "boot camp" the students should expect to spend more time than in an average three-credit course. Because this course is viewed as a foundation for subsequent graduate classes, we assume the students are self motivated and are consequently more interested in learning than in the grade received in the course.

The course is taught with a more empirical than theoretical approach, using the Python (and IPython [Perez2007]) shell to try out code snippets and see what happens. We occasionally quote Ms. Frizzle from The Magic School Bus: "Take chances, make mistakes, and get messy!" (http://magicschoolbus.wikia.com/wiki/Ms._Frizzle)

First taught in 2005, the nature of the course has transformed as the available computational and pedagogical tools have expanded. For example, learning how to read files with Pandas [McKinney2010] (http://pandas.pydata.org/) has replaced exercises in reading and parsing comma-separated files using low-level Python functionality. Similarly, static slides have been replaced by interactive IPython/Jupyter notebooks (http://ipython.org/notebook.html) and YouTube videos.

Course Structure

The course is structured around weekly homework assignments and a course project. Additional features have included quizzes (scheduled and pop), in-class and take-home exams, peer code-review, and in-class individual, pair, group, and class-wide programming assignments. Homeworks are designed to both reinforce topics that were covered in class and to require students to learn additional material on their own, primarily in the form of finding and using modules within the Python standard library. Course projects are intended to allow students to focus on an area of interest, to require them to learn additional tools, and to require them to integrate various topics covered in class. For example, they must define a base class and inherited class, interface with a database (e.g., SQLite), and have some sort of graphical user interface (e.g., IPython notebook, TKinter (https://docs.python.org/2/library/tkinter.html), Flask (http://flask.pocoo.org/), Django (https://www.djangoproject.com/)).

The semester class is roughly split in half. In the first half-semester, the course covers the fundamentals of imperative programming including numeric and string manipulation, if/else, while/for, functions, and classes. Homework assignments become progressively more demanding. In the second half-semester, topics learned in the first half are reinforced through exploration and illustration of various Python packages. Homeworks are decreased to allow the students more time to focus on their term projects. Because the illustrative applications are somewhat arbitrary, the students can request/select which topics are covered.

In-class lectures are minimized in favor of interactive programming assignments, either in the form of class-wide, small group, or individual programming projects, code reviews, or discussions about sticking points encountered during the homework. To ensure that students are motivated to be prepared for class, a "random student selector" is used to determine who will be at the podium for the next explanation or problem.

Students are encouraged to work together on homeworks and optionally can work together on term projects.

Evaluation Methods

We reviewed previous course materials and end-of-course student evaluations. Course evaluation formats varied across years and institutions making quantitative analysis difficult, but were valuable for qualitative information. In addition, we solicited input from past teaching assistants and sent a questionnaire to previous students to better assess the long-term usefulness of the course. The questionnaire was generated using SurveyMonkey and consisted of a combination of multiple-choice, Likert scale, and free-response questions. Past course lists were obtained from the University of Pittsburgh and the University of Utah. Where current e-mails were not known from the University, connections were sought through LinkedIn and other social media. Previous teaching assistants for the courses were e-mailed directly. Course materials were reviewed to observe changes in content over the years. Previous teaching assistants for the course were solicited for their analysis of the course. Twenty-seven previous students responded to the survey. However, one of the responses was blank on all questions, and so our results are based on 26 responses.

Results

Instructors' Perceived Successes and Challenges

All in all, we believe that the course has been very successful. The vast majority of students enrolling in the class achieve a functional proficiency in Python by the end of the semester. Frequently, the term project for the class has expanded into thesis or dissertation projects. At least one student with little prior programming experience started taking on "moonlighting" Python programming projects for other students and faculty. The personally communicated responses of two students remain memorable. The first student who took the course later in her graduate studies referred

to the course as "liberating." Specifically, she felt liberated from dependency on her advisor's programming staff for conducting her own graduate work. She ultimately changed course and completed a programming-centric dissertation project. The second student, a physician who attended the course as part of a short-term fellowship, referred to the class as "life changing." After completing the fellowship, he left his medical practice, received a graduate degree in biomedical informatics from Stanford University, and is currently employed by a company recently named as one of the 50 smartest companies of 2015 by MIT Technology Review (http://www.technologyreview.com/lists/companies/2015/).

The greatest challenge we have observed in teaching programming to the biomedical science graduate students is the lack of basic computer skills among students. Students have had difficulty using a shell, installing Python and an appropriate code editor and/or an integrated development environment, getting environment variables set, etc. These challenges have been substantially diminished by the use of third-party, complete Python installations, such as Anaconda or Canopy. The use of the IPython notebook has also simplified getting started for the students. However, the notebook has in some ways become a long-term detriment to some students as they are slower to adopt more powerful code editors or debugging tools.

Another challenge that we have observed repeatedly is a lack of general problem solving skills among students. This is immediately manifested in the difficulty students have in learning how to debug their programs, but lack of problem solving skills has also been manifested in tackling open-ended problems. Students have struggled with how to break a problem into small parts, and how to start with a partial solution, test it, and then move on to a more complete solution.

A final challenge with the course has been keeping the class relevant to each student. This challenge can be broken down into three parts. First, a common pedagogical problem is the breadth of prior programming experience of the students. With the limited teaching support available in most health sciences settings, it is not feasible to have multiple courses where skill levels can better match student backgrounds. Consequently, we must continually strive to not drown the weaker students while not boring the more advanced students. We believe the course evaluations indicate that we generally achieve this balance, but the balance always feels unstable. Further, we have observed that as we make the classroom more interactive, there is more opportunity for students to become frustrated with each other. Second, as the computational fields within biomedical sciences expand, it is more difficult to fashion a single course in which the instructor can meaningfully match the increasingly diverse needs of the students. Third, and perhaps most important, it has been difficult to provide relevant data sets for the students to explore. This is particularly true for students interested in clinical informatics, where privacy rules severely restrict access to data. Thankfully, federally funded efforts to increase data sharing have resulted in many relevant publicly available medical data sets. The NCI Biomedical Imaging Archive (https://imaging.nci.nih.gov/ncia/login.jsf), MT Samples (http://www.mtsamples.com/), MIMIC II [Goldberger2000]. A variety of -omic datasets (see for example http://www.ncbi.nlm.nih.gov/guide/all/ for a partial list) are now publicly available, largely due to NIH data sharing requirements connected to funding. Nonetheless, availability of large, rich data sets remains a limitation for the dual purpose of the class.

Students' Retrospective Assessment of the Course

Overall Assessment: We assessed the students' overall retrospective assessment of the course value with four Likert-scale (1: Strongly Disagree, 2: Disagree, 3: Neither Disagree or Agree, 4: Agree, 5: Strongly Agree) questions:

The responses to these questions are tabulated in Table 1.

In addition to these Likert-scale questions, we asked two open-ended questions:

- "What weaknesses and strengths do you perceive Python as having related to your work? What other programming languages (if any) do you now use? Please comment on how and why you chose them with respect to Python."
- "Please provide a short paragraph describing your retrospective analysis of the usefulness (or lack thereof) of the course. Please comment on how difficult it was for your to learn, how well you feel you still remember what you learned in the class, and whether what you learned in the class seemed relevant and up to date.

In response to our first open-ended question, reasons people listed for not using Python after the class included not programming at all, limitations of the language (memory management, speed), not considering it a statistical language (as compared to R), and collaborators using other languages (Java, Perl).

Responses to the second question were primarily positive and were similar to comments made in course evaluations. "Because I had only brief programming experience prior, the course made me much more comfortable with not only my own work and trying to incorporate automation or analysis, but also with understanding the work of others." "For me- being a novice at programming. Understanding the basics of Object Oriented Programming how to read code and think logically within a program was the best part which continues to help me today." "I thought this was a great course and perfect way to introduce OOP. I left the course feeling confident of taking on most programming challenges. Initially is was difficult to learn, but once you start thinking that way the learning accelerates."

Negative comments primarily addressed the work load of the class. "The class was too time-consuming." "I was behind on day one and was drowning in information pretty much the whole time." Similar comments can be found in course evaluation. For example, in one recent evaluation a student commented, "I felt like the class was preparing to take the mid-term on the second day of class. A fire house [hose] of information." In another evaluation a student wrote "way too much homework. I cannot stress this enough....Spending 12+hrs on homework is not conducive to a graduate student." Some negative comments indicate that we could do better in scaffolding the learning process for the students.

Prior Programming Experience of Students: We asked the students to assess their own programming experience at the time they enrolled in the class. Responses are shown in Figure 1. For students with prior programming experience, most of that prior experience was with Java (9 students) or C/C++ (9 students) with a few students reporting experience with BASIC (2), Perl (2), and JavaScript (1).

Although these responses are anonymous, and we do not know which responses correspond to which students, as an instructor BEC did not see a noticeable difference in class performance between students with no and with some prior experience. However, at least one TA felt strongly that prior experience was necessary for success in the course. Acknowledging that the course is certainly

Question	1	2	3	4	5
Learning Python was valuable for helping me subsequently learn additional programming language(s)	1	1	3	12	9
Learning Python was valuable for my career development	0	1	1	10	14
Programming is an integral part of my professional work	2	3	4	12	5
Python is my primary programming tool	3	4	5	9	5

TABLE 1: *Students' retrospective evaluation of course value*

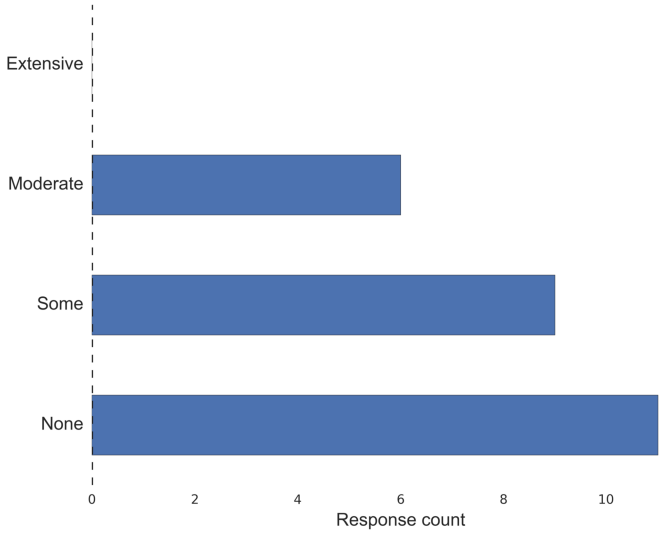


Fig. 1: *Figure 1. Prior programming experience*

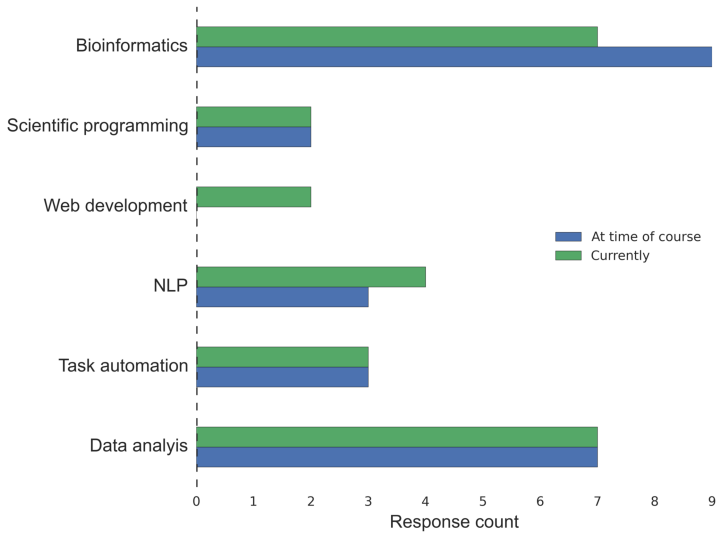


Fig. 2: *Figure 2. Student areas of focus when they enrolled in class and currently.*

easier for someone with prior programming experience, it was not uncommon for a student with no prior programming experience to be the top performing student in the course. Responses from students with some programming experience indicate that they thought the class could be difficult for a student with no prior programming experience.

Several students have suggested breaking the class into two parts: one class where the very basics of programming were covered and a second course that assumed basic knowledge of programming and covered most of the materials in the present course.

Application Areas and Valued Skill Sets: Students reported what their focus area was when they enrolled in the class and what it is currently (Figure 2). Related to this we asked them to report what topics covered in class were most valuable for them (Figure 3).

As mentioned previously, we view it as a challenge to keep the course relevant to all students. Responses indicate that we are doing reasonably well in this. Most topics covered in the class are broadly valued by the students, with web programming being less valued. However, free responses indicate that we are not covering all the topics students would have liked to learn (e.g., Biopython, scikit-learn). Some responses demonstrate a lack of understanding by students about why certain topics were covered, indicating a need for better explanation of motivation for a topic by the instructors. We concur with the following critique: "I didn't see the usefulness of some of the material while I was taking the class. Now, I wish I had continued learning some of the material after the class had ended. As a result, I am re-learning some of the

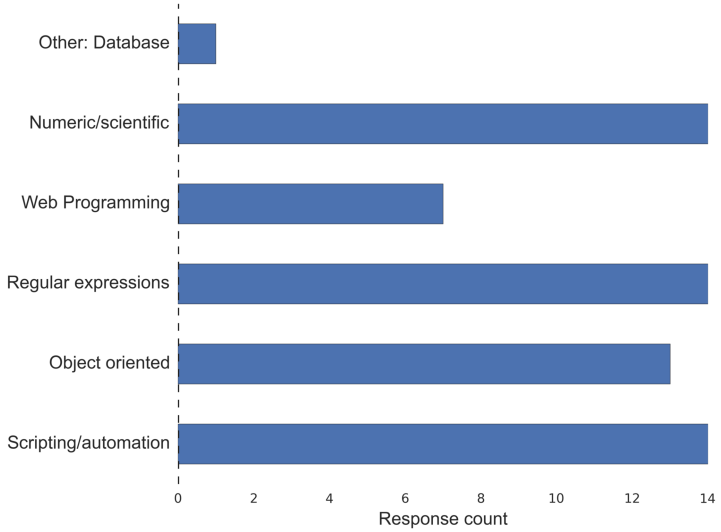


Fig. 3: *Figure 3. Topics most valuable to the students.*

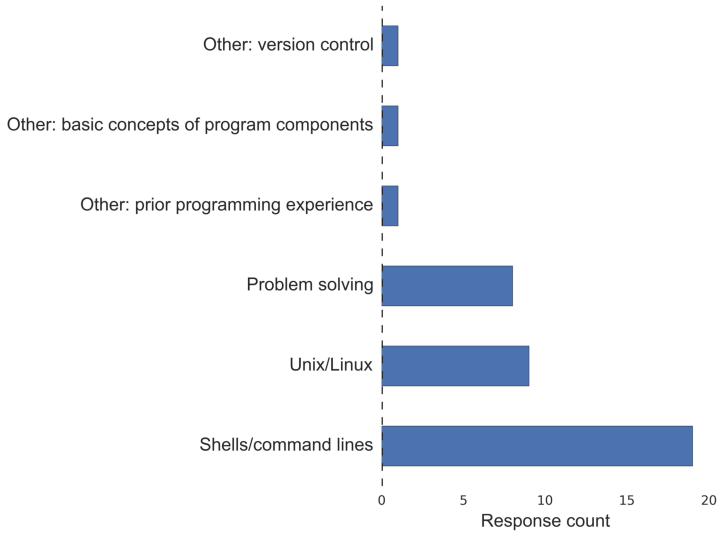


Fig. 4: *Figure 4.*

scientific tools so that I can apply them to data science concepts. Perhaps a stronger emphasis on motivating the subject would be good."

Suggested Pre-course Preparation: In the retrospective student survey, ten respondents said they would like to have been taught how to work in computer shells prior to beginning instruction in programming. In a related response, six would have liked to have been taught UNIX/Linux skills prior to beginning instruction in Python.

These responses affirm our own experience that the greatest barrier to the students' success is lack of basic computer skills. It should also be noted that the survey was only sent to students who had completed the course. Anecdotally a large number of students dropped the class before programming really began simply out of frustration with trying to install Python and text editors, set environment variables, etc. (In the most recent course, about one-third of the students dropped the course within the first month.) This was especially true of Windows users. In the most recent class, we used git for homework, and Windows users almost all adopted the git shell as their default shell rather than the Windows terminal. Anecdotally, the adoption of the git shell and the survey responses showing interest in learning UNIX/Linux occurs in the context of students (primarily bioinformatics focused) becoming familiar with a wide variety of Linux-based tools being used in their field as well as learning the power of such UNIX/Linux tools as grep, awk, and sed.

Some of our peers insist that all instruction be done in Linux and provide Linux virtual machines for their students. We concur in the value of learning the value of Linux, since it is arguably the primary scientific programming platform. However, in this class, we have opted to emphasize the platform-independent nature of Python and have let students use their own platform, particularly since clinical environments are dominated by Windows. BEC has always taught with a Mac while JI was a Windows user. Platform independence is, however, only an approximation, and there were frequent problems with the variety of platforms being used in the class. In one course evaluation a student wrote, "The instructor used a different platform (mac) but many many times there were differences between mac & windows which is what the students used. This led to annoying delays/struggles. The instructor should have done all the homework in advance on windows before assignments were given to class as well as in class examples too." In another evaluation, a student complained, "Use of Mac OS by the instructor created problems in teaching and homework, etc."

With the interest in UNIX/Linux expressed by the students, the nuisance of teaching across platforms, the acknowledged role of Linux in scientific programming, and the availability of cross-platform virtualization tools (e.g., VirtualBox, Vagrant, Docker), we believe the course would be best run using a common Linux platform.

One-third of survey respondents requested being taught general problem solving skills prior to starting programming. Two of the respondents to our survey touched upon this in their open responses. One student wrote "it did take some time to work in that problem-solving mindset," and the other wrote, "Since I came from the natural sciences it was a challenge to approach programming abstraction tasks."

Summary and Conclusion

Based on our experience over the last decade, we believe that Python is an excellent choice for teaching programming to graduate students in biomedical sciences, even when they have no prior programming experience. In the course of a semester, students were able to progress from absolute beginners to students tackling fairly complex and often useful term projects. Student responses to our survey and course evaluations support this conclusion. While including a range of responses, these survey responses and end-of-course evaluations primarily reflect the fact that our Python course is challenging but useful. We acknowledge that there might be biases in our responses in that we only e-mailed people who completed the course (not all those who enrolled in the class) and for students enrolled at the University of Pittsburgh, we were limited to contacting students for whom the Department of Biomedical Informatics had current contact information (thus excluding students from outside of the department who had enrolled) or with whom we had maintained professional contact with.

In open responses to our survey, former students expressed a variety of ways Python has helped them. The majority of students continue to use Python, and even those who do not describe Python as an important current tool, valued taking the course. In addition to expected comments about increased personal productivity and confidence, one former student who does not program as part of his professional responsibilities noted how valuable the class was for their future work supervising programmers.

The Python course has primarily been seen as a stand-alone course. However, our past experience indicates that the programming with Python course should be part of a larger series of courses. First, the students need to be introduced to working with the shell, preferably Linux. To avoid requiring students to learn another skill before class (virtualization), we are building an on-line, computational learning environment based on GitLab, Docker, and the Jupyter notebook. The Terminado emulator (https://github.com/takluyver/terminado) in the IPython notebook will be used to help students learn Linux shells. Thus the students can be exposed to the shell, Linux, and programming with no prior technical skill other than running a web browser. We believe the students would also benefit from a primer in problem solving heuristics. The classic text on this is George Pólya's *How to*

Solve It [Pólya1971]. We are interested in whether this has been generalized to problem solving outside of mathematics.

In addition to developing prelude courses, we also believe the programming instruction would be improved by breaking the course into smaller, sub-semester (quarter) pieces. In some sense, our habit of teaching 3-credit courses has shaped the course structure more than the needs of the students. By breaking the course into smaller pieces that take part of a semester (or quarter) and that the students can step into (or out of) as appropriate would better serve the students.

These ideas are being implemented for a summer biomedical data science boot camp for clinicians and others without a computational background. Python will be used as the programming language. As discussed here, the Python programming course, similar to what is described here, will be preceded by mini courses on working with Linux shells and problem solving. Following the programming course, there will be short courses on visualization, statistics, and machine learning, also using Python. The plan is for the boot camp to feed into various computationally-oriented biomedical graduate programs.

A final question related to this course might be, "Why teach a beginning course when there are many excellent on-line resources for learning Python (or other programming languages)?" We have tried to create not just another programming class, but a programming class for a specific subset of graduate students. We try to incorporate as much as possible these excellent resources into our course, but try to add to them the context of the students' academic focus. We also believe value remains for traditional face-to-face classes. Students especially valued in-class programming illustrations. And, as one student reported, "one of the not so obvious benefit of the class is the connection you made with other students who now know python. Creating a user / support group."

REFERENCES

[Koulouri2014] T. Koulouri, et al. *Teaching Introductory Programming: A Quantitative Evaluation of Different Approaches,* Trans. Comput. Educ., 14(4):1---26, December 2014.

[Stefik2013] A. Stefik and S. Siebert. *An Emperical Investigation into Programming Language Syntax,* Trans. Comput. Educ., 13(4):1---19, November 2013.

[McKinney2010] Wes McKinney. *Data Structures for Statistical Computing in Python,* Proceedings of the 9th Python in Science Conference, 51-56 (2010)

[Perez2007] Fernando Pérez and Brian E. Granger. *IPython: A System for Interactive Scientific Computing,* Computing in Science & Engineering, 9, 21-29 (2007), DOI:10.1109/MCSE.2007.53

[Pólya1971] George Pólya. *How to Solve it: A New Aspect of Mathematical Method,* Princeton University Press, 1971. publisher={Princeton University Press}

[Goldberger2000] Goldberger AL, et al. *PhysioBank, PhysioToolkit, and PhysioNet: Components of a New Research Resource for Complex Physiologic Signals.* Circulation 101(23):e215-e220

librosa: Audio and Music Signal Analysis in Python

Brian McFee[¶§*], Colin Raffel[‡], Dawen Liang[‡], Daniel P.W. Ellis[‡], Matt McVicar[**], Eric Battenberg[‖], Oriol Nieto[§]

https://www.youtube.com/watch?v=MhOdbtPhbLU

Abstract—This document describes version 0.4.0 of librosa: a Python package for audio and music signal processing. At a high level, librosa provides implementations of a variety of common functions used throughout the field of music information retrieval. In this document, a brief overview of the library's functionality is provided, along with explanations of the design goals, software development practices, and notational conventions.

Index Terms—audio, music, signal processing

Introduction

The emerging research field of music information retrieval (MIR) broadly covers topics at the intersection of musicology, digital signal processing, machine learning, information retrieval, and library science. Although the field is relatively young—the first international symposium on music information retrieval (ISMIR)[1] was held in October of 2000—it is rapidly developing, thanks in part to the proliferation and practical scientific needs of digital music services, such as iTunes, Pandora, and Spotify. While the preponderance of MIR research has been conducted with custom tools and scripts developed by researchers in a variety of languages such as MATLAB or C++, the stability, scalability, and ease of use these tools has often left much to be desired.

In recent years, interest has grown within the MIR community in using (scientific) Python as a viable alternative. This has been driven by a confluence of several factors, including the availability of high-quality machine learning libraries such as `scikit-learn` [Pedregosa11] and tools based on `Theano` [Bergstra11], as well as Python's vast catalog of packages for dealing with text data and web services. However, the adoption of Python has been slowed by the absence of a stable core library that provides the basic routines upon which many MIR applications are built. To remedy this situation, we have developed librosa:[2] a Python package for audio and music signal processing.[3] In doing so, we hope to both ease the transition of MIR researchers into Python (and modern software development practices), and also to make core MIR techniques readily available to the broader community of scientists and Python programmers.

Design principles

In designing librosa, we have prioritized a few key concepts. First, we strive for a low barrier to entry for researchers familiar with MATLAB. In particular, we opted for a relatively flat package layout, and following `scipy` [Jones01] rely upon `numpy` data types and functions [VanDerWalt11], rather than abstract class hierarchies.

Second, we expended considerable effort in standardizing interfaces, variable names, and (default) parameter settings across the various analysis functions. This task was complicated by the fact that reference implementations from which our implementations are derived come from various authors, and are often designed as one-off scripts rather than proper library functions with well-defined interfaces.

Third, wherever possible, we retain backwards compatibility against existing reference implementations. This is achieved via regression testing for numerical equivalence of outputs. All tests are implemented in the `nose` framework.[4]

Fourth, because MIR is a rapidly evolving field, we recognize that the exact implementations provided by librosa may not represent the state of the art for any particular task. Consequently, functions are designed to be *modular*, allowing practitioners to provide their own functions when appropriate, e.g., a custom onset strength estimate may be provided to the beat tracker as a function argument. This allows researchers to leverage existing library functions while experimenting with improvements to specific components. Although this seems simple and obvious, from a practical standpoint the monolithic designs and lack of interoperability between different research codebases have historically made this difficult.

Finally, we strive for readable code, thorough documentation and exhaustive testing. All development is conducted on GitHub. We apply modern software development practices, such as continuous integration testing (via Travis[5]) and coverage (via Coveralls[6]). All functions are implemented in pure Python, thoroughly documented using Sphinx, and include example code demonstrating usage. The implementation mostly complies with

* *Corresponding author: brian.mcfee@nyu.edu*
¶ *Center for Data Science, New York University*
§ *Music and Audio Research Laboratory, New York University*
‡ *LabROSA, Columbia University*
** *Department of Engineering Mathematics, University of Bristol*
‖ *Silicon Valley AI Lab, Baidu, Inc.*

1. http://ismir.net
2. https://github.com/bmcfee/librosa
3. The name *librosa* is borrowed from *LabROSA*: the LABoratory for the Recognition and Organization of Speech and Audio at Columbia University, where the initial development of librosa took place.
4. https://nose.readthedocs.org/en/latest/

PEP-8 recommendations, with a small set of exceptions for variable names that make the code more concise without sacrificing clarity: e.g., `y` and `sr` are preferred over more verbose names such as `audio_buffer` and `sampling_rate`.

Conventions

In general, librosa's functions tend to expose all relevant parameters to the caller. While this provides a great deal of flexibility to expert users, it can be overwhelming to novice users who simply need a consistent interface to process audio files. To satisfy both needs, we define a set of general conventions and standardized default parameter values shared across many functions.

An audio signal is represented as a one-dimensional `numpy` array, denoted as `y` throughout librosa. Typically the signal `y` is accompanied by the *sampling rate* (denoted `sr`) which denotes the frequency (in Hz) at which values of `y` are sampled. The duration of a signal can then be computed by dividing the number of samples by the sampling rate:

```
>>> duration_seconds = float(len(y)) / sr
```

By default, when loading stereo audio files, the `librosa.load()` function downmixes to mono by averaging left- and right-channels, and then resamples the monophonic signal to the default rate `sr=22050` Hz.

Most audio analysis methods operate not at the native sampling rate of the signal, but over small *frames* of the signal which are spaced by a *hop length* (in samples). The default frame and hop lengths are set to 2048 and 512 samples, respectively. At the default sampling rate of 22050 Hz, this corresponds to overlapping frames of approximately 93ms spaced by 23ms. Frames are centered by default, so frame index `t` corresponds to the slice:

```
y[(t * hop_length - frame_length / 2):
  (t * hop_length + frame_length / 2)],
```

where boundary conditions are handled by reflection-padding the input signal `y`. Unless otherwise specified, all sliding-window analyses use Hann windows by default. For analyses that do not use fixed-width frames (such as the constant-Q transform), the default hop length of 512 is retained to facilitate alignment of results.

The majority of feature analyses implemented by librosa produce two-dimensional outputs stored as `numpy.ndarray`, e.g., `S[f, t]` might contain the energy within a particular frequency band `f` at frame index `t`. We follow the convention that the final dimension provides the index over time, e.g., `S[:, 0]`, `S[:, 1]` access features at the first and second frames. Feature arrays are organized column-major (Fortran style) in memory, so that common access patterns benefit from cache locality.

By default, all pitch-based analyses are assumed to be relative to a 12-bin equal-tempered chromatic scale with a reference tuning of `A440 = 440.0 Hz`. Pitch and pitch-class analyses are arranged such that the 0th bin corresponds to `C` for pitch class or `C1` (32.7 Hz) for absolute pitch measurements.

Package organization

In this section, we give a brief overview of the structure of the librosa software package. This overview is intended to be superficial and cover only the most commonly used functionality. A complete API reference can be found at https://bmcfee.github.io/librosa.

5. https://travis-ci.org
6. https://coveralls.io

Core functionality

The `librosa.core` submodule includes a range of commonly used functions. Broadly, `core` functionality falls into four categories: audio and time-series operations, spectrogram calculation, time and frequency conversion, and pitch operations. For convenience, all functions within the `core` submodule are aliased at the top level of the package hierarchy, e.g., `librosa.core.load` is aliased to `librosa.load`.

Audio and time-series operations include functions such as: reading audio from disk via the `audioread` package[7] (`core.load`), resampling a signal at a desired rate (`core.resample`), stereo to mono conversion (`core.to_mono`), time-domain bounded auto-correlation (`core.autocorrelate`), and zero-crossing detection (`core.zero_crossings`).

Spectrogram operations include the short-time Fourier transform (`stft`), inverse STFT (`istft`), and instantaneous frequency spectrogram (`ifgram`) [Abe95], which provide much of the core functionality for down-stream feature analysis. Additionally, an efficient constant-Q transform (`cqt`) implementation based upon the recursive down-sampling method of Schoerkhuber and Klapuri [Schoerkhuber10] is provided, which produces logarithmically-spaced frequency representations suitable for pitch-based signal analysis. Finally, `logamplitude` provides a flexible and robust implementation of log-amplitude scaling, which can be used to avoid numerical underflow and set an adaptive noise floor when converting from linear amplitude.

Because data may be represented in a variety of time or frequency units, we provide a comprehensive set of convenience functions to map between different time representations: seconds, frames, or samples; and frequency representations: hertz, constant-Q basis index, Fourier basis index, Mel basis index, MIDI note number, or note in scientific pitch notation.

Finally, the core submodule provides functionality to estimate the dominant frequency of STFT bins via parabolic interpolation (`piptrack`) [Smith11], and estimation of tuning deviation (in cents) from the reference `A440`. These functions allow pitch-based analyses (e.g., `cqt`) to dynamically adapt filter banks to match the global tuning offset of a particular audio signal.

Spectral features

Spectral representations—the distributions of energy over a set of frequencies—form the basis of many analysis techniques in MIR and digital signal processing in general. The `librosa.feature` module implements a variety of spectral representations, most of which are based upon the short-time Fourier transform.

The Mel frequency scale is commonly used to represent audio signals, as it provides a rough model of human frequency perception [Stevens37]. Both a Mel-scale spectrogram (`librosa.feature.melspectrogram`) and the commonly used Mel-frequency Cepstral Coefficients (MFCC) (`librosa.feature.mfcc`) are provided. By default, Mel scales are defined to match the implementation provided by Slaney's auditory toolbox [Slaney98], but they can be made to match the Hidden Markov Model Toolkit (HTK) by setting the flag `htk=True` [Young97].

While Mel-scaled representations are commonly used to capture timbral aspects of music, they provide poor resolution of

7. https://github.com/sampsyo/audioread

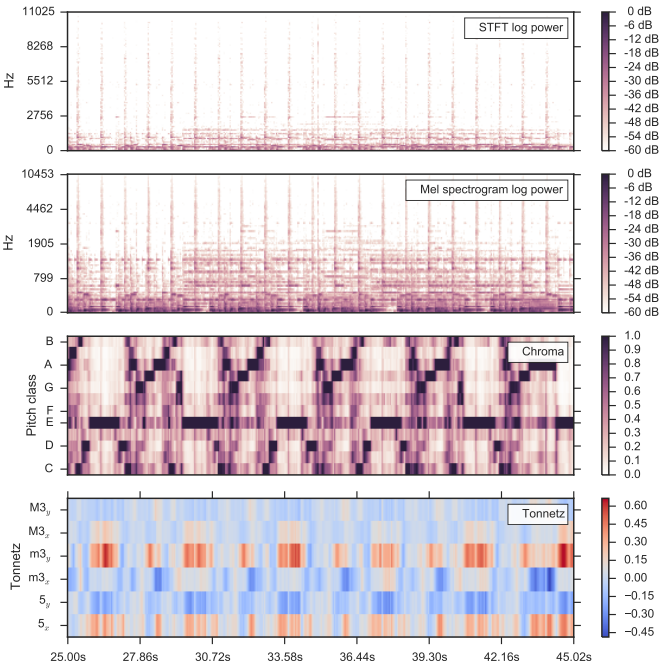


Fig. 1: *First: the short-time Fourier transform of a 20-second audio clip (`librosa.stft`). Second: the corresponding Mel spectrogram, using 128 Mel bands (`librosa.feature.melspectrogram`). Third: the corresponding chromagram (`librosa.feature.chroma_cqt`). Fourth: the Tonnetz features (`librosa.feature.tonnetz`).*

pitches and pitch classes. Pitch class (or *chroma*) representations are often used to encode harmony while suppressing variations in octave height, loudness, or timbre. Two flexible chroma implementations are provided: one uses a fixed-window STFT analysis (`chroma_stft`)[8] and the other uses variable-window constant-Q transform analysis (`chroma_cqt`). An alternative representation of pitch and harmony can be obtained by the `tonnetz` function, which estimates tonal centroids as coordinates in a six-dimensional interval space using the method of Harte et al. [Harte06]. Figure 1 illustrates the difference between STFT, Mel spectrogram, chromagram, and Tonnetz representations, as constructed by the following code fragment:[9]

```
>>> filename = librosa.util.example_audio_file()
>>> y, sr = librosa.load(filename,
...                      offset=25.0,
...                      duration=20.0)
>>> spectrogram = np.abs(librosa.stft(y))
>>> melspec = librosa.feature.melspectrogram(y=y,
...                                          sr=sr)
>>> chroma = librosa.feature.chroma_cqt(y=y,
...                                     sr=sr)
>>> tonnetz = librosa.feature.tonnetz(y=y, sr=sr)
```

In addition to Mel and chroma features, the `feature` submodule provides a number of spectral statistic representations, including `spectral_centroid`, `spectral_bandwidth`, `spectral_rolloff` [Klapuri07], and `spectral_contrast` [Jiang02].[10]

Finally, the `feature` submodule provides a few functions to implement common transformations of time-series features in MIR. This includes `delta`, which provides a smoothed estimate of the time derivative; `stack_memory`, which concatenates an input feature array with time-lagged copies of itself (effectively simulating feature *n*-grams); and `sync`, which applies a user-supplied aggregation function (e.g., `numpy.mean` or `median`) across specified column intervals.

Display

The `display` module provides simple interfaces to visually render audio data through `matplotlib` [Hunter07]. The first function, `display.waveplot` simply renders the amplitude envelope of an audio signal `y` using matplotlib's `fill_between` function. For efficiency purposes, the signal is dynamically downsampled. Mono signals are rendered symmetrically about the horizontal axis; stereo signals are rendered with the left-channel's amplitude above the axis and the right-channel's below. An example of `waveplot` is depicted in Figure 2 (top).

The second function, `display.specshow` wraps matplotlib's `imshow` function with default settings (`origin` and `aspect`) adapted to the expected defaults for visualizing spectrograms. Additionally, `specshow` dynamically selects appropriate colormaps (binary, sequential, or diverging) from the data type and range.[11] Finally, `specshow` provides a variety of acoustically relevant axis labeling and scaling parameters. Examples of `specshow` output are displayed in Figures 1 and 2 (middle).

Onsets, tempo, and beats

While the spectral feature representations described above capture frequency information, time information is equally important for many applications in MIR. For instance, it can be beneficial to analyze signals indexed by note or beat events, rather than absolute time. The `onset` and `beat` submodules implement functions to estimate various aspects of timing in music.

More specifically, the `onset` module provides two functions: `onset_strength` and `onset_detect`. The `onset_strength` function calculates a thresholded spectral flux operation over a spectrogram, and returns a one-dimensional array representing the amount of increasing spectral energy at each frame. This is illustrated as the blue curve in the bottom panel of Figure 2. The `onset_detect` function, on the other hand, selects peak positions from the onset strength curve following the heuristic described by Boeck et al. [Boeck12]. The output of `onset_detect` is depicted as red circles in the bottom panel of Figure 2.

The `beat` module provides functions to estimate the global tempo and positions of beat events from the onset strength function, using the method of Ellis [Ellis07]. More specifically, the beat tracker first estimates the tempo, which is then used to set the target spacing between peaks in an onset strength function. The output of the beat tracker is displayed as the dashed green lines in Figure 2 (bottom).

8. `chroma_stft` is based upon the reference implementation provided at http://labrosa.ee.columbia.edu/matlab/chroma-ansyn/

9. For display purposes, spectrograms are scaled by `librosa.logamplitude`. We refer readers to the accompanying IPython notebook for the full source code to recontsruct figures.

10. `spectral_*` functions are derived from MATLAB reference implementations provided by the METLab at Drexel University. http://music.ece.drexel.edu/

11. If the `seaborn` package [Waskom14] is available, its version of *cubehelix* is used for sequential data.

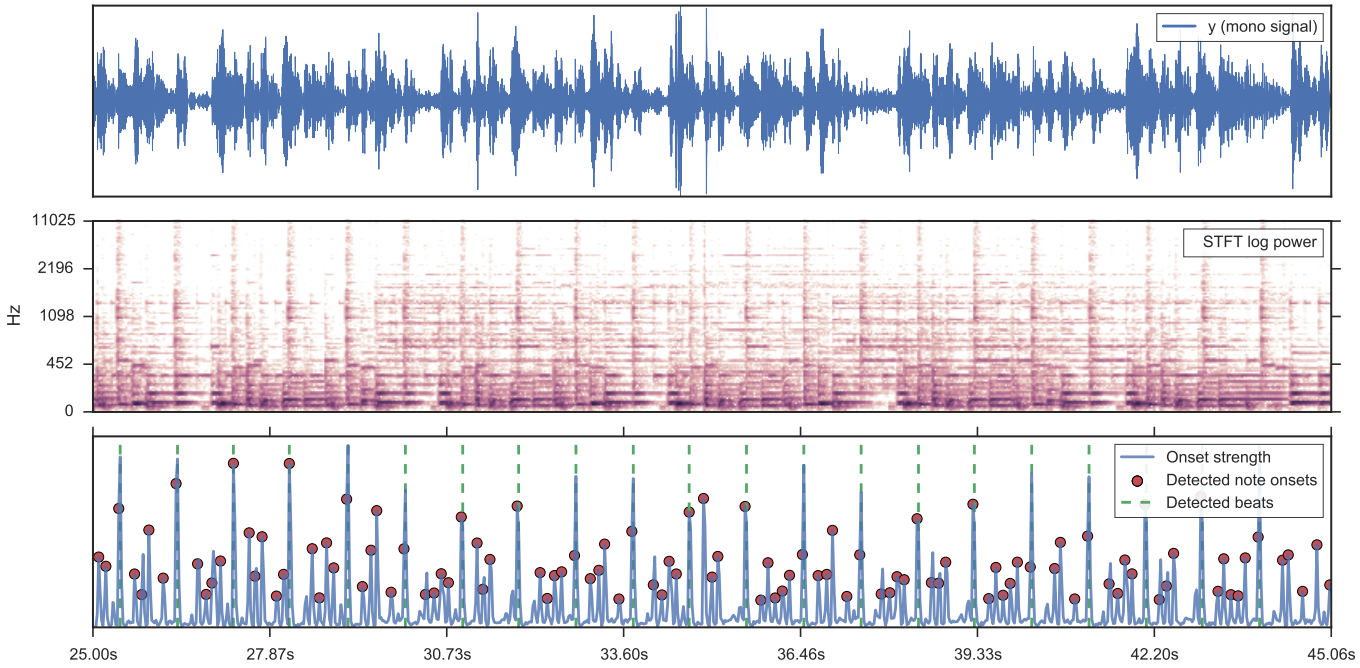


Fig. 2: *Top: a waveform plot for a 20-second audio clip* `y`*, generated by* `librosa.display.waveplot`*. Middle: the log-power short-time Fourier transform (STFT) spectrum for* `y` *plotted on a logarithmic frequency scale, generated by* `librosa.display.specshow`*. Bottom: the onset strength function (*`librosa.onset.onset_strength`*), detected onset events (*`librosa.onset.onset_detect`*), and detected beat events (*`librosa.beat.beat_track`*) for* `y`*.*

Tying this all together, the tempo and beat positions for an input signal can be easily calculated by the following code fragment:

```
>>> y, sr = librosa.load(FILENAME)
>>> tempo, frames = librosa.beat.beat_track(y=y,
...                                         sr=sr)
>>> beat_times = librosa.frames_to_time(frames,
...                                     sr=sr)
```

Any of the default parameters and analyses may be overridden. For example, if the user has calculated an onset strength envelope by some other means, it can be provided to the beat tracker as follows:

```
>>> oenv = some_other_onset_function(y, sr)
>>> librosa.beat.beat_track(onset_envelope=oenv)
```

All detection functions (beat and onset) return events as frame indices, rather than absolute timing. The downside of this is that it is left to the user to convert frame indices back to absolute time. However, in our opinion, this is outweighed by two practical benefits: it simplifies the implementations, and it makes the results directly accessible to frame-indexed functions such as `librosa.feature.sync`.

Structural analysis

Onsets and beats provide relatively low-level timing cues for music signal processing. Higher-level analyses attempt to detect larger structure in music, e.g., at the level of bars or functional components such as *verse* and *chorus*. While this is an active area of research that has seen rapid progress in recent years, there are some useful features common to many approaches. The `segment` submodule contains a few useful functions to facilitate structural analysis in music, falling broadly into two categories.

First, there are functions to calculate and manipulate *recurrence* or *self-similarity* plots. The `segment.recurrence_matrix` constructs a binary *k*-nearest-neighbor similarity matrix from a given feature array and a user-specified distance function. As displayed in Figure 3 (left), repeating sequences often appear as diagonal bands in the recurrence plot, which can be used to detect musical structure. It is sometimes more convenient to operate in *time-lag* coordinates, rather than *time-time*, which transforms diagonal structures into more easily detectable horizontal structures (Figure 3, right) [Serra12]. This is facilitated by the `recurrence_to_lag` (and `lag_to_recurrence`) functions.

Second, temporally constrained clustering can be used to detect feature change-points without relying upon repetition. This is implemented in librosa by the `segment.agglomerative` function, which uses `scikit-learn`'s implementation of Ward's agglomerative clustering method [Ward63] to partition the input into a user-defined number of contiguous components. In practice, a user can override the default clustering parameters by providing an existing `sklearn.cluster.AgglomerativeClustering` object as an argument to `segment.agglomerative()`.

Decompositions

Many applications in MIR operate upon latent factor representations, or other decompositions of spectrograms. For example, it is common to apply non-negative matrix factorization (NMF) [Lee99] to magnitude spectra, and analyze the statistics of the resulting time-varying activation functions, rather than the raw observations.

The `decompose` module provides a simple interface to factor spectrograms (or general feature arrays) into *components* and *activations*:

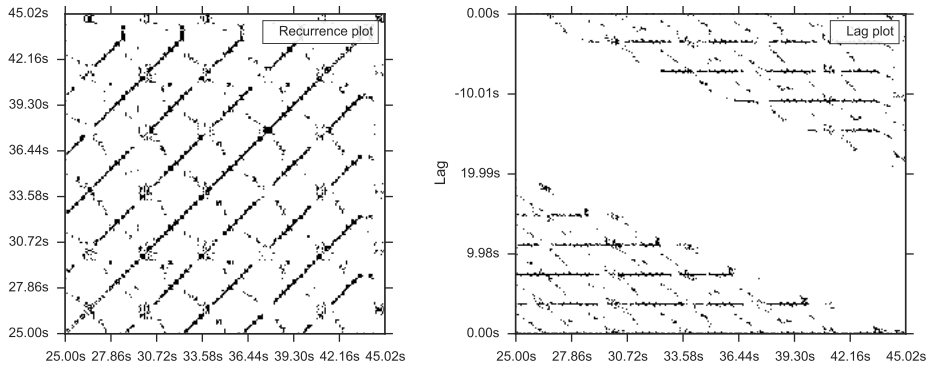


Fig. 3: *Left: the recurrence plot derived from the chroma features displayed in Figure 1. Right: the corresponding time-lag plot.*

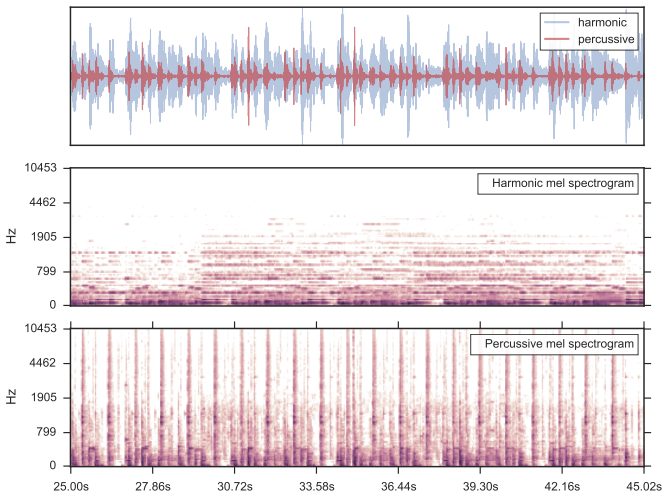


Fig. 4: *Top: the separated harmonic and percussive waveforms. Middle: the Mel spectrogram of the harmonic component. Bottom: the Mel spectrogram of the percussive component.*

```
>>> comps, acts = librosa.decompose.decompose(S)
```

By default, the `decompose()` function constructs a `scikit-learn` NMF object, and applies its `fit_transform()` method to the transpose of `S`. The resulting basis components and activations are accordingly transposed, so that `comps.dot(acts)` approximates `S`. If the user wishes to apply some other decomposition technique, any object fitting the `sklearn.decomposition` interface may be substituted:

```
>>> T = SomeDecomposer()
>>> librosa.decompose.decompose(S, transformer=T)
```

In addition to general-purpose matrix decomposition techniques, librosa also implements the harmonic-percussive source separation (HPSS) method of Fitzgerald [Fitzgerald10] as `decompose.hpss`. This technique is commonly used in MIR to suppress transients when analyzing pitch content, or suppress stationary signals when detecting onsets or other rhythmic elements. An example application of HPSS is illustrated in Figure 4.

Effects

The `effects` module provides convenience functions for applying spectrogram-based transformations to time-domain signals. For instance, rather than writing

```
>>> D = librosa.stft(y)
>>> Dh, Dp = librosa.decompose.hpss(D)
>>> y_harmonic = librosa.istft(Dh)
```

one may simply write

```
>>> y_harmonic = librosa.effects.harmonic(y)
```

Convenience functions are provided for HPSS (retaining the harmonic, percussive, or both components), time-stretching and pitch-shifting. Although these functions provide no additional functionality, their inclusion results in simpler, more readable application code.

Output

The `output` module includes utility functions to save the results of audio analysis to disk. Most often, this takes the form of annotated instantaneous event timings or time intervals, which are saved in plain text (comma- or tab-separated values) via `output.times_csv` and `output.annotation`, respectively. These functions are somewhat redundant with alternative functions for text output (e.g., `numpy.savetxt`), but provide sanity checks for length agreement and semantic validation of time intervals. The resulting outputs are designed to work with other common MIR tools, such as `mir_eval` [Raffel14] and `sonic-visualiser` [Cannam10].

The `output` module also provides the `write_wav` function for saving audio in `.wav` format. The `write_wav` simply wraps the built-in `scipy` wav-file writer (`scipy.io.wavfile.write`) with validation and optional normalization, thus ensuring that the resulting audio files are well-formed.

Caching

MIR applications typically require computing a variety of features (e.g., MFCCs, chroma, beat timings, etc) from each audio signal in a collection. Assuming the application programmer is content with default parameters, the simplest way to achieve this is to call each function using audio time-series input, e.g.:

```
>>> mfcc = librosa.feature.mfcc(y=y, sr=sr)
>>> tempo, beats = librosa.beat.beat_track(y=y,
...                                        sr=sr)
```

However, because there are shared computations between the different functions—`mfcc` and `beat_track` both compute log-scaled Mel spectrograms, for example—this results in redundant

(and inefficient) computation. A more efficient implementation of the above example would factor out the redundant features:

```
>>> lms = librosa.logamplitude(
...         librosa.feature.melspectrogram(y=y,
...                                        sr=sr))
>>> mfcc = librosa.feature.mfcc(S=lms)
>>> tempo, beats = librosa.beat.beat_track(S=lms,
...                                        sr=sr)
```

Although it is more computationally efficient, the above example is less concise, and it requires more knowledge of the implementations on behalf of the application programmer. More generally, nearly all functions in librosa eventually depend upon STFT calculation, but it is rare that the application programmer will need the STFT matrix as an end-result.

One approach to eliminate redundant computation is to decompose the various functions into blocks which can be arranged in a computation graph, as is done in Essentia [Bogdanov13]. However, this approach necessarily constrains the function interfaces, and may become unwieldy for common, simple applications.

Instead, librosa takes a lazy approach to eliminating redundancy via *output caching*. Caching is implemented through an extension of the `Memory` class from the `joblib` package[12], which provides disk-backed memoization of function outputs. The cache object (`librosa.cache`) operates as a decorator on all non-trivial computations. This way, a user can write simple application code (i.e., the first example above) while transparently eliminating redundancies and achieving speed comparable to the more advanced implementation (the second example).

The cache object is disabled by default, but can be activated by setting the environment variable `LIBROSA_CACHE_DIR` prior to importing the package. Because the `Memory` object does not implement a cache eviction policy (as of version 0.8.4), it is recommended that users purge the cache after processing each audio file to prevent the cache from filling all available disk space[13]. We note that this can potentially introduce race conditions in multi-processing environments (i.e., parallel batch processing of a corpus), so care must be taken when scheduling cache purges.

Parameter tuning

Some of librosa's functions have parameters that require some degree of tuning to optimize performance. In particular, the performance of the beat tracker and onset detection functions can vary substantially with small changes in certain key parameters.

After standardizing certain default parameters—sampling rate, frame length, and hop length—across all functions, we optimized the beat tracker settings using the parameter grid given in Table 1. To select the best-performing configuration, we evaluated the performance on a data set comprised of the Isophonics Beatles corpus[14] and the SMC Dataset2 [Holzapfel12] beat annotations. Each configuration was evaluated using `mir_eval` [Raffel14], and the configuration was chosen to maximize the Correct Metric Level (Total) metric [Davies14].

Similarly, the onset detection parameters (listed in Table 2) were selected to optimize the F1-score on the Johannes Kepler University onset database.[15]

12. https://github.com/joblib/joblib
13. The cache can be purged by calling `librosa.cache.clear()`.
14. http://isophonics.net/content/reference-annotations
15. https://github.com/CPJKU/onset_db

Parameter	Description	Values
`fmax`	Maximum frequency value (Hz)	8000, **11025**
`n_mels`	Number of Mel bands	32, 64, **128**
`aggregate`	Spectral flux aggregation function	`np.mean`, **np.median**
`ac_size`	Maximum lag for onset autocorrelation (s)	2, **4**, 8
`std_bpm`	Deviation of tempo estimates from 120.0 BPM	0.5, **1.0**, 2.0
`tightness`	Penalty for deviation from estimated tempo	50, **100**, 400

TABLE 1: *The parameter grid for beat tracking optimization. The best configuration is indicated in bold.*

Parameter	Description	Values
`fmax`	Maximum frequency value (Hz)	8000, **11025**
`n_mels`	Number of Mel bands	32, 64, **128**
`aggregate`	Spectral flux aggregation function	**np.mean**, `np.median`
`delta`	Peak picking threshold	0.0--0.10 **(0.07)**

TABLE 2: *The parameter grid for onest detection optimization. The best configuration is indicated in bold.*

We note that the "optimal" default parameter settings are merely estimates, and depend upon the datasets over which they are selected. The parameter settings are therefore subject to change in the future as larger reference collections become available. The optimization framework has been factored out into a separate repository, which may in subsequent versions grow to include additional parameters.[16]

Conclusion

This document provides a brief summary of the design considerations and functionality of librosa. More detailed examples, notebooks, and documentation can be found in our development repository and project website. The project is under active development, and our roadmap for future work includes efficiency improvements and enhanced functionality of audio coding and file system interactions.

Citing librosa

We request that when using librosa in academic work, authors cite the Zenodo reference [McFee15]. For references to the *design* of the library, citation of the present document is appropriate.

Acknowledgements

BM acknowledges support from the Moore-Sloan Data Science Environment at NYU. Additional support was provided by NSF grant IIS-1117015.

References

[Pedregosa11] Pedregosa, Fabian, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel et al. *Scikit-learn: Machine learning in Python.* The Journal of Machine Learning Research 12 (2011): 2825-2830.

16. https://github.com/bmcfee/librosa_parameters

[Bergstra11] Bergstra, James, Frédéric Bastien, Olivier Breuleux, Pascal Lamblin, Razvan Pascanu, Olivier Delalleau, Guillaume Desjardins et al. *Theano: Deep learning on gpus with python.* In NIPS 2011, BigLearning Workshop, Granada, Spain. 2011.

[Jones01] Jones, Eric, Travis Oliphant, and Pearu Peterson. *SciPy: Open source scientific tools for Python.* http://www.scipy.org/ (2001).

[VanDerWalt11] Van Der Walt, Stefan, S. Chris Colbert, and Gael Varoquaux. *The NumPy array: a structure for efficient numerical computation.* Computing in Science & Engineering 13, no. 2 (2011): 22-30.

[Abe95] Abe, Toshihiko, Takao Kobayashi, and Satoshi Imai. *Harmonics tracking and pitch extraction based on instantaneous frequency.* International Conference on Acoustics, Speech, and Signal Processing, ICASSP-95., Vol. 1. IEEE, 1995.

[Schoerkhuber10] Schoerkhuber, Christian, and Anssi Klapuri. *Constant-Q transform toolbox for music processing.* 7th Sound and Music Computing Conference, Barcelona, Spain. 2010.

[Smith11] Smith, J.O. "Sinusoidal Peak Interpolation", in Spectral Audio Signal Processing, https://ccrma.stanford.edu/~jos/sasp/Sinusoidal_Peak_Interpolation.html , online book, 2011 edition, accessed 2015-06-15.

[Stevens37] Stevens, Stanley Smith, John Volkmann, and Edwin B. Newman. *A scale for the measurement of the psychological magnitude pitch.* The Journal of the Acoustical Society of America 8, no. 3 (1937): 185-190.

[Slaney98] Slaney, Malcolm. *Auditory toolbox.* Interval Research Corporation, Tech. Rep 10 (1998): 1998.

[Young97] Young, Steve, Evermann, Gunnar, Gales, Mark, Hain, Thomas, Kershaw, Dan, Liu, Xunying (Andrew), Moore, Gareth, Odell, Julian, Ollason, Dave, Povey, Dan, Valtchev, Valtcho, and Woodland, Phil. *The HTK book.* Vol. 2. Cambridge: Entropic Cambridge Research Laboratory, 1997.

[Harte06] Harte, C., Sandler, M., & Gasser, M. (2006). *Detecting Harmonic Change in Musical Audio.* In Proceedings of the 1st ACM Workshop on Audio and Music Computing Multimedia (pp. 21-26). Santa Barbara, CA, USA: ACM Press. doi:10.1145/1178723.1178727.

[Jiang02] Jiang, Dan-Ning, Lie Lu, Hong-Jiang Zhang, Jian-Hua Tao, and Lian-Hong Cai. *Music type classification by spectral contrast feature.* In ICME'02. vol. 1, pp. 113-116. IEEE, 2002.

[Klapuri07] Klapuri, Anssi, and Manuel Davy, eds. *Signal processing methods for music transcription.* Springer Science & Business Media, 2007.

[Hunter07] Hunter, John D. *Matplotlib: A 2D graphics environment.* Computing in science and engineering 9, no. 3 (2007): 90-95.

[Waskom14] Michael Waskom, Olga Botvinnik, Paul Hobson, John B. Cole, Yaroslav Halchenko, Stephan Hoyer, Alistair Miles, et al. *Seaborn: v0.5.0 (November 2014).* ZENODO, 2014. doi:10.5281/zenodo.12710.

[Boeck12] Böck, Sebastian, Florian Krebs, and Markus Schedl. *Evaluating the Online Capabilities of Onset Detection Methods.* In 11th International Society for Music Information Retrieval Conference (ISMIR 2012), pp. 49-54. 2012.

[Ellis07] Ellis, Daniel P.W. *Beat tracking by dynamic programming.* Journal of New Music Research 36, no. 1 (2007): 51-60.

[Serra12] Serra, Joan, Meinard Müller, Peter Grosche, and Josep Lluis Arcos. *Unsupervised detection of music boundaries by time series structure features.* In Twenty-Sixth AAAI Conference on Artificial Intelligence. 2012.

[Ward63] Ward Jr, Joe H. *Hierarchical grouping to optimize an objective function.* Journal of the American statistical association 58, no. 301 (1963): 236-244.

[Lee99] Lee, Daniel D., and H. Sebastian Seung. *Learning the parts of objects by non-negative matrix factorization.* Nature 401, no. 6755 (1999): 788-791.

[Fitzgerald10] Fitzgerald, Derry. *Harmonic/percussive separation using median filtering.* 13th International Conference on Digital Audio Effects (DAFX10), Graz, Austria, 2010.

[Cannam10] Cannam, Chris, Christian Landone, and Mark Sandler. *Sonic visualiser: An open source application for viewing, analysing, and annotating music audio files.* In Proceedings of the international conference on Multimedia, pp. 1467-1468. ACM, 2010.

[Holzapfel12] Holzapfel, Andre, Matthew E.P. Davies, José R. Zapata, João Lobato Oliveira, and Fabien Gouyon. *Selective sampling for beat tracking evaluation.* Audio, Speech, and Language Processing, IEEE Transactions on 20, no. 9 (2012): 2539-2548.

[Davies14] Davies, Matthew E.P., and Boeck, Sebastian. *Evaluating the evaluation measures for beat tracking.* In 15th International Society for Music Information Retrieval Conference (ISMIR 2014), 2014.

[Raffel14] Raffel, Colin, Brian McFee, Eric J. Humphrey, Justin Salamon, Oriol Nieto, Dawen Liang, and Daniel PW Ellis. *mir eval: A transparent implementation of common MIR metrics.* In 15th International Society for Music Information Retrieval Conference (ISMIR 2014), pp. 367-372. 2014.

[Bogdanov13] Bogdanov, Dmitry, Nicolas Wack, Emilia Gómez, Sankalp Gulati, Perfecto Herrera, Oscar Mayor, Gerard Roma, Justin Salamon, José R. Zapata, and Xavier Serra. *Essentia: An Audio Analysis Library for Music Information Retrieval.* In 12th International Society for Music Information Retrieval Conference (ISMIR 2013), pp. 493-498. 2013.

[McFee15] Brian McFee, Matt McVicar, Colin Raffel, Dawen Liang, Oriol Nieto, Josh Moore, Dan Ellis, et al. *Librosa: v0.4.0.* Zenodo, 2015. doi:10.5281/zenodo.18369.

PyEDA: Data Structures and Algorithms for Electronic Design Automation

Chris Drake[‡*]

https://www.youtube.com/watch?v=cljDuK0ouRs

Abstract—This paper introduces PyEDA, a Python library for electronic design automation (EDA). PyEDA provides both a high level interface to the representation of Boolean functions, and blazingly-fast C extensions for fundamental algorithms where performance is essential. PyEDA is a hobby project which has the simple but audacious goal of improving the state of digital design by using Python.

Introduction

Chip design and verification is a complicated undertaking. You must assemble a large team of engineers with many different specialties: front-end design entry, logic verification, power optimization, synthesis, place and route, physical verification, and so on. Unfortunately, the tools, languages, and work flows offered by the electronic design automation (EDA) industry are, in this author's opinion, largely a pit of despair. The languages most familiar to chip design and verification engineers are Verilog (now SystemVerilog), C/C++, TCL, and Perl. Flows are patched together from several proprietary tools with incompatible data representations. Even with Python's strength in scientific computing, it has largely failed to penetrate this space. In short, EDA needs more Python!

This paper surveys some of the features and applications of PyEDA, a Python library for electronic design automation. PyEDA provides both a high level interface to the representation of Boolean functions, and blazingly-fast C extensions for fundamental algorithms where performance is essential.

PyEDA is a hobby project, but in the past year it has seen some interesting adoption from University students. For example, students at Vanderbilt University used it to model system reliability [Nan14], and students at Saarland University used as part of a fast DQBF Refutation tool [Fin14].

Even though the name "PyEDA" implies that the library is specific to EDA, it is actually general in nature. Some of the techniques used for designing and verifying digital logic are fundamental to computer science. For example, we will discuss applications of Boolean satisfiability (SAT), the definitive NP-complete problem.

PyEDA's repository is hosted at https://github.com/cjdrake/pyeda.git, and its documentation is hosted at http://pyeda.rtfd.org.

* *Corresponding author: cjdrake@gmail.com*
‡ *Drake Enterprises*

Note About Code Blocks

This document contains several Python code blocks. For the sake of simplicity, we assume you have PyEDA installed, and have prepared an interactive terminal by executing:

```
>>> from pyeda.inter import *
```

Boolean Variables and Functions

At its core, PyEDA provides a powerful API for creating and manipulating Boolean functions.

First, let us provide the standard definitions.

A Boolean *variable* is an abstract numerical quantity that can take any value in the set $\{0,1\}$. A Boolean function is a rule that maps points in an N-dimensional Boolean space to an element in $\{0,1\}$. Formally, $f : B^N \Rightarrow B$, where B^N means the Cartesian product of N sets of type $\{0,1\}$. For example, if you have three input variables, a,b,c, each defined on $\{0,1\}$, then $B^3 = \{0,1\}^3 = \{(0,0,0),(0,0,1),...,(1,1,1)\}$. B^3 is the **domain** of the function (the input part), and $B = \{0,1\}$ is the **range** of the function (the output part). The set of all input variables a function depends on is called its *support*.

There are several ways to represent a Boolean function, and different data structures have different tradeoffs. In the following sections, we will give a brief overview of PyEDA's API for logic expressions, truth tables, and binary decision diagrams. In addition, we will provide implementation notes for several useful applications.

Logic Expressions

Logic expressions are a powerful and flexible way to represent Boolean functions. They are implemented as a graph, with *atoms* at the branches, and *operators* at the leaves. Atomic elements are *literals* (variables and complemented variables), and *constants* (zero and one). The supported algebraic operators are `Not`, `Or`, `And`, `Xor`, `Equal`, `Implies`, and `ITE` (if-then-else).

For general purpose use, symbolic logic expressions are PyEDA's central data type. Since release `0.27`, they have been implemented using a high performance C library.

Expressions are fast, and reasonably compact. On the other hand, they are generally not canonical, and determining expression equivalence is NP-complete. Conversion to a canonical expression form can result in exponential size.

Name	OR	AND
Commutativity	$x+y=y+x$	$x\cdot y=y\cdot x$
Associativity	$x+(y+z)=(x+y)+z$	$x\cdot(y\cdot z)=(x\cdot y)\cdot z$
Identity	$x+0=x$	$x\cdot 1=x$
Domination	$x+1=1$	$x\cdot 0=0$
Idempotence	$x+x=x$	$x\cdot x=x$
Inverse	$x+x'=1$	$x\cdot x'=0$

TABLE 1: *Boolean OR/AND Identities*

Construction

To construct a logic expression, first start by defining some symbolic *variables* of type `Expression`:

```
>>> a, b, c, d = map(exprvar, 'abcd')
```

By overloading Python's logical operators, you can build expression algebraically:

```
>>> F = a | ~b & c ^ ~d
```

Use methods from the `Function` base class to explore the function's basic properties:

```
>>> F.support
frozenset({a, b, c, d})
>>> list (F.iter_relation())
[({a: 0, b: 0, c: 0, d: 0}, 0),
 ({a: 1, b: 0, c: 0, d: 0}, 1),
 ({a: 0, b: 1, c: 0, d: 0}, 0),
 ...
 ({a: 0, b: 1, c: 1, d: 1}, 0),
 ({a: 1, b: 1, c: 1, d: 1}, 1)]
```

There are also several factory functions that offer more power than Python's built-in binary operators. For example, operators such as `Or`, `And`, and `Xor` allow you to construct N-ary expressions:

```
>>> a ^ b ^ c
Xor(Xor(a, b), c)
>>> Xor(a, b, c)
Xor(a, b, c)
```

Also, functions such as `OneHot`, and `Majority` implement powerful, higher order functions:

```
>>> OneHot(a, b, c)
And(Or(~a, ~b), Or(~a, ~c), Or(~b, ~c), Or(a, b, c))
>>> Majority(a, b, c)
Or(And(a, b), And(a, c), And(b, c))
```

Simplification

The laws of Boolean Algebra can be used to simplify expressions. For example, Table 1 enumerates a partial list of Boolean identities for the `Or` and `And` operators.

Most laws are computationally easy to apply. PyEDA allows you to construct unsimplified Boolean expressions, and provides the `simplify` method to perform such inexpensive transformations.

For example:

```
>>> F = ~a | a
>>> F
Or(~a, a)
>>> F.simplify()
1
>>> Xor(a, ~b, Xnor(~a, b), c)
~c
```

Performing simplification can dramatically reduce the size and depth of your logic expressions.

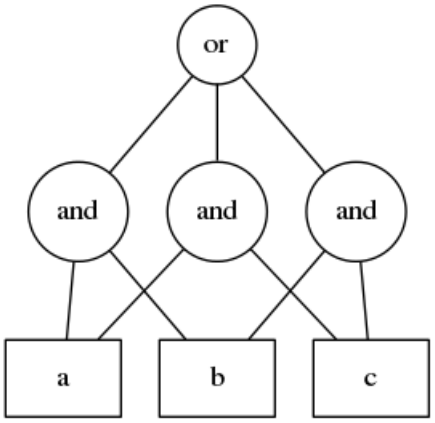


Fig. 1: *Majority expression rendered by Graphviz*

Transformation

PyEDA also supports a growing list of expression transformations. Since expressions are not a canonical form, transformations can help explore tradeoffs in time and space, as well as convert an expression to a form suitable for a particular algorithm.

For example, in addition to the primary operators `Not`, `Or`, and `And`, expressions also natively support the secondary `Xor`, `Equal`, `Implies`, and `ITE` (if-then-else) operators. By transforming all secondary operators into primary operators, and pushing all `Not` operators down towards the leaf nodes, you arrive at what is known as "negation normal form".

```
>>> F = Xor(a >> b, c.eq(d))
>>> F.to_nnf()
And(Or(And(Or(c, d), Or(~c, ~d)), And(a, ~b)),
       Or(~a, b, And(~c, ~d), And(c, d)))
```

Currently, expressions also support conversion to the following forms:

- Binary operator (only two args per `Or`, `And`, etc)
- Disjunctive Normal Form (DNF)
- Conjunctive Normal Form (CNF)

DNF and CNF expressions are "two-level" forms. That is, the entire expression is either an `Or` of `And` clauses (DNF), or an `And` of `Or` clauses (CNF). DNF expressions are also called "covers", and are important in both two-level and multi-level logic minimization. CNF expressions play an important role in satisfiability. We will briefly cover both of these topics in subsequent sections.

Visualization

Boolean expressions support a `to_dot()` method, which can be used to convert the graph structure to DOT format for consumption by Graphviz. For example, Figure 1 shows the Graphviz output on the majority function in three variables:

```
>>> F = Majority(a, b, c)
>>> F.to_dot()
```

The `expr` Function

The `expr` function is a factory function that attempts to transform any input into a logic expression. It does the obvious thing when converting inputs that look like Boolean values:

```
>>> expr(False)
0
>>> expr(1)
```

```
1
>>> expr("0")
0
```

But it also implements a full top-down parser of expressions. For example:

```
>>> expr("a | b ^ c & d")
Or(a, Xor(b, And(c, d)))
```

See the documentation for a complete list of supported operators accepted by the `expr` function.

Boolean Satisfiability

One of the most interesting questions in computer science is whether a given Boolean function is satisfiable, or SAT. That is, for a given function F, is there a set of input assignments that will produce an output of 1?

PyEDA Boolean functions implement two functions for this purpose, `satisfy_one`, and `satisfy_all`. The former answers the question in a yes/no fashion, returning a satisfying input point if the function is satisfiable, and `None` otherwise. The latter returns a generator that will iterate through all satisfying input points.

SAT has all kinds of applications in both digital design and verification. In digital design, it can be used in equivalence checking, test pattern generation, model checking, formal verification, and constrained-random verification, among others. SAT finds its way into other areas as well. For example, modern package management systems such as apt and yum might use SAT to guarantee that certain dependencies are satisfied for a given configuration.

The `pyeda.boolalg.picosat` module provides an interface to the modern SAT solver PicoSAT [Bie08]. When a logic expression is in conjunctive normal form (CNF), calling the `satisfy_*` methods will invoke PicoSAT transparently.

For example:

```
>>> F = OneHot(a, b, c)
>>> F.is_cnf()
True
>>> F.satisfy_one()
{a: 0, b: 0, c: 1}
>>> list(F.satisfy_all())
[{a: 0, b: 0, c: 1},
 {a: 0, b: 1, c: 0},
 {a: 1, b: 0, c: 0}]
```

When an expression is not a CNF, PyEDA will resort to a standard, backtracking algorithm. The worst-case performance of this implementation is exponential, but is acceptable for many real-world scenarios.

Tseitin Transformation

The worst case memory consumption when converting to CNF is exponential. This is due to the fact that distribution of M `Or` clauses over N `And` clauses (or vice-versa) requires $M \times N$ clauses.

```
>>> Or(And(a, b), And(c, d)).to_cnf()
And(Or(a, c), Or(b, c), Or(a, d), Or(b, d))
```

Logic expressions support the `tseitin` method, which perform's Tseitin's transformation on the input expression. For more information about this transformation, see [Tse68].

The Tseitin transformation does not produce an equivalent expression, but rather an *equisatisfiable* CNF, with the addition of auxiliary variables. The important feature is that it can convert any expression into a CNF, which can be solved using PicoSAT.

```
>>> F = Xor(a, b, c, d)
>>> soln = F.tseitin().satisfy_one()
>>> soln
{a: 0,
 aux[0]: 1,
 aux[1]: 1,
 ...
 b: 0,
 c: 0,
 d: 1}
```

You can safely discard the `aux` variables to get the solution:

```
>>> {k: v for k, v in soln.items() if k.name != 'aux'}
{a: 0, b: 0, c: 0, d: 1}
```

Truth Tables

The most straightforward way to represent a Boolean function is to simply enumerate all possible mappings from input assignment to output values. This is known as a truth table, It is implemented as a packed list, where the index of the output value corresponds to the assignment of the input variables. The nature of this data structure implies an exponential size. For N input variables, the table will be size 2^N. It is therefore mostly useful for manual definition and inspection of functions of reasonable size.

To construct a truth table from scratch, use the `truthtable` factory function. For example, to represent the `And` function:

```
>>> truthtable([a, b], [False, False, False, True])
# This also works
>>> truthtable([a, b], "0001")
```

You can also convert expressions to truth tables using the `expr2truthtable` function:

```
>>> expr2truthtable(OneHot0(a, b, c))
c b a
0 0 0 : 1
0 0 1 : 1
0 1 0 : 1
0 1 1 : 0
1 0 0 : 1
1 0 1 : 0
1 1 0 : 0
1 1 1 : 0
```

Partial Definitions

Another use for truth tables is the representation of *partially defined* functions. Logic expressions and binary decision diagrams are *completely defined*, meaning that their implementation imposes a complete mapping from all points in the domain to $\{0, 1\}$. Truth tables allow you to specify some function outputs as "don't care". You can accomplish this by using either `"-"` or `"X"` with the `truthtable` function.

For example, a seven segment display is used to display decimal numbers. The codes "0000" through "1001" are used for 0-9, but codes "1010" through "1111" are not important, and therefore can be labeled as "don't care".

```
>>> X = ttvars('x', 4)
>>> F1 = truthtable(X, "0000011111------")
>>> F2 = truthtable(X, "0001111100------")
```

To convert a table to a two-level, disjunctive normal form (DNF) expression, use the `truthtable2expr` function:

```
>>> truthtable2expr(F1)
Or(And(x[0], ~x[1], x[2], ~x[3]),
   And(~x[0], x[1], x[2], ~x[3]),
   And(x[0], x[1], x[2], ~x[3]),
   And(~x[0], ~x[1], ~x[2], x[3]),
   And(x[0], ~x[1], ~x[2], x[3]))
```

Two-Level Logic Minimization

When choosing a physical implementation for a Boolean function, the size of the logic network is proportional to its cost, in terms of area and power. Therefore it is desirable to reduce the size of that network.

Logic minimization of two-level forms is an NP-complete problem. It is equivalent to finding a minimal-cost set of subsets of a set S that covers S. This is sometimes called the "paving problem", because it is conceptually similar to finding the cheapest configuration of tiles that cover a floor. Due to the complexity of this operation, PyEDA uses a C extension to the Berkeley Espresso library [Bra84].

After calling the `espresso_tts` function on the `F1` and `F2` truth tables from above, observe how much smaller (and therefore cheaper) the resulting DNF expression is:

```
>>> F1M, F2M = espresso_tts(F1, F2)
>>> F1M
Or(x[3], And(x[0], x[2]), And(x[1], x[2]))
```

Binary Decision Diagrams

A binary decision diagram is a directed acyclic graph used to represent a Boolean function. They were originally introduced by Lee, and later by Akers. In 1986, Randal Bryant introduced the reduced, ordered BDD (ROBDD).

The ROBDD is a canonical form, which means that given an identical ordering of input variables, equivalent Boolean functions will always reduce to the same ROBDD. This is a desirable property for determining formal equivalence. Also, it means that unsatisfiable functions will be reduced to zero, making SAT/UNSAT calculations trivial. Due to these auspicious properties, the term BDD almost always refers to some minor variation of the ROBDD devised by Bryant.

The downside of BDDs is that certain functions, no matter how cleverly you order their input variables, will result in an exponentially-sized graph data structure.

Construction

Like logic expressions, you can construct a BDD by starting with symbolic variables and combining them with operators.

For example:

```
>>> a, b, c = map(bddvar, 'abc')
>>> F = a & b & c
>>> F.support
frozenset({a, b, c})
>>> F.restrict({a: 1, b: 1})
c
>>> F & 0
0
```

The `expr2bdd` function can also be used to convert any expression into an equivalent BDD:

```
>>> expr2bdd(expr("(s ? d1 : d0) <=> (s & d1 | ~s & d0)"))
1
```

Equivalence

As we mentioned before, BDDs are a canonical form. This makes checking for SAT, UNSAT, and formal equivalence trivial.

```
>>> ~a & a
0
>>> ~a & ~b | ~a & b | a & ~b | a & b
1
>>> F = a ^ b
```

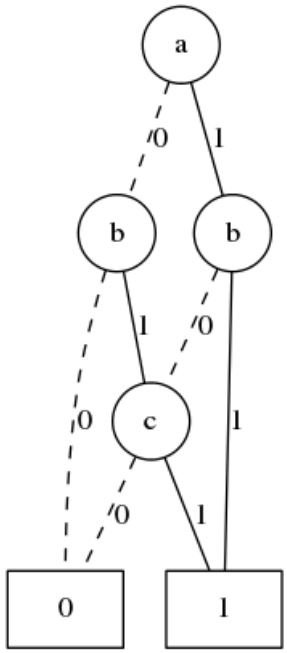


Fig. 2: *Majority BDD rendered by Graphviz*

```
>>> G = ~a & b | a & ~b
>>> F.equivalent(G)
True
>>> F is G
True
```

PyEDA's BDD implementation uses a unique table, so `F` and `G` from the previous example are actually just two different names for the same object.

Visualization

Like expressions, binary decision diagrams also support a `to_dot()` method, which can be used to convert the graph structure to DOT format for consumption by Graphviz. For example, Figure 2 shows the Graphviz output on the majority function in three variables:

```
>>> expr2bdd(expr("Majority(a, b, c)")).to_dot()
```

Future Directions for Function Data Structures

The implementation of Boolean functions is a vast field, and PyEDA is really only scratching the surface. In this section we will describe several directions for improvement.

Due to their fundamentally exponential size, truth tables have limited application. It is more common for tabular function representations to use an implicant table, sometimes referred to as a "cover". PyEDA has some support for implicant tables in the Espresso C extension, but this functionality is not exposed to the user interface.

PyEDA's current implementation of BDDs is written in pure Python. Given that BDDs are memory limited, the `PyObject` data type imposes a hefty overhead on the size of the DAG. Also, there are currently no complemented edges or automatic variable reordering, features that more complete decision diagram libraries implement. One solution is to implement a Python C extension to a more complete and high performance library such as [CUDD].

There are several function representations left for consideration. Within the realm of decision diagrams, we have not considered algebraic decision diagrams (ADDs), or zero-suppressed decision diagrams (ZDDs). Within the realm of graph-based structures primarily for logic synthesis, we have not considered the and-inverter-graph (AIG), or the majority-inverter-graph (MIG).

Function Arrays

When dealing with several related Boolean functions, it is usually convenient to index the inputs and outputs. For this purpose, PyEDA includes a multi-dimensional array (MDA) data type, called an `farray` (function array).

The most pervasive example is computation involving any numeric data type. For example, let's say you want to add two numbers `A`, and `B`. If these numbers are 32-bit integers, there are 64 total inputs, not including a carry-in. The conventional way of labeling the input variables is $a_0, a_1, \ldots, a_{31}$, and $b_0, b_1, \ldots, b_{31}$.

Furthermore, you can extend the symbolic algebra of Boolean functions to arrays. For example, the element-wise XOR of `A` and `B` is also an array.

In this section, we will briefly discuss `farray` construction, slicing operations, and algebraic operators. Function arrays can be constructed using any `Function` implementation, but for simplicity we will restrict the discussion to logic expressions.

Construction

The `farray` constructor can be used to create an array of arbitrary expressions.

```
>>> a, b, c, d = map(exprvar, 'abcd')
>>> F = farray([a, b, And(a, c), Or(b, d)])
>>> F.ndim
1
>>> F.size
4
>>> F.shape
((0, 4), )
```

As you can see, this produces a one-dimensional array of size 4.

The shape of the previous array uses Python's conventional, exclusive indexing scheme in one dimension. The `farray` constructor also supports multi-dimensional arrays:

```
>>> G = farray([ [a, b],
                 [And(a, c), Or(b, d)],
                 [Xor(b, c), Equal(c, d)] ])
>>> G.ndim
2
>>> G.size
6
>>> G.shape
((0, 3), (0, 2))
```

Though arrays can be constructed from arbitrary functions in arbitrary shapes, it is far more useful to start with arrays of variables and constants, and build more complex arrays from them using operators.

To construct arrays of expression variables, use the `exprvars` factory function:

```
>>> xs = exprvars('x', 8)
>>> xs
farray([x[0], x[1], x[2], x[3], x[4], x[5], x[6], x[7]])
>>> ys = exprvars('y', 4, 4)
farray([[y[0,0], y[0,1], y[0,2], y[0,3]],
        [y[1,0], y[1,1], y[1,2], y[1,3]],
        [y[2,0], y[2,1], y[2,2], y[2,3]],
        [y[3,0], y[3,1], y[3,2], y[3,3]]])
```

Use the `uint2exprs` and `int2exprs` function to convert integers to their binary encoding in unsigned, and twos-complement, respectively.

```
>>> uint2exprs(42, 8)
farray([0, 1, 0, 1, 0, 1, 0, 0])
>>> int2exprs(-42, 8)
farray([0, 1, 1, 0, 1, 0, 1, 1])
```

Note that the bits are in order from LSB to MSB, so the conventional bitstring representation of `-42` in eight bits would be "11010110".

Slicing

PyEDA's function arrays support numpy-style slicing operators:

```
>>> xs = exprvars('x', 4, 4, 4)
>>> xs[1,2,3]
xs[1,2,3]
>>> xs[2,:,2]
farray([x[2,0,2], x[2,1,2], x[2,2,2], x[2,3,2]])
>>> xs[...,1]
farray([[x[0,0,1], x[0,1,1], x[0,2,1], x[0,3,1]],
        [x[1,0,1], x[1,1,1], x[1,2,1], x[1,3,1]],
        [x[2,0,1], x[2,1,1], x[2,2,1], x[2,3,1]],
        [x[3,0,1], x[3,1,1], x[3,2,1], x[3,3,1]]])
```

A special feature of PyEDA `farray` slicing that is useful for digital logic is the ability to multiplex (mux) array items over a select input. For example, to create a simple, 4:1 mux:

```
>>> X = exprvars('x', 4)
>>> S = exprvars('s', 2)
>>> X[S]
Or(And(x[0], ~s[0], ~s[1]),
   And(x[1],  s[0], ~s[1]),
   And(x[2], ~s[0],  s[1]),
   And(x[3],  s[0],  s[1]))
```

Algebraic Operations

Function arrays are algebraic data types, which support the following symbolic operators:

- unary reductions (`uor, uand, uxor, ...`)
- bitwise logic (`~ | & ^`)
- shifts (`<< >>`)
- concatenation (+)
- repetition ($\star$)

Combining function and array operators allows us to implement a reasonably complete domain-specific language (DSL) for symbolic Boolean algebra in Python.

Consider, for example, the implementation of the `xtime` function, which is an integral part of the AES algorithm.

The Verilog implementation, as a `function`:

```
function automatic logic [7:0]
xtime(logic [7:0] b, int n);
    xtime = b;
    for (int i = 0; i < n; i++)
        xtime = {xtime[6:0], 1'b0}
              ^ (8'h1b & {8{xtime[7]}});
endfunction
```

And the PyEDA implementation:

```
def xtime(b, n):
    for _ in range(n):
        b = (exprzeros(1) + b[:7]
          ^  uint2exprs(0x1b, 8) & b[7]*8)
    return b
```

Practical Applications

Arrays of functions have many practical applications. For example, the `pyeda.logic.addition` module contains implementations of ripple-carry, brent-kung, and kogge-stone addition logic. Here is the digital logic implementation of $2+2=4$:

```
>>> from pyeda.logic.addition import kogge_stone_add
>>> A = exprvars('a', 8)
>>> B = exprvars('b', 8)
>>> S, C = kogge_stone_add(A, B)
>>> S.vrestrict({A: "01000000", B: "01000000"})
farray([0, 0, 1, 0, 0, 0, 0, 0])
```

Related Work

It is truly an exciting time for Python in digital logic. There are several available libraries implementing features that are competitive with PyEDA's.

SymPy was an early influence for PyEDA's design [Sympy]. It features a logic module that implements symbolic logic expressions. SymPy is implemented in 100% pure Python, and therefore will have some trouble competing with the raw performance of PyEDA's C extensions.

Another tremendous influence was Ilan Schnell's `pycosat` module [Pycosat]. It implements a similar Python interface to the PicoSAT SAT solver [Bie08], but does not delve into the area of symbolic Boolean algebra.

Steve Haynal and others at the University of California Santa Barbara have implemented PyCUDD, a Python binding to the well-known [CUDD] library.

The Sage Math project implements logic and sat modules with similar features to PyEDA's.

Lastly, there are a few notable Python bindings to other SAT libries. python-minisat, and pycryptosat implement Python wrappers around MiniSAT and CryptoMiniSAT, respectively. Also, Microsoft recently open sourced the truly excellent Z3 theorem prover library, which has its own SMT SAT solver and Python bindings.

REFERENCES

[Ake78] S.B. Akers, *Binary Decision Diagrams*, IEEE Transactions on Computers, Vol. C-27, No. 6, June 1978, pp. 509-516.

[Bah93] R. I. Bahar, E. A. Frohm, C. M. Gaona, G. D. Hachtel, E. Macii, A. Pardo, and F. Somenzi. *Algebraic Decision Diagrams and Their Applications*, Proceedings of the International Conference on Computer-Aided Design, pages 188-191, Santa Clara, CA, November 1993.

[Bie08] A. Biere. *PicoSAT Essentials*, Journal on Satisfiability, Boolean Modeling and Computation (JSAT), vol. 4, pages 75-97, Delft University, 2008.

[Bra84] R. Brayton, G. Hatchel, C. McMullen, and A. Sangiovanni-Vincentelli, *Logic Minimization Algorithms for VLSI Synthesis*, Kluwer Academic Publishers, Boston, MA, 1984.

[Bry86] R.E. Bryant. *Graph-based algorithms for Boolean function manipulation*, IEEE Transactions on Computers, C-35(8):677-691, August 1986. http://www.cs.cmu.edu/~bryant/pubdir/ieeetc86.pdf

[Dec04] J. Decaluwe. *MyHDL: A Python-based Hardware Description Language*, Linux Journal, November 2004. http://www.myhdl.org

[Fin14] B. Finkbeiner, L. Tentrup, *Fast DQBF Refutation*, SAT 2014 https://www.react.uni-saarland.de/tools/bunsat/

[Graphviz] Graphviz - Graph Visualization Software http://www.graphviz.org/

[Loc14] D. Lockhart, G. Zibrat, C. Batten. *PyMTL: A Unified Framework for Vertically Integrated Computer Architecture Research*, Int'l Symp. on Microarchitecture (MICRO-47), December 2014. http://csl.cornell.edu/~cbatten/pdfs/lockhart-pymtl-micro2014.pdf

[Min93] S.I. Minato. *Zero-suppressed BDDs for set manipulation in combinatorial problems*, In Proceedings of the Design Automation Conference, pages 272-277, Dallas, TX, June 1993.

[Nan14] S. Nannapaneni, et al. *A Model-Based Approach for Reliability Assessment in Component-Based Systems*, https://www.phmsociety.org/sites/phmsociety.org/files/phm_submission/2014/phmc_14_025.pdf

[Pycosat] Ilan Schnell https://github.com/ContinuumIO/pycosat/

[Ros03] K. Rosen. *Discrete Mathematics and its Applications* McGraw Hill, 2003.

[CUDD] F. Somenzi. *CUDD: CU Decision Diagram Package*, http://vlsi.colorado.edu/~fabio/CUDD/

[Sympy] Sympy - Python library for symbolic mathematics http://docs.sympy.org

[Lee59] C.Y. Lee, *Representation of Switching Circuits by Binary-Decision Programs*, Bell System Technical Journal, Vol. 38, July 1959, pp. 985-999.

[Tse68] G.S. Tseitin, *On the complexity of derivation in propositional calculus*, Slisenko, A.O. (ed.) Structures in Constructive Mathematics and Mathematical Logic, Part II, Seminars in Mathematics pp. 115–125. Steklov Mathematical Institute, 1968.

Scientific Data Analysis and Visualization with Python, VTK, and ParaView

Cory Quammen[‡*]

https://www.youtube.com/watch?v=8ugmkKaYKxM

Abstract—VTK and ParaView are leading software packages for data analysis and visualization. Since their early years, Python has played an important role in each package. In many use cases, VTK and ParaView serve as modules used by Python applications. In other use cases, Python modules are used to generate visualization components within VTK. In this paper, we provide an overview of Python integration in VTK and ParaView and give some concrete examples of usage. We also provide a roadmap for additional Python integration in VTK and ParaView in the future.

Index Terms—data analysis, scientific visualization, VTK, ParaView

Introduction

The Visualization Toolkit (VTK) is an open-source, freely available software system for 3D visualization. It consists of a set of C++ class libraries and bindings for Python and several other languages. VTK supports a wide variety of visualization algorithms for 2D and 3D scalar, vector, tensor, and volumetric data, as well as advanced algorithms such as implicit modeling, polygon reduction, mesh smoothing, cutting, contouring, and Delaunay triangulation. VTK has an extensive information visualization framework and a suite of 3D interaction widgets. The toolkit supports parallel processing and integrates with various GUI toolkits such as Qt. Python bindings expose nearly all VTK classes and functions, making it possible to write full VTK-based applications exclusively in Python. VTK also includes interfaces to popular Python packages such as NumPy and matplotlib. Support for writing custom VTK algorithms in Python is also available.

ParaView is a scalable visualization tool based on VTK that runs on a variety of platforms ranging from PCs to some of the largest supercomputers in the world. The ParaView package consists of a suite of executables for generating data visualizations using the techniques available in VTK. ParaView executables interface with Python in a number of ways: data sources, filters, and plots can be defined via Python code, data can be queried with Python expressions, and several executables can be controlled interactively with Python commands. Batch processing via Python scripts that are written either by hand or generated as a trace of events during an interactive visualization session is available for offline visualization generation.

* *Corresponding author: cory.quammen@kitware.com*
‡ *Kitware, Inc.*

This paper is organized into two main sections. In the first section, I introduce basic VTK usage, describe the relationship between VTK and Python, and describe interfaces between the two. In the second section, I detail the relationship between ParaView and Python. Examples of Python usage in VTK 6.2 and ParaView 4.3 are provided throughout. I also provide a roadmap for additional Python support in VTK and ParaView.

Python and VTK

VTK Data Model

To understand Python usage in VTK, it is important to understand the VTK data and processing models. At the most basic level, data in VTK is stored in a data object. Different types of data objects are available including graphs, trees, and data sets representing spatially embedded data from sensors or simulations such as uniform rectilinear grids, structured/unstructured grids, and Adaptive Mesh Refinement (AMR) data sets. This paper focuses on spatially embedded data sets.

Each spatially embedded data set consists of *cells*, each of which defines a geometric entity that defines a volume of space, and *points* that are used to define the vertices of the cells. Data values that represent a quantity, e.g. pressure, temperature, velocity, may be associated with both cells and points. Each quantity might be a scalar, vector, tensor, or string value. Vectors and tensors typically have more than one numerical *component*, and the quantity as a whole is known as a *tuple*.

The full collection of a quantity associated with points or cells is known by a number of names including "attribute", "field", "variable", and "array". VTK stores each attribute in a separate data array. For a point-associated array (point array), the number of tuples is expected to match the number of points. Likewise, for cell-associated arrays (cell array) the number of tuples is expected to match the number of cells.

VTK Pipeline

Data processing in VTK follows the data-flow paradigm. In this paradigm, data flows through a sequence of processing algorithms. These algorithms are chained together in a *pipeline*. At the beginning of a pipeline, a *source* generates a VTK data set. For example, an STL file reader source reads an STL file and produces a polygonal VTK data set as an output. A *filter* can be connected to the file reader to process the raw data from the file. For example, a smoothing filter may be used to smooth the polygonal data read by the STL reader. The output of the smoothing filter can be further

processed with a clipping filter to cut away part of the smoothed data set. Results from this operation can then be saved to a file with a file writer.

An algorithm in a pipeline produces one or more VTK data sets that are passed to the next algorithm in the pipeline. Algorithms need only update when one of their properties changes (e.g., smoothing amount) or when the algorithm upstream of it has produced a new data set. These updates are handled automatically by an internal VTK pipeline executive whenever an algorithm is updated.

Because VTK is intended to produce 3D interactive visualizations, output from the final algorithm in a pipeline is typically connected to a *mapper* object. A mapper is responsible for converting a data set into a set of rendering instructions. An *actor* represents the mapper in a scene, and has some properties that can modify the appearance of a rendered data set. One or more actors can be added to a *renderer* which executes the rendering instructions to generate an image.

Python Language Bindings for VTK

Since 1997, VTK has provided language bindings for Python. Over the years, Python has become increasingly important to VTK, both as a route to using VTK, as well as to the development of VTK itself.

The Python binding support in VTK has evolved so that today nearly every semantic feature of C++ used by VTK has a direct semantic analog in Python. C++ classes from VTK are wrapped into Python equivalents. The few classes that are not wrapped are typically limited to classes that are meant for internal use in VTK.

Python Wrapping Infrastructure

Python classes for VTK classes and special types are generated using a shared lex/yacc-based parser tailored for VTK programming conventions and custom code generation utilities for Python wrapping. VTK is organized into a number of C++ modules. When built with shared libraries enabled, a library containing C++ classes is generated at build time for each C++ module. Each Python-wrapped source file is likewise compiled into a shared library corresponding to the C++ module. All wrapped VTK C++ modules are provided in a single `vtk` Python package.

VTK Usage in Python

For convenience, an executable named `vtkpython` is provided in VTK binaries. This is the standard Python executable with environment variables set to make it simple to import the `vtk` package. It is also possible to use VTK in the same `python` executable from the Python installation against which VTK was built by prepending the location of VTK's shared libraries and the location of the parent directory of the file `vtk/__init__.py` to the `PYTHONPATH` environment variable, but using `vtkpython` avoids the need to do this.

To access VTK classes, you simply import `vtk`:

```
import vtk
```

VTK is somewhat unusual for a Python package in that all modules are loaded by this import statement.

Creation of VTK objects is straightforward:

```
contourFilter = vtk.vtkContourFilter()
```

Each Python object references an underlying VTK object. Objects in VTK are reference counted and automatically deleted when no longer used. The wrapping interface updates the underlying VTK object's reference count and alleviates the need for explicit memory management within Python.

One particularly nice semantic equivalence between VTK's C++ and Python interfaces involves member functions that accept a pointer to a C++ array representing a small tuple of elements. Such functions are common in VTK to do things like set a 3D Cartesian coordinate as a property of a class. In Python, the corresponding function accepts a tuple or list object. This works well as long as the list or tuple has the expected number of elements.

```
sphere = vtk.vtkSphereSource()

# Express point as list
sphere.SetCenter([0, 1, 0])

# Express point as tuple
sphere.SetCenter((0, 1, 0))
```

Member functions that return pointers to arrays with a fixed number of elements are also supported. Such functions require a hint to the wrapping infrastructure indicating how many elements are in the tuple that is returned.

```
>>> center = sphere.GetCenter()
>>> print center
(0, 1, 0)
```

For VTK classes that have operators <, <=, ==, >=, > defined, equivalent Python operators are provided.

Some functions in VTK return information via parameters passed by reference. For example, in the following code block, the parameter `t` is a return parameter from the member function `IntersectWithLine`.

```
double t, x[3]
plane->IntersectWithLine(point1, point2, t, x);
```

In Python, the equivalent is

```
t = vtk.mutable(0.0)
plane.IntersectWithLine(point1, point2, t, x)
```

Class and function documentation is processed by the wrapping infrastructure to make it available via Python's built-in help system.

```
>>> help(vtk.vtkSphereSource)
```

The above shows the full documentation of the `vtkSphereSource` class (too extensive to list here), while the code below produces help for only the `SetCenter` member function.

```
>>> help(vtk.vtkSphereSource.SetCenter)

Help on built-in function SetCenter:

SetCenter(...)
    V.SetCenter(float, float, float)
    C++: void SetCenter(double, double, double)
    V.SetCenter((float, float, float))
    C++: void SetCenter(double a[3])
```

Some less often used mappings between C++ and Python semantics, as well as limitations, are described in the file `VTK/Wrapping/Python/README_WRAP.txt` in the VTK source code repository in versions 4.2 and above.

A full example below shows how to create a VTK pipeline in Python that loads an STL file, smooths it, and displays the smoothed result in a 3D render window.

```
import vtk
```

```
reader = vtk.vtkSTLReader()
reader.SetFileName('somefile.stl')

smoother = vtk.vtkLoopSubdivisionFilter()
smoother.SetInputConnection(reader.GetOutputPort())

mapper = vtk.vtkPolyDataMapper()
mapper.SetInputConnection(smoother.GetOutputPort())

actor = vtk.vtkActor()
actor.SetMapper(mapper)

renderer = vtk.vtkRenderer()
renderer.AddActor(actor)

renWin = vtk.vtkRenderWindow
renWin.AddRenderer(renderer)

interactor = vtk.vtkRenderWindowInteractor()
interactor.SetRenderWindow(renWin)
interactor.Initialize()
renWin.Render()
iren.Start()
```

Many additional examples of VTK usage in Python are available in the VTK/Examples/Python wiki page [Wik15].

Integration with NumPy

There are limited functions within VTK itself to process or analyze point and cell arrays. Since 2008, a low-level interface layer between VTK arrays and NumPy arrays has been available in VTK. This interface layer can be used to map VTK arrays to NumPy arrays and vice versa, enabling the full power of NumPy operations to be used on VTK data. For example, suppose that we have a data set from a computational fluid dynamics simulation that we can load with a VTK reader class, and suppose further that the data set has a point array representing pressure. We can find several properties of this array using NumPy, e.g.,

```
import numpy as np
import vtk.util.numpy_support as nps

# Load data with a VTK reader instantiated earlier
reader.Update()

ds = reader.GetOutput()
pd = ds.GetPointData()
pressure = pd.GetArray('pressure')
np_pressure = nps.vtk_to_numpy(pressure)

min_p = np.min(np_pressure)
max_p = np.max(np_pressure)
```

This interface can also be used to add data arrays to loaded data sets that can be handed off to VTK for visualization:

```
norm_pressure = (np_pressure - min_pressure) / \
    (max_pressure - min_pressure)
vtk_norm_pressure = np.numpy_to_vtk(norm_pressure, 1)
vtk_norm_pressure.SetName('normalized pressure')
pd.AddArray(vtk_norm_pressure)
```

The second argument to `np.numpy_to_vtk` indicates that the NumPy array should be deep copied to the VTK array. This is necessary if no reference to the NumPy array will otherwise be kept. If a reference to the numpy array will be kept, then the second argument can be omitted and the NumPy array will be shallow copied instead, saving memory and time because the array data does not need to be copied. Note that the Python interpretter might crash if a NumPy array reference is not held and the data is shallow copied.

More recently, a higher-level NumPy-like interface layer has been added to VTK. This `numpy_interface` was designed to combine the ease of use of NumPy with the distributed memory parallel computing capabilities and broad data set type support of VTK. The straightforward interface between VTK data arrays and NumPy described above works only when the entire data set is available on one node. However, data sets in VTK may be distributed across different computational nodes in a parallel computer using the Message Passing Interface [Sni99]. In this scenario, global reduction operations using NumPy are not possible. For this reason, a NumPy-like interface has been added to VTK that properly handles distributed data sets [Aya14].

A key building block in VTK's `numpy_interface` is a set of classes that wrap VTK data set objects to have a more Pythonic interface.

```
import vtk
from vtk.numpy_interface import dataset_adapter as dsa

reader = vtk.vtkXMLPolyDataReader()
reader.SetFileName(filename)
reader.Update()
ds = dsa.WrapDataObject(reader.GetOutput())
```

In this code, `ds` is an instance of a `dataset_adapter.PolyData` that wraps the `vtkPolyData` output of the `vtkXMLPolyDataReader`. Point and cell arrays are available in member variables `PointData` and `CellData`, respectively, that provide the dictionary interface.

```
>>> ds.PointData.keys()
['pressure']

>>> pressure = ds.PointData['pressure']
```

Note that the `pressure` array here is an instance of `VTKArray` rather than a wrapped VTK data array. `VTKArray` is a wrapper around the VTK array object that inherits from `numpy.ndarray`. Hence, all the standard `ndarray` operations are available on this wrapped array, e.g.,

```
>>> pressure[0]
0.112

>>> pressure[1:4]
VTKArray([34.2432, 47.2342, 38.1211], dtype=float32)

>>> pressure[1:4] + 1
VTKArray([35.2432, 48.2342, 39.1211], dtype=float32)

>>> pressure[pressure > 40]
VTKArray([47.2342], dtype=float32)
```

The `numpy_interface.algorithms` module also provides NumPy-like functionality:

```
import vtk.numpy_interface.algorithms as algs

>>> algs.min(pressure)
VTKArray(0.1213)

>>> algs.where(pressure > 38)
(array([2, 3], dtype=int64),)
```

In addition to providing most of the ufuncs provided by NumPy, the `algorithms` interface provides some functions to access quantities that VTK can compute in the wide variety of data set types available in VTK. This can be used to compute, for instance, the total volume of cells in an unstructured grid:

```
>>> cell_volumes = algs.volume(ds)
>>> algs.sum(cell_volumes)
VTKArray(847.02)
```

This example illustrates nicely the power of combining a NumPy-like interface with VTK's uniform API for computing various quantities on different types of data sets.

Another distinct advantage of the `numpy_interface.algorithms` module is that all operations are supported in parallel when data sets are distributed across computational nodes. [Aya14] describes this functionality in more detail.

Integration with matplotlib

While VTK excels at interactive 3D rendering of scientific data, matplotlib excels at producing publication-quality 2D plots. VTK leverages each toolkit's strengths in two ways.

First, as described earlier, convenience functions for exposing VTK data arrays as NumPy arrays are provided in the `vtk.util.numpy_support` and `numpy_interface.algorithms` modules. These arrays can be passed to matplotlib plotting functions to produce publication-quality plots.

Second, VTK itself incorporates some of matplotlib's rendering capabilities directly when possible. When VTK Python wrapping is enabled and matplotlib is available, VTK uses the `matplotlib.mathtext` module to render LaTeX math expressions to either `vtkImageData` objects that can be displayed as images or to paths that may be rendered to a `vtkContextView` object, VTK's version of a canvas. The `vtkTextActor`, a class for adding text to visualizations, uses this module to support rendering complex LaTeX math expressions.

Qt applications with Python

Python support in VTK is robust enough to create full-featured applications without writing a single line of C++ code. PyQt [PyQt15] (or PySide [PyS15]) provide Python bindings for Qt. A simple PyQt example adapted from an example by Michka Popoff is provided below:

```
import sys
import vtk
from PyQt4 import QtCore, QtGui
from vtk.qt4.QVTKRenderWindowInteractor \
    import QVTKRenderWindowInteractor

class MainWindow(QtGui.QMainWindow):

    def __init__(self, parent = None):
        QtGui.QMainWindow.__init__(self, parent)

        self.frame = QtGui.QFrame()

        layout = QtGui.QVBoxLayout()
        self.vtkWidget = \
            QVTKRenderWindowInteractor(self.frame)
        layout.addWidget(self.vtkWidget)

        self.renderer = vtk.vtkRenderer()
        rw = self.vtkWidget.GetRenderWindow()
        rw.AddRenderer(self.renderer)
        self.interactor = rw.GetInteractor()

        cylinder = vtk.vtkCylinderSource()
        mapper = vtk.vtkPolyDataMapper()
        mapper.SetInputConnection( \
            cylinder.GetOutputPort())
        actor = vtk.vtkActor()
        actor.SetMapper(mapper)

        self.renderer.AddActor(actor)
        self.renderer.ResetCamera()

        self.frame.setLayout(layout)
        self.setCentralWidget(self.frame)

        self.show()
        self.interactor.Initialize()

if __name__ == "__main__":
    app = QtGui.QApplication(sys.argv)
    window = MainWindow()
    sys.exit(app.exec_())
```

This simple application does little besides what is possible with pure VTK code alone. However, this example can easily be expanded to provide interaction through UI elements such as a menu bar, buttons, text entries, sliders, etc.

VTK filters defined in Python

While VTK sources and filters are available in Python, they cannot be subclassed to create new sources or filters because the virtual function table defined in C++ cannot dispatch to member functions defined in Python. Instead, one can subclass from a special `VTKAlgorithm` class defined in `vtk.util.vtkAlgorithm`. This class specifies the interface for classes that interact with `vtkPythonAlgorithm`, a C++ class that delegates the primary VTK pipeline update functions to equivalent pipeline update functions in the Python `VTKAlgorithm` class. Subclasses of `VTKAlgorithm` can (and usually should) override these functions. By doing this, it is possible to implement complex new sources and filters using Python alone. For more details on the `VTKAlgorithm` class, see [Gev2014].

Python integration in VTK tests

As a project that follows a quality software process, VTK has many regression tests. At present, 26% of tests (544 out of 2046) are written in Python. This integration of Python in VTK's testing infrastructure shows how important Python is in VTK's development.

Obtaining VTK

VTK and its Python bindings are available on many Linux distributions including Ubuntu, Debian, OpenSUSE. It is also available in Anaconda and Enthought Canopy. Binary installers and source code for the most recent versions are available on the VTK web site [VTK15] for Windows, Mac, and Linux.

Python and ParaView

ParaView is a suite of scalable parallel visualization executables that use VTK to read data, process it, and create visualizations. One of the executables includes a graphical user interface (GUI) to make it possible to create visualizations without programming (when ParaView is mentioned in this section, it is the executable with a GUI unless otherwise specified). Data processing in ParaView follows the same data-flow paradigm that VTK follows. In ParaView, sources and filters are chained together in a Pipeline Browser as shown in Figure 1. Visualization controls are modified with user interaction widgets provided by Qt.

While ParaView can be used to make visualizations without programming, it is also possible to use Python scripting to automate certain operations or even create entire visualizations. In this section, I describe how Python scripting is integrated into

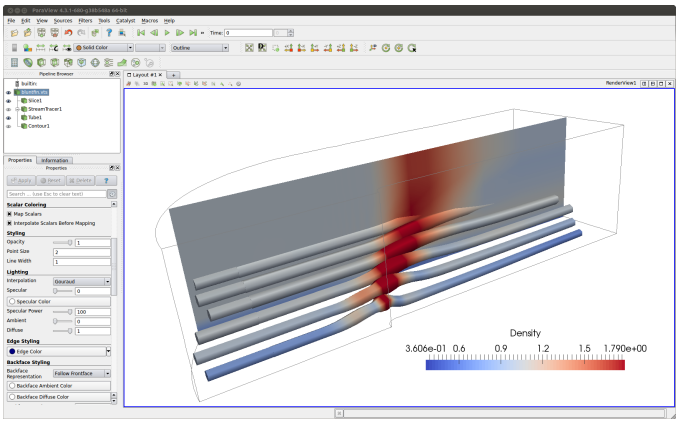


Fig. 1: *The ParaView GUI with an example visualization of a data set from a simulation of airflow past a blunt fin. The Pipeline Browser (upper left) shows the sources and filters used to create the visualization. Filter and visualization parameters are shown in the Property window (lower left).*

ParaView at several different levels. At a high level, Python commands are issued via a console to change properties of a visualization. At a lower level, Python commands are used to set up entire visualizaion pipelines. At an even lower level, Python is used to create custom sources and filters to provide additional data analysis and visualization functionality.

Python Console

ParaView includes a Python console available under the `Tools -> Python Console` menu item. This console is a fully-featured Python console with the environment set up so that the `vtk` package and a `paraview` package are available. When first started, the command

```
from paraview.simple import *
```

is automatically executed to import the `paraview.simple` module. This layer is described in more detail later.

Running commands in ParaView's Python console is identical to running commands in other Python consoles. The key difference is that commands can be used to change the state of the ParaView application. This provides a similar experience to using a Python console to change matplotlib plots.

The Python console also provides a button to load and execute a Python script with ParaView commands from a file. This feature is ideal for iterative Python script development.

pvpython and pvbatch

The ParaView suite of tools includes two Python-based utilities for both interactive and batch generation of visualizations. `pvpython` is an interactive Python shell that provides the same access to the `vtk` and `paraview` packages as provided by the Python console in ParaView. The key difference between ParaView and `pvpython` is that no GUI controls are available to modify pipeline or visualization state. `pvbatch` is a non-interactive executable that runs a Python script and is intended to perform offline data processing and visualization generation.

Python Tracing and State Files

While documentation is available to learn how to write Python scripts for ParaView, it can take some time to find the function calls needed to replicate a sequence of actions performed through the GUI. To reduce script development time, ParaView supports tracing of user interactions where the generated trace is in the form of a Python script. Running the resulting trace script through the ParaView Python console, `pvpython` or `pvbatch` reproduces the effects of the user interactions with the GUI.

Python tracing is implemented by instrumenting the ParaView application with Python generation code at various user event handlers. The tracing mechanism can record either the entire state of ParaView objects or just modifications of state to non-default values to reduce the trace size. Traces can be started and stopped at any time - they do not need to record the full user interaction history.

An application where tracing is useful is the batch conversion of data files. If ParaView can read the source file format and write the destination file format, it is easy to perform the conversion manually one time with the ParaView GUI. For a large list of files, though, a more automated approach is useful. Creating a trace of the actions needed to perform the conversion of a single file produces most of the script that would be needed to convert a list of files. The trace script can then be changed to apply to a list of files.

In addition to saving a trace of user interaction sequences, a Python *state file* may also be produced. Like a Python trace, the state file contains Python commands that set up the pipeline and visualization settings, but unlike a trace, it does not record interaction events as they happen but rather the final state of ParaView.

Simple Python Interface

Much of ParaView is implemented in C++ as VTK classes. These classes are wrapped in Python with the same mechanism that wraps VTK classes. As such, they are accessible within the Python console, `pvpython`, and `pvbatch`. However using these classes directly is often unwieldy. The example below illustrates how to use the direct ParaView API to create a sphere source with radius 2.

```
from paraview import servermanager as sm

pm = sm.vtkSMProxyManager.GetProxyManager()
controller = \
    sm.vtkSMParaViewPipelineControllerWithRendering()

ss = pm.NewProxy('sources', 'SphereSource')
ss.GetProperty('Radius').SetElement(0, 2.0)
controller.RegisterPipelineProxy(ss)

view = pm.GetProxy('views', 'RenderView1')
rep = view.CreateDefaultRepresentation(ss, 0)
controller.RegisterRepresentationProxy(rep)
rep.GetProperty('Input').SetInputConnection(0, ss, 0)
rep.GetProperty('Visibility').SetElement(0, 1)

controller.Show(ss, 0, view)
view.ResetCamera()
view.StillRender()
```

Note in this example the various references to proxies. A *proxy* here refers to the proxy programming design pattern where one object provides an interface to another object. Proxies are central to ParaView's design. In a number of the various client/server configuration in which ParaView can be run, the client software running on a local workstation connects to a remote server running one or more processes on different nodes of a high-performance computing resource. Proxies for each pipeline object exist on the ParaView client, and they provide the interface for communicating state to all the VTK objects in each client and server process.

In the example above, a new proxy for a `vtkSphereSource` object is created. This proxy has a property named 'Radius' that is modified to the value 2.0. Changes to the 'Radius' property are forwarded to the 'Radius' property of the underlying `vtkSphereSource`.

As this example demonstrates, creating a new data source, a representation for it (how it is rendered), and adding the representation to the view (where it is rendered), is an involved process when using the `paraview.servermanager` module directly. Fortunately, ParaView provides a simplified Python interface that hides most of these details, making Python scripting much more accessible.

The `paraview.simple` layer provides simpler Python functions to create pipelines and modify filter and visualization properties. The same example above expressed with `paraview.simple` functions is reduced to

```
from paraview import simple

Sphere(Radius=2.0)
Show()
Render()
```

ParaView traces and Python state files are expressed in terms of `paraview.simple` module functions. For more information on how to use this module, see [Kit15].

Python Programmable Filter

ParaView provides many data filters for transforming data and performing analysis tasks. There are, however, an infinite number of operations one may want to perform on a data set. To address the need for custom filters, ParaView supports a rich plugin architecture that makes it possible to create additional filters in C++. Unfortunately, creating a plugin this way is a relatively involved process.

Aside from the C++ plugin architecture, ParaView provides a Programmable Filter that enables a potentially faster development path. The Programmable Filter has a text property that stores a Python script to execute when the filter is updated. Inputs to the Programmable Filter are available within this script. Complete specification of the output data set is possible within the script, including setting the output data type, the data set toplogy (i.e., type and number of cells), as well as point and cell arrays.

At its core, the Programmable Filter is defined by the VTK-derived C++ class named `vtkPythonProgrammableFilter`. Using the Python C API, the `vtkPythonProgrammableFilter` passes a reference to itself to the Python environment in which the script executes so that it is available within the script itself. This makes it possible to access the inputs and outputs to the filter via:

```
input = self.GetInput()
output = self.GetOutput()
```

Arbitrarily complex Python scripts can be executed to generate the filter's output. The following example moves points in an input `vtkPointSet` along normals associated with the points if available.

```
ipd = self.GetInput()
opd = self.GetOutput()

# Output is shallow-copied by default
# Deep copy the points so that we are not modifying
# the input points.
opd.DeepCopy(ipd)

na = ipd.GetPointData().GetArray('Normals')
if na != None:
    for i in xrange(ipd.GetNumberOfPoints()):
        pt = ipd.GetPoint(i)
        n = na.GetTuple(i)
        newPt = (pt[0]+n[0], pt[1]+n[1], pt[2]+n[2])
        opd.GetPoints().SetPoint(i, newPt)
```

The Programmable Filter also uses the `vtk.numpy_interface.dataset_adapter` module to wrap the inputs to the filter. All of the wrapped inputs are added to a list named `inputs`, and the single output is wrapped in an object named `output`. By using the wrapped inputs and outputs, the filter above becomes simply

```
ipts = inputs[0].Points
normals = inputs[0].PointData['Normals']

output.Points = ipts + normals
```

It is important to note that Python scripts in the Programmable Filter may use only VTK classes and other Python modules, but not any of the modules in the `paraview` package. If those modules are imported, the behavior is undefined.

Python Programmable Source

Within ParaView it is also possible to define Python script that defines data sources using the Python Programmable Source. This source functions much like the Python Programmable Filter, but does not require any input data sets.

Python Calculator

ParaView's Python Calculator filter is a light-weight alternative to the Programmable Filter used to compute additional point or cell arrays using NumPy or the `numpy_interface.algorithms` module. The following expression computes the areas of polygons in a surface mesh:

```
algs.area(inputs[0])
```

Note that the `numpy_interface.algorithms` is imported with the name `algs` in the Python environment in which the expression is evaluated. In the Python Calculator, the property 'Array Association', which indicates whether the output array should be a point or cell array, must be set to 'Cell Data' because one area value is produced per cell. Note that like the Programmable Filter, the inputs are wrapped with the `vtk.numpy_interface.dataset_adapter` module functions and stored in an `inputs` list.

Python Annotation

It is often desirable to annotate visualizations with numerical values taken either directly from the data set or computed from the data. The Python Annotation filter in ParaView provides this capability in a convenient way. The filter takes a Python expression that is evaluated when the filter is executed and the value returned by the expression is displayed in the render view. Importantly, these annotations can come from data analysis results from NumPy or `numpy_interface.algorithms`. Figure 2 shows an example using the Python Annotation filter.

Python View

While ParaView's roots are in the loading and display of traditional 3D scientific visualizations, it has grown over the years to support more data set types and different displays of those data set types. These different displays, or "Views" in ParaView

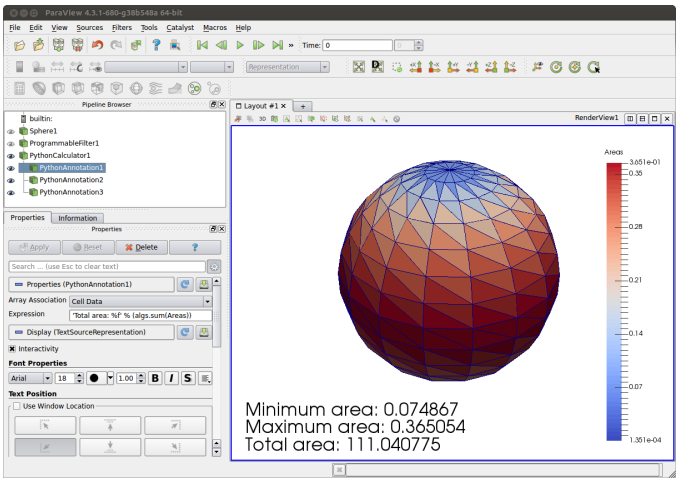


Fig. 2: *Three annotations filters in the scene show the minimum, maximum, and total areas of polygons in the sphere source.*

parlance, include a 3D interactive rendering view, a histogram view, a parallel coordinates view, and a large number of others.

One of these other view types is the Python View. This view is similar to the programmable filter in that the user supplies a Python script that generates some data. In the case of the Python View, the data that is generated is an image to display in the ParaView window. This makes it possible to use Python plotting packages, such as matplotlib, to generate plots to be displayed directly in ParaView.

Scripts used in the Python view are required to define two functions, a `setup_data` function and a `render` function. Rendering in the Python view is done on the local client, so data that resides on remote server processes must first be brought over to the client. Because data sets may be larger than the client's RAM, only a subset of the data arrays in a data set are copied to the client. By default, no arrays are copied. Arrays can be requested using functions available in the `vtkPythonView` class instance that is passed in as an argument to the `setup_data` function, e.g.,

```
def setup_data(view):
    view.SetAttributeArrayStatus(0, \
        vtkDataObject.POINT, "Density", 1)
```

The actual generation of the plot image is expected to be done in the `render` function. This function is expected to take the same `view` object as is passed to the `setup_data` function. It also takes a width and height parameter that tells how large the plotted image should be in terms of pixels. This function is expected to return an instance of `vtkImageData` containing the plot image. A few utilities are included in the `paraview.python_view` module to convert Python arrays and images to `vtkImageData`. An example that creates a histogram of an array named "Density" is provided here:

```
def render(view, width, height):
    from paraview \
        import python_view.matplotlib_figure
    figure = matplotlib_figure(width, height)

    ax = figure.add_subplot(1,1,1)
    ax.minorticks_on()
    ax.set_title('Plot title')
    ax.set_xlabel('X label')
    ax.set_ylabel('Y label')

    # Process only the first visible object in the
    # pipeline browser
    do = view.GetVisibleDataObjectForRendering(0)

    dens = do.GetPointData().GetArray('Density')

    # Convert VTK data array to numpy array
    from paraview.numpy_support import vtk_to_numpy

    ax.hist(vtk_to_numpy(dens), bins=10)

    return python_view.figure_to_image(figure)
```

For more information on the Python View, see Section 4.11 in [Aya15] or [Qua13].

ParaViewWeb

ParaViewWeb is a framework for remote VTK and ParaView processing and visualization via a web browser. The framework on the server side is based on the Autobahn, Twisted, Six, and ZopeInterface Python libraries. On the client side, ParaViewWeb provides a set of JavaScript libraries that use WebGL, JQuery, and Autobahn.js. Images are typically generated on the server and sent to the client for display, but if the visualized geometry is small enough, geometry can be sent to the client and rendered with WebGL.

A nice feature of ParaViewWeb is that the server component can be launched with `pvpython`. No separate web server is needed. For example, on Linux, the following command launches the ParaViewWeb server from the ParaView installation directory

```
./bin/pvpython                                  \
   lib/paraview-4.1/site-packages/paraview/\
   web/pv_web_visualizer.py --port 8080         \
         --content ./share/paraview-4.1/www \
         --data-dir /path-to-share/ &           \
```

Once the server is running, it can be accessed through a web browser at the URL http://localhost:8080/apps/Visualizer. This is one example application that comes with the framework. It has much of the same functionality as the ParaView desktop application. ParaViewWeb can also be used to display images within an iPython notebook. For additional information about using and extending the ParaViewWeb framework, see [Pvw15].

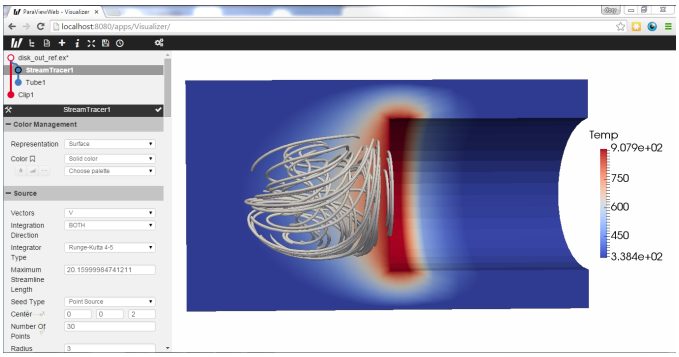


Fig. 3: *The ParaViewWeb Visualizer application web interface.*

Unified Server Bindings

As previously discussed, ParaView uses proxies to manage state among VTK class instances associated with pipeline objects on distributed process. For example, when the proxy for a cross-section filter has its cutting plane property changed, the underlying VTK filter on each process is updated so that is has the same

cutting plane. These instances are updated via a client/server communication layer that is generated automatically using a wrapping mechanism. The client/server layer consists of one communication class per VTK class that serializes and deserializes state in the VTK class.

As discussed, a similar wrapping process is also performed to generate Python bindings for VTK classes and ParaView classes. Each of these wrappings adds to the size of the executable files and shared libraries. On very large scale parallel computing resources, the amount of RAM available per node can be relatively limited. As a result, when running ParaView on such a resource, it is important to reduce the size of the executables as much as possible to leave room for the data. One way to do this is to use the Python wrapping to communicate among processes instead of using the client/server communication class. Indeed, when this option is enabled, the process of creating the special communication classes is skipped. Instead, communication is performed by sending strings with Python expressions to destination processes. These expressions are then evaluated on each process to change the state of local VTK classes. In this approach, we get the same functionality as the custom client/server communication layer wrapping, but with smaller executables.

Conclusions

Python has been integrated into VTK and ParaView for many years. The integration continues to mature and expand as Python is used in an increasing number of ways in both software packages. As Python continues to grow in popularity among the scientific community, so too does the need for providing easy-to-use Pythonic interfaces to scientific visualization tools. As demonstrated in this paper, VTK and ParaView are well-positioned to continue adapting to the future needs of scientific Python programmers.

Future Work

VTK and ParaView currently support Python 2.6 and 2.7. Support for Python 3 is targeted for sometime in 2016.

Acknowledgements

Contributions to Python support in VTK and ParaView have come from many VTK community members. Deserving special recognition are key contributors David Gobbi, Prabhu Ramachandran, Ken Martin, Berk Geveci, Utkarsh Ayachit, Ben Boeckel, Andy Cedilnik, Brad King, David Partyka, George Zagaris, Marcus Hanwell, and Mathieu Malaterre.

References

[Aya14] U. Ayachit, B. Geveci, *Scientific data analysis and visualization at scale in VTK/ParaView with NumPy*, 4th Workshop on Python for High Performance and Scientific Computing PyHPC 2014, November, 2014.

[Aya15] U. Ayachit, *The ParaView Guide: A Parallel Visualization Application*, Kitware, Inc. 2015, ISBN 978-1930934306.

[Gev14] B. Geveci, *vtkPythonAlgorithm is great*, Kitware Blog, September 10, 2014. http://www.kitware.com/blog/home/post/737

[Kit15] *simple Module*, http://www.paraview.org/ParaView/Doc/Nightly/www/py-doc/paraview.simple.html

[Pvw15] *ParaViewWeb*, http://paraviewweb.kitware.com/#!/guide

[PyQt15] *PyQt4 Reference Guide*, http://pyqt.sourceforge.net/Docs/PyQt4/

[PyS15] *PySide 1.2.2*, https://pypi.python.org/pypi/PySide

[Qua13] C. Quammen. *ParaView: Python View is now more versatile*, http://www.kitware.com/blog/home/post/704

[Sch04] W. Schroeder, K. Martin, and B. Lorensen, *The Visualization Toolkit: An Object-Oriented Approach to 3D Graphics*, 4th ed. Kitware, Inc., 2004, ISBN 1-930934-19-X.

[Sni99] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra, *MPI - The Complete Reference: Volume 1, The MPI Core*, 2nd ed., MIT Press, 1999, ISBN 0-262-69215-5.

[VTK15] *VTK - The Visualization Toolkit*, http://www.vtk.org/

[Wik15] *VTK/Examples/Python*, http://www.vtk.org/Wiki/VTK/Examples/Python

Creating a Real-Time Recommendation Engine using Modified K-Means Clustering and Remote Sensing Signature Matching Algorithms

David Lippa‡*, Jason Vertrees‡

Abstract—Built on Google App Engine (GAE), RealMassive encountered challenges while attempting to scale its recommendation engine to match its nationwide, multi-market expansion. To address this problem, we borrowed a conceptual model from spectral data processing to transform our domain-specific problem into one that the GAE's search engine could solve. Rather than using a more traditional heuristics-based relevancy ranking, we filtered and scored results using a modified version of a spectral angle. While this approach seems to have little in common with providing a recommendation based on similarity, there are important parallels: filtering to reduce the search space; independent variables that can be resampled into a signature; a signature library to identify meaningful similarities; and an algorithm that lends itself to an accurate but flexible definition of similarity. We implemented this as a web service that provides recommendations in sub-second time. The RealMassive platform currently covers over 4.5 billion square feet of commercial real estate inventory and is expanding quickly.

Index Terms—algorithms, clustering, recommendation engine, remote sensing

Introduction

RealMassive helps tenants and their representatives locate commercial real estate (CRE) space to lease or buy. Finding suitable space in a market can be difficult. Each tenant has specific requirements, and often, the knowledge of the current market lives exclusively in the memory of domain experts. Automated recommendation tools have substantial value, provided that they operate in real time on an ever-increasing dataset and provide similar or better accuracy to the judgment of domain experts. Our initial recommendation engine attempted to use a variance-based calculation that could not scale to match our growing database of CRE listings, which now covers more than 30 US markets and 4.5 billion square feet. We set out to create a new real-time recommendation engine to meet these needs while negotiating the restrictions of our platform, Google App Engine (GAE). This is a classic problem of pattern matching and information retrieval adapted to a specific domain of expertise coupled with engineering restrictions and product requirements.

GAE is a powerful platform built to scale, yet it brings along certain challenges that make implementing algorithms, such as a recommendation engine, more difficult. Several of these constraints are particularly difficult to overcome. Instances are outfitted with at most 1 GB of memory and prohibited from executing native code with the exception of a few provided libraries, such as numpy [Goo15]. Though fast for relevance-based search operations, the GAE search engine trades speed for limited functionality: only a small subset of mathematical functions (addition, subtraction, multiplication, division, minimum, maximum, geographical distance, natural logarithm, and absolute value) are available [Goo15]. Implementing algorithms via the GAE search infrastructure keeps memory usage low, provided that the only functionality needed is a very limited math toolbox.

We set out to implement our recommendation engine using GAE search to produce a solution that fits within the constraints of our platform. The search results are ordered not by search term relevance, but by a modified version of a spectral angle—a simple computation borrowed from the domain of linear algebra and spectral analysis. The Spectral Angle Mapper (SAM) algorithm treats each pixel of an image as an n-dimensional vector $\vec{v_{ij}}$ and computes the angle θ between $\vec{v_{ij}}$ and a vector $\vec{s}$ for all rows i and all columns j: $\cos^{-1}\left(\frac{\vec{s}\cdot\vec{v_{ij}}}{|\vec{s}||\vec{v_{ij}}|}\right)$. A potential candidate match usually has an angle between 5 and 10 degrees, while a collinear match has an angle of 0. For remote sensing applications, SAM has a roughly 83% accuracy rate when predicting exact signature matches [Pet11] in a variety of applications and domains including: determining the chemical composition of stars [Ric15], analyzing the health of vegetation [Zha09], measuring the quality of an RGB image, and detecting camouflage in times of war [Lan05]. Unfortunately, one of the weaknesses of the SAM algorithm is that a collinear match can show up as a false positive[1], requiring additional algorithmic steps that takes vector magnitude into account.

We can draw some important parallels between the recommendation algorithm and SAM. The "pixels" of an image are similar to the pool of candidates to match against. User inputs, which in our case are spaces added to a CRE survey, can represent a library of "signatures," with the intensity of each signature component taking its value from each item's orthogonal attributes. The dependence between variables, such as cost per unit, number

* Corresponding author: david.lippa@realmassive.com
‡ RealMassive, Inc.

1. A signature whose vector is $(1,2,3)$ is an exact match when compared against a candidate pixel of $(10,20,30)$, since they are collinear and therefore the angle between them is 0.

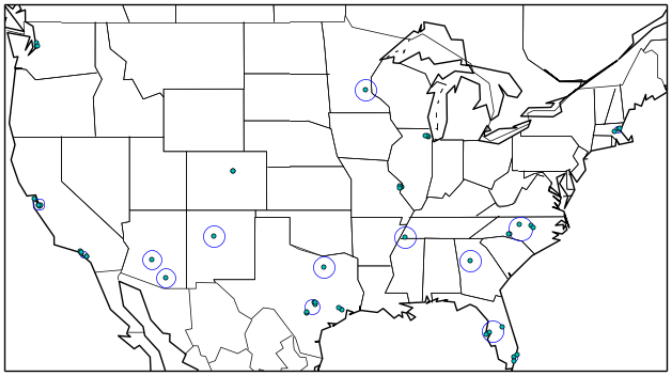


Fig. 1: *Clustering of 50 Spaces from across the US [Rea15].*

of units, and total cost, are "within the same wavelength range," comparable to a spectral sensor's center frequency and full-width half maximum[2]. The attributes contained within an object are like the wavelengths of a spectral signature, provided each vector component is independent[3] of all others. Furthermore, the false positives observed from using SAM in remote sensing applications translates into an asset when used as a recommendation score: an angle of 0, regardless of vector magnitude, is indicative of similarity.

Method

The implementation of our recommendation engine transforms a domain-specific problem into a modified version of SAM. This expands the potential use of Google App Engine to solve a subset of linear algebra based problems using its search engine. There are three phases of the algorithm: dynamically clustering user data to produce signatures, applying fixed filters to limit search results, and scoring those results based on a signature match instead of search term relevance. Each are necessary to overcome one or more constraints imposed by GAE: clustering reduces the length of query strings and sort expressions, which are restricted to 2,000 and 5,000 characters respectively, and filtering keeps the results within the 10,000 hit sort limit [Goo15].

The first phase starts with k-means clustering: the process of breaking up n data points into k discrete clusters. Traditionally, one divides the n data points into an initial set of clusters, whose new center points are calculated as an unweighted average, causing the clusters to shift in response to the distribution of their contents. Iteratively, clusters are merged and center points are re-calculated until no cluster intersects any other [Vat11]. We chose not to use an iterative approach which has a worst-case complexity of $2^{\Omega(n)}$ even in 2 dimensions [Vat11]. Though the worst case scenario doesn't seem to arise in practice, we use a quick guess-and-check method that has good asymptotic complexity and converges quickly, even though other algorithms may produce better results. The algorithm takes advantage of a few known attributes of the data: there is a limited amount of overlap between data points because they represent physical objects in 3-dimensional space; the data points have a limited range since they are latitude and longitude coordinates; and since we use the clusters as a geofence in our search parameters, using a global KDTree of all building coordinates in our datastore allows us to make a good estimation of the initial cluster sizes. The algorithm executes as follows:

1) Create a set P of points $p_1, p_2, \ldots, p_n$, each representing an office space.
2) Create a KDTree K using the set P.
3) Iterating while P is not empty, take the first point p_i and compute the radius r_i of the circle containing the nearest 50 neighboring buildings using a pre-built SciPy KDTree[4] with a starting maximum distance $d = 0.082° \approx$ 9 km. Using K, find all nearest neighbors within r_i, adding them to cluster c_i and removing them from P. Merge c_i if it intersects any other cluster.
4) If the number of clusters is greater than k, recursively perform the previous step with the original set P and $2d$ as the new maximum distance. Otherwise, merge intersecting clusters and compute a weighted centroid and radius for each cluster.

The maximum number of recursive calls is determined by the maximum distance between latitude and longitude points, which if treated as cartesian coordinates, is $\sqrt{180^2 + 360^2} \approx 402.5$, and would have at most 26 calls[5] when starting with an initial radius of 1 meter $\approx 9 \cdot 10^{-6}$ degrees. This never happens in practice, since we take the nearest 50 buildings to compute the starting radius. At worst, the radius, at its smallest, falls between 0.5 and 1 km, which would result in at most 17 recursive calls. The worst case has a high constant, but is still asymptotically acceptable at $O(kn \log_k n)$. Since building the KDTree takes $O(kn \log_k n)$ time [Man01] and the clustering algorithm requires at most 26 passes, each computing at most n lookups in the KDTree per pass at a total cost of $26n \log_k n$ operations, the overall asymptotic complexity is unchanged. The final result is similar to the mapless representation of clusters shown in Fig 1. Once the spaces have been clustered, it is trivial to compute each cluster's aggregated characterization, such as an average of each vector component, to produce its signature $\vec{s_k}$.

The next part of the algorithm involves applying fixed filters informed by domain expertise. For commercial real estate, this includes the building type (such as "office", "industrial", etc.) and location, along with any necessary exclusions[6]. These constraints produce a reasonably sized subset of no more than 10,000 results that can be matched against the signatures generated during the clustering phase.

Executing the SAM algorithm on a reduced dataset of 10,000 items is comparable to performing material identification on a 115 x 87 pixel data collection[7] from a 3-band multi-spectral sensor, easily accomplished in sub-second time. The sample Python code below illustrates the process of executing SAM on a 2-dimensional array of pixels in $\mathbb{R}^3$:

```
from math import acos
import numpy as np
```

2. The full-width half maximum value for a band expresses the difference between the extreme values that the center frequency could detect. For example, QuickBird detects blue in the range of 446-512, whose center frequency is 478 [War09].

3. When attributes are not independent of each other, some elements of the signature are over-represented, skewing results. Mixing different types of units, such as rates and values, is another form of variable dependence.

4. Since GAE restricts external libraries to be purely implemented in Python, we have stripped out functionality from kdtree.py [Git15] that depends on native code. Since the building KDTree is limited in size, we maintain a KDTree singleton that periodically updates, following GAE's guidelines of eventual consistency.

5. The maximum number of iterations is determined by $\lceil \log_2 (\frac{402.5}{x}) \rceil$, where x is the starting radius.

```
def SAM(img, sig):
  """
  >>> sig = [2, 2, 2]
  >>> img = np.array([[(1, 2, 3), (1, 1, 0)],
                      [(4, 3, 2), (0, 1, 1)],
                      [(1, 1, 1), (4, 4, 1)]])
  >>> SAM(img, sig)
  """
  matches = []
  sig_norm = sig/np.linalg.norm(sig)
  for r in range(len(img)):
    for c in range(len(img[r])):
      pix = img[r][c]
      cos_t = pix.dot(sig_norm)/np.linalg.norm(pix)
      theta = acos(round(cos_t, 7))
      if theta < .1745329:  # 10 degrees, in radians
        matches.append((r, c, theta))
  return sorted(
    matches,
    cmp=lambda x, y: cmp(x[-1], y[-1]))
```

This solution fails our speed requirement, since it requires loading the subset of candidates into memory and sorting the results. GAE's search service provides a faster mechanism in the form of a sort expression, but it lacks the inverse cosine function [Goo15]. Our solution uses the cosine ratio as a proxy for the angle. Since the components $s_1, s_2, \ldots, s_n$ of a signature vector $\vec{s}$ and the components of all of the candidate vectors $\vec{v_1}, \vec{v_2}, \ldots, \vec{v_n}$ are all non-negative, the cosine ratio between $\vec{s}$ and $\vec{v_i} = \frac{\vec{s} \cdot \vec{v_i}}{|\vec{s}||\vec{v_i}|} \in [0,1]$ and is monotonically increasing. From this, we can deduce $\theta \in [0, \frac{\pi}{2}]$ and is monotonically decreasing[8]. This means that sorting by the cosine ratio in descending order is functionally equivalent to sorting by the angle θ in ascending order to find the best match.

Results

From design to production, the recommendation engine took 3 weeks to complete, and in practice, has been performant, executing on every page view of a space [Rea15] and on-demand in the survey view. To date, it has generated more than 302,925 recommendations, sifting through over 80,000 spaces at sub-second speeds. The workload averaged in the thousands per work day, with loads peaking at 18,327 per day and 1,407 per minute[9]. These speeds were reached when deployed as part of the RealMassive API[10] on F4_1G instances, each outfitted with a 2.4 Ghz proecssor, 1 GB RAM, and configured with automatic scaling on a per-request basis [Goo15]. Benchmarks of the GAE search service showed that search queries limited to 100 results clocked in between 6-600 ms depending on caching and query complexity. The clustering and SAM algorithm added up to 200 ms, mostly in the form of reads from the datastore prior to building clusters. At 6-800 ms, GAE performed nearly 8 times slower than consumer hardware[11] but scaled well during traffic spikes. Recently, we performed a stress test outside of a standard use case: 80+ recommendations generated from 100+ user inputs lagged at an unacceptable 3 seconds—a performance hit caused by returning entire objects rather than utilizing a projection query, an optimization that would lower response time back to sub-second speeds.

Future Work

There are three improvements that we can make to the recommendation algorithm. First, we can use a 3-dimensional projection for the geo-coordinates rather than cartesian coordinates. Earth-Centered Earth-Fixed coordinates would make nearest-neighbor calculations in the KDTree more accurate, especially with extreme coordinates that are adjacent, but do not appear to be so when represented in 2 dimensions. Second, we can consider generating the clusters in parallel using a tasklet [Goo15]. And lastly, we may investigate other cluster signature calculations, though averaging vector components works well in practice and is simple to implement.

Conclusions

Google App Engine provides a scalable infrastructure with an advanced search engine that can be utilized for purposes beyond typical search use cases. In this paper, we present a novel approach to recommendation systems by drawing parallels between domain-specific recommendation matching and material identification processes used in remote sensing. Remapping independent object attributes into vectors allows for sub-second scoring and sorting. This implementation enables RealMassive to scale its recommendation engine and continue to innovate in an industry that is currently hampered by closed data and its dependence upon a select few domain experts.

Though our specific problem is a case of pattern matching, the strategy of leveraging, rather than fighting, constraints can produce innovations that prefer satisficing over optimizing [Bra00]. Rather than simply considering only the input dataset, we used a related dataset to inform initial cluster sizes, sacrificing some speed in the average case to put an upper bound on the worst-case. Instead of continuing to use a variance-based approach to signature matching, the simpler Spectral Angle Mapper suffices for positive vectors whose magnitude are irrelevant. The seemingly restrictive toolbox provided by Google App Engine became a catalyst for a mathematically simpler solution that yielded the speed and accuracy required. Our experience with implementing a recommendation engine on Google's App Engine platform shows that the structure, and not just the content, of a problem is significant, and may be the key to a new breed of solutions.

Acknowledgments

The authors would like to thank Fatih Akici, Natalya Shelburne, and Hannah Kocurek for providing suggestions and edits for this paper.

Appendix

For a demonstration of RealMassive's clustering service used by the recommendation engine, you may use the search query language described in our Apiary documentation with the clustering endpoint *https://www.realmassive.com/api/v1/spaces/cluster*, such as this.

6. Some reasons to exclude are items that have insufficient data to be a valid comparison or have been declined by a user.

7. For a 4:3 image: $4\sqrt{\frac{10000}{12}} \approx 115 \times 3\sqrt{\frac{10000}{12}} \approx 87$

8. This can easily be proven graphically or by contradiction: if the angle $\theta > \frac{\pi}{2}$, at least one component of $\vec{v_i} < 0$ or one component of $\vec{s} < 0$.

9. Results were calculated as of Jun 27, 2015 from KeenIO event data.

10. http://docs.realmassive.apiary.io

11. Benchmarks were performed with the Opticks toolkit [Opt15] on a 614 x 512 pixels x 224 band AVARIS spectral data cube [AVA15], courtesy of NASA/JPL-Caltech. Processing time was no larger than 3 seconds using a memory-mapped file.

REFERENCES

[AVA15] AVARIS Home page. (2015, June 26). Retrieved from http://aviris.jpl.nasa.gov/data/free_data.html

[Bra00] Bradley, P. S., Bennett, K. P., & Demiriz, A. (2000). Constrained k-means clustering. Microsoft Research, Redmond, 1-8.

[DeC00] De Carvalho, O. A., & Meneses, P. R. (2000, February). Spectral correlation mapper (SCM): an improvement on the spectral angle mapper (SAM). In Summaries of the 9th JPL Airborne Earth Science Workshop, JPL Publication 00-18 (Vol. 9). Pasadena, CA: JPL Publication.

[Git15] Github. (2015, June 11). SciPy source code. Retrieved from https://github.com/scipy/scipy/blob/master/scipy/spatial/kdtree.py

[Goo15] Google. (2015, June 11). Google App Engine for Python 1.9.21 Documentation. Retrieved from https://cloud.google.com/appengine/docs/python

[Lan05] Landgrebe, David A (2005). Signal Theory Methods in Multispectral Remote Sensing. Hoboken, NJ: John Wiley & Sons.

[Man01] Maneewongvatana, S., & Mount, D. M. (2001). On the efficiency of nearest neighbor searching with data clustered in lower dimensions (pp. 842-851). Springer Berlin Heidelberg.

[Opt15] Opticks. (2015, June 26). Opticks remote sensing toolkit. Retrieved from https://opticks.org

[Pet11] G. Petropoulos, K. Vadrevu, et. al. *A Comparison of Spectral Angle Mapper and Artificial Neural Network Classifiers Combined with Landsat TM Imagery Analysis for Obtaining Burnt Area Mapping*, Sensors. 10(3):1967-1985. 2011.

[Rea15] RealMassive. (2015, June 10). Retrieved from https://www.realmassive.com

[Ric15] M. Richmond. Licensed under Creative Commons. Retrieved from http://spiff.rit.edu/classes/phys301/lectures/comp/comp.html

[Vat11] A. Vattani. *k-means Requires Exponentially Many Iterations Even in the Plane*, Discrete Comput Geom. 45(4): 596–616. 2011.

[War09] T. Warner, G. Foody, M. Duane Nellis (2009). The SAGE Handbook of Remote Sensing. Thousand Oaks, CA: SAGE Publications Inc.

[Zha09] H. Zhang, Y. Lan, R. Lacey, W. Hoffmann, Y. Huang. *Analysis of vegetation indices derived from aerial multispectral and ground hyperspectral data*, International Journal of Agricultural and Biological Engineering. 2(3): 33. 2009.

The James Webb Space Telescope Data Calibration Pipeline

Howard Bushouse‡*, Michael Droettboom‡, Perry Greenfield‡

https://www.youtube.com/watch?v=o-D4TpRFza4

Abstract—The James Webb Space Telescope (JWST) is the successor to the Hubble Space Telescope (HST) and is currently expected to be launched in late 2018. The Space Telescope Science Institute (STScI) is developing the software systems that will be used to provide routine calibration of the science data received from JWST. The calibration operations use a processing environment provided by a Python module called `stpipe` that provides many common services to each calibration step, relieving step developers from having to implement such functionality. The `stpipe` module provides common configuration handling, parameter validation and persistence, and I/O management.

Individual steps are written as Python classes that can be invoked individually from within Python or from the `stpipe` command line. Any set of step classes can be configured into a pipeline, with `stpipe` handling the flow of data between steps. The `stpipe` environment includes the use of standard data models. The data models, defined using json schema, provide a means of validating the correct format of the data files presented to the pipeline, as well as presenting an abstract interface to isolate the calibration steps from details of how the data are stored on disk.

Index Terms—pipelines, astronomy

Introduction

Data coming from the electronic detectors in scientific instruments attached to telescopes (both on the ground and in space) look nothing like the end product on which astronomers do their analysis or the pictures that show up in the media. Raw images and spectra contain artifacts and extra signals that are intrinsic to the instrumentation itself, rather than the source being observed. These artifacts include things like dead detector pixels, pixel-to-pixel variations in sensitivity, background signal from the detector and instrument, non-linear detector response, anomalous signals due to impacts of cosmic-rays, and spatial distortions due to the optics. All anomalies must be removed or corrected before the data are suitable for scientific analysis. In addition, processing such as combining the data from multiple exposures and extracting one-dimensional spectra from the two-dimensional detector format in which they were recorded must also be performed. This is the job of astronomical data reduction and calibration pipelines.

The Space Telescope Science Institute (STScI), which is the science operations center for the Hubble Space Telescope (HST), has developed and maintained data calibration pipelines for all of the HST scientific instruments and is now in the process of developing the pipelines that will be used for the James Webb Space Telescope (JWST) after it is launched in late 2018. The HST pipelines for the different scientific instruments on the telescope were developed over a span of more than 20 years and hence show an evolution in both software languages and design. The pipelines for each instrument, which now number 11 over the 25 year history of HST, were all written independently of one another and used an assortment of programming languages, including the Subset Preprocessor (SPP) language [Tody83], which is unique to the astronomical community, Fortran, C, and Python. This assortment of languages made maintenance and enhancement rather difficult, and precluded any code sharing between instruments. The HST calibration pipelines also used monolithic, procedural designs, with very little modularity. This approach worked as long as data were allowed to flow uninterrupted from beginning to end, but made it very difficult, if not impossible, to start or stop processing midstream, skip one or more steps, or insert additional steps. Customizing the processing in this way is often necessary for an astronomer to get the most out of their particular observations.

The JWST calibration pipelines are being developed from scratch using a completely new design approach and using almost nothing but Python. There is a common framework for all 4 of the scientific instruments, with extensive sharing of routines and a common code base. The new design allows for flexibility in swapping in and out specific processing steps, easily changing the ordering of steps within pipelines, and the ability for astronomers to plug-in custom processing. This flexibility is necessary due to the fact that the knowledge of the science instruments and the intricacies of the data they produce is constantly evolving, often over the entire lifetime of the mission. The calibration pipelines will be used not only in the production environment at STScI, which will apply an initial round of processing to all data coming from JWST and archiving the results, but will also be distributed to astronmers to run at their home institutions. This gives the users the ability to rerun and refine the processing applied to their observations. The highly modular and flexible nature of the design will allow them to even add in their own custom processing steps, either as part of the pipeline itself or as standalone routines that are run on the data and then reinserted back into the pipeline flow.

Before continuing, a clarification of exactly what we mean by the term "pipeline" is in order. A high-level workflow management system is used to guide the entire flow of data processing. This end-to-end process includes the receipt of telemetry downlinks from the telescope, reformatting the raw telemetry packets into

* *Corresponding author: bushouse@stsci.edu*
‡ *Space Telescope Science Institute*

useful data file formats, integrating meta data from various database systems, reducing and calibrating the raw data read out from the detectors in order to remove instrumental artifacts, storing the fully reduced data into an archive, and automaticlly notifying the astronomers who obtained the observations that the data are available. The calibration pipelines reported on here concern only the middle step of reducing and calibrating the raw images and spectra so that they are ready for scientific analysis. As such, the calibration pipelines do *not* provide any kind of high-level process management functions, interfaces to databases, and so on. The calibration pipelines are strictly devoted to applying a series of operations to the pixel values that comprise an image in order to remove instrumental artifacts and place the data values onto scales involving physical units. The particular series of such steps varies according to the observation modes used by the different instruments on the telescope. The calibration pipelines define and control the data flow within these different series of processing steps. The calibration pipelines, therefore, don't require a large, high-level task scheduling and workflow management system (e.g. Luigi [BF12]). A separate high-level process management system is used to control the execution of all the pieces involved in the end-to-end system described above, of which the calibration pipelines are one small part.

A primary goal for the JWST calibration pipelines is to have the system distributable to astronomers to execute on their own systems at their home institutions. It's often necessary for an astronomer to tailor or modify the details of the processing that's applied to their particular observations in order to get the greatest scientific return. The calibration pipeline package has therefore been designed to be as light-weight and self-contained as possible in order to make it as easy as possible for users to install and run themselves. The only external interface required is to our Calibration Reference Data System (CRDS), which is used to supply reference data needed by some of the calibration steps. The CRDS server at STScI will accept requests for reference files from the client on an astronomer's home system and automatically download the requested files to their systems for use locally.

stpipe

The heart - or perhaps more appropriately, the nervous system - of the JWST calibration pipeline environment is a Python module called `stpipe`. `stpipe` manages individual processing steps that can be combined into pipelines. The `stpipe` environment provides functionality that is common to all steps and pipelines so that they behave in a consistent manner. It provides:

- running steps and pipelines from the command line
- parsing of configuration settings
- composing steps into pipelines
- file management and data I/O between pipeline steps
- interface to the Calibration Reference Data System (CRDS)
- logging

Each pipeline step is embodied as a Python class, with a pipeline being composed of multiple steps. Pipelines can in turn be strung together, just like steps, to compose an even higher-order flow. Steps and pipelines can be executed from the command-line using `stpipe`, which is the normal mode of operations in the production environment that processes data in real-time as it is downlinked from the telescope. The step and pipeline classes can also be instantiated and executed from within a Python shell, which provides a lot of flexibility for developers when testing the code and to astronomers who may need to occasionally tweak or otherwise customize the processing of their particular data sets.

When run from the command line, `stpipe` handles the parsing of configuration parameters that can be provided either as arguments on the command line or within configuration files. Configuration files use the well-known ini-file format and `stpipe` uses the ConfigObj library to parse them. `stpipe` handles all of the file I/O for each step and the passing of data between pipeline steps, as well as providing access within each step to a common logging facility. It also provides a common interface for all steps to reference data files that are stored in the STScI Calibration Reference Data System (CRDS). Having all of these functions handled by the `stpipe` environment relieves developers from having to include these features in each step or pipeline and provides a consistent interface to users as well.

Command-line Execution

`stpipe` can be used from the command line to execute a step or pipeline by providing either the class name of the desired step/pipeline or a configuration file that references the step/pipeline class and provides optional argument values. An example that directly calls a class is:

```
> strun jwst_pipeline.SloperPipeline input.fits
  --output_file="myimage.fits"
```

The same thing can be accomplished by specifying a config file, e.g.:

```
> strun sloper.cfg input.fits
```

where sloper.cfg contains:

```
name = "SloperPipeline"
class = "jwst_pipeline.SloperPipeline"
output_file = "myimage.fits"
save_calibrated_ramp = True
```

Note that in the absence of the user explicitly specifying an output file name for saving the results, `stpipe` includes a mechanism for constructing an output file name that is composed of the input root file name and the name of the pipeline or step class that has been applied to produce the output.

Python Execution

Steps and pipelines can also be called from within Python using the class "call" method:

```
>>> from jwst_pipeline import SloperPipeline
>>> SloperPipeline.call('input.fits',
                        config_file='sloper.cfg')
```

Logging

The `stpipe` logging mechanism is based on the standard Python logging framework. The framework has certain built-in things that it automatically logs, such as the step and pipeline start/stop times, as well as platform information. Steps can log their own specific items and every log entry is time-stamped. Every log message that's posted has an associated level of severity, including DEBUG, INFO, WARN, ERROR, and CRITICAL (the same levels provided in the Python stdlib). The user can control how verbose the logging is via arguments in the config file or on the command line.

Steps and Pipelines

Steps define the parameters that are available, their data types (specified in "configspec" format), and their default values. As mentioned earlier, users can override the default parameter values by supplying values in configuration files or on the command-line. Steps can be combined into pipelines, and pipelines are themselves steps, allowing for arbitrary levels of nesting.

Simple linear pipelines can be constructed as a straight sequence of steps, where the output of each step feeds into the input of the next. These linear pipelines can be started and stopped at arbitrary points, via arguments supplied by the user, with all of the status saved to disk and then resumed later if desired. More complex (non-linear) pipelines can be defined using a Python function, so that the flow between steps is completely flexible. This is useful, for example, when the output of a step is multiple products that need to be looped over by subsequent steps. Because of their non-linear nature, these more complex types of pipeline can not be started or stopped mid-stream. Both types of pipelines, however, allow the user to skip certain steps by supplying configuration overrides.

Step configuration files can also specify pre- and post-hooks, to introduce custom processing into the pipeline. The hooks can be Python functions or shell commands. This allows astronomers to examine or modify data, or insert a custom correction, at any point along the pipeline without needing to write their own Python code.

A hypothetical pipeline is shown below. In this example, the input data is modified in-place by each processing step and the results passed along from one step to the next. The final result is saved to disk by the `stpipe` environment. Each pipeline subclass inherits from the *Pipeline* class. The subclass defines the Steps that will be used so that the framework can configure parameters for the individual Steps. This is done with the `step_defs` member, which is a dictionary that maps step names to step classes. This dictionary defines what the Steps are, but says nothing about their order or how data flows from one Step to the next. That is defined in Python code in the Pipeline's `process` method. By the time the Pipeline's `process` method is called, the Steps in `step_defs` will be instantiated as member variables.

```
from jwst_lib.stpipe import Pipeline

# pipeline step imports
from jwst_pipeline.dq import dq_step
from jwst_pipeline.ipc import ipc_step
from jwst_pipeline.bias import bias_step
from jwst_pipeline.reset import reset_step
from jwst_pipeline.frame import frame_step
from jwst_pipeline.jump import jump_step
from jwst_pipeline.ramp import ramp_step

# setup logging
import logging
log = logging.getLogger()
log.setLevel(logging.DEBUG)

# the pipeline class
class SloperPipeline(Pipeline)

    spec = """
        save_cal = boolean(default=False)
    """

    # step definitions
    step_defs = {"dq" : dq_step.DQInitStep,
                 "ipc" : ipc_step.IPCStep,
                 "bias" : bias_step.SuperBiasStep,
                 "reset" : reset_step.ResetStep,
                 "frame" : frame_step.LastFrameStep,
                 "jump" : jump_step.JumpStep,
                 "ramp_fit" : ramp_step.RampFitStep,
                }

    # the pipeline process
    def process(self, input):
        log.info("Starting calwebb_sloper ...")

        input = self.dq(input)
        input = self.ipc(input)

        # don't apply superbias to MIRI data
        if input.meta.instrument.name != "MIRI":
            input = self.bias(input)

        # only apply reset and lastframe to MIRI data
        if input.meta.instrument.name == "MIRI":
            input = self.reset(input)
            input = self.frame(input)

        input = self.jump(input)

        # save the results so far
        if save_cal:
            input.save(product_name(self, "cal"))

        input = self.ramp_fit(input)

        log.info("... ending calwebb_sloper")
        return input
```

Another example listed below shows how a pipeline can be included within a pipeline, just like a step, using all the same means to declare the pipeline and receiving all the same configuration handling from `stpipe`. In this example an existing pipeline is first applied to the input, followed by two more individual steps.

```
from jwst_lib.stpipe import Pipeline

# pipeline and step imports
from jwst_pipeline.pipeline import sloper_pipe
from jwst_pipeline.wcs import wcs_step
from jwst_pipeline.flat import flat_step

# setup logging
import logging
log = logging.getLogger()
log.setLevel(logging.DEBUG)

# the pipeline class
class MyPipeline(Pipeline)

    # step definitions
    step_defs = {"sloper": sloper_pipe.SloperPipe,
                 "wcs" : wcs_step.WcsStep,
                 "flat" : flat_step.FlatStep,
                }

    # the pipeline process
    def process(self, input):

        slope_model = self.sloper(input)
        slope_model = self.wcs(slope_model)
        result = self.flat(slope_model)

        return result
```

Data Models

For nearly 35 years most astronomers, observatories, and astronomical data processing packages have used a common data file format known as the Flexible Image Transport System (FITS).

While a common file format has made it very easy to share data across groups of people and software, the format is used in many different ways to store the unique aspects of different types of observational data (e.g. images versus spectra). The burden of loading, parsing, and interpreting the contents of any particular FITS file has always fallen to the processing code that's trying to do something to the data. For the JWST calibration pipelines, the `stpipe` environment takes care of all the file I/O, leaving the developers of steps and pipelines to concentrate on processing the data itself.

This has been implemented through the use of software data models in `stpipe`, through which it performs all the necessary I/O between files on disk and the data models. The data models allow the on-disk representation of the data to be abstracted from the pipeline steps via the I/O mechanisms built into `stpipe`. The use of software data models in the processing steps also has the benefit of eliminating or at least being able to manage dependencies between the various steps. Because all of the actual science data and its associated meta data are completely self-contained within a model, each step has all of the information it needs to do its work. For example, if one of the final steps in a particular pipeline gets modified in some way, there's no need to restart the processing for a particular data set from the beginning. The results from the step immediately preceding the change can be reloaded and the modified step executed from that point. If a particular processing step changes the overall format or content of the data set in some way, the result is saved in a different type of data model. Each step can perform a check to ensure that the input it's been given conforms to the type of data model expected in that step. Any inconsistencies will be detected immediately and the process will shutdown with a warning to the user, rather than the undesirable behavior of having a step crash because the input data were not compatible with that step.

The `stpipe` models interface currently reads and writes FITS files, but will soon also support the Advanced Scientific Data Format (ASDF) file format being developed by STScI [DB15]. The interface provides the same methods of access within the pipeline steps whether the data is on disk or already in memory. Futhermore, the `stpipe` interface can decide the best way to manage memory, rather than leaving it up to the code in individual steps. The use of the data models isolates the processing code from future changes in file formats or keywords.

Each model is a bundle of array or tabular data, along with metadata. The structure of the data and metadata for any model is defined using JSON Schema [Dro14]. JSON Schema works with any structured data, such as YAML and XML. The data model schemas are modular, such that a core schema that contains elements common to all models can also include any number of additional sub-schema that are unique to one or more particular models.

An example is the simple "ImageModel", shown below, which contains a total of three 2-dimensional data arrays. The schema defines the name of each model attribute, its data type, array dimensions (in the case of data arrays), and default values. Attributes can also be designated as required or optional. The "core.schema.json" and "sens.schema.json" files contain additional definitions of metadata attributes.

```
{ "allOf": [
    {"$ref": "core.schema.json"},
    {"type" : "object",
        "properties" : {
            "data" :
            {"type" : "data",
             "title" : "The science data",
             "fits_hdu" : "SCI",
             "default" : 0.0,
             "ndim" : 2,
             "dtype" : "float32"
            },

            "dq" :
            {"type" : "data",
             "title" : "Data quality array",
             "fits_hdu" : "DQ",
             "default" : 0,
             "dtype" : "uint32"
            },

            "err" :
            {"type" : "data",
             "title" : "Error array",
             "fits_hdu" : "ERR",
             "default" : 0.0,
             "dtype" : "float32"
            },
            "sens" : {"$ref": "sens.schema.json"}
        }
    }
  ]
}
```

Within the pipeline or step code the developer loads a data model using simple statements like:

```
from jwst_lib.stpipe import Step, cmdline
from jwst_lib import models

class FlatFieldStep(Step):

    def process(self, input):

        with models.ImageModel(input) as im:
            result = flat_field.correct(im)

        return result
```

In a case like this, `stpipe` takes care of determining whether "input" is a model already loaded into memory or a file on disk. If the latter, it opens and loads the file contents into an ImageModel. The step code then has direct access to all the attributes of the ImageModel, such as the data, dq, and err arrays defined in the ImageModel schema above. If this is the only step being executed, `stpipe` will save the returned data model to disk. If this step is part of a pipeline, on the other hand, `stpipe` will pass the returned data model in memory to the next step. At the end of the pipeline the final model will be saved to disk.

Conclusions

We are in the process of building the data calibration pipelines that will be used to remove instrumental artificats from images and spectra obtained by the James Webb Space Telescope. The calibration pipelines rely on the `stpipe` environment developed at STScI, which handles all data I/O and configuration hanlding for the individual calibration steps. The entire package is designed to be relatively light-weight and self-contained so that it can be easily distributed to and run by individual astronomers at their home institutions. Calibration steps and pipelines can be executed from the command line, or their classes can be instantiated and called from with an interactive Python environment. This latter feature in particular allows for great flexibility to tweak or enhance the processing that's applied to a given data set. A user can, for

example, invoke a standard pipeline or a set of individual steps from within Python and at any point during the processing apply their own custom processing to the resulting data model in an interactive way. The ability to interact in real time with the data as it proceeds through the processing is new to the JWST calibration environment and did not exist at all for users of Hubble Space Telescope data.

REFERENCES

[BF12] E. Bernhardsson and E. Freider. *The Luigi Python module*, https://github.com/spotify/luigi

[Dro14] M. Droettboom. *JSON Schema*, http://json-schema.org

[DB15] M. Droettboom and E. Bray. *The ASDF Standard*, http://asdf-standard.readthedocs.org/en/latest/

[Tody83] D. Tody. *A Reference Manual for the IRAF Subset Preprocessor Language*, 1983

Circumventing The Linker: Using SciPy's BLAS and LAPACK Within Cython

Ian Henriksen‡*

https://www.youtube.com/watch?v=R4yB-8tB0J0

Abstract—BLAS, LAPACK, and other libraries like them have formed the underpinnings of much of the scientific stack in Python. Until now, the standard practice in many packages for using BLAS and LAPACK has been to link each Python extension directly against the libraries needed. Each module that calls these low-level libraries directly has had to link against them independently. The task of finding and linking properly against the correct libraries has, in the past, been a substantial obstacle in the development and distribution of Python extension modules.

Cython has existing machinery that allows C-level declarations to be shared between Cython-compiled extension modules without linking against the original libraries. The Cython BLAS and LAPACK API in SciPy uses this functionality to make it so that the same BLAS and LAPACK libraries that were used to compile SciPy can be used in Python extension modules via Cython. This paper will demonstrate how to create and use these APIs for both Fortran and C libraries in a platform-independent manner.

Index Terms—Cython, BLAS, LAPACK, SciPy

Introduction

Many of the primary underpinnings of the scientific Python stack rely on interfacing with lower-level languages, rather than working with code that is exclusively written in Python. SciPy [SciPy], for example, is a collection of algorithms and libraries implemented in a variety of languages that are wrapped to provide convenient and usable APIs within Python. Because programmers often need to call low-level libraries, F2PY [F2PY], Cython [Cython], and a variety of similar tools have been introduced to simplify that process.

In spite of the large number of tools for automatically wrapping low-level libraries, interfacing with low-level languages can still present a significant challenge. If performance bottlenecks depend on any third party algorithms, developers are faced with the daunting task of rewriting their algorithms to interface with completely different packages and adding large dependencies on existing low-level libraries. Adding these dependencies to an existing project can complicate the build process and expose the project to a much wider variety of bugs. When developers distribute code meant to work reliably with a variety of compilers in a variety of environments, low-level dependencies become a never-ending source of trouble. The problems caused by these dependencies are further complicated by the fact that, currently, each Python module must shoulder the burden of distributing or finding the libraries it uses.

For example, consider the case of a simple tridiagonal matrix solve. This sort of solve can be done easily within Python.

```
import numpy as np
def pytridiag(a, b, c, x):
    """ Solve the system A y = x for y
    where A is the square matrix with subdiagonal
    'a', diagonal 'b', and superdiagonal 'c'. """
    A = np.zeros((b.shape[0], b.shape[0]))
    np.fill_diagonal(A[1:], a)
    np.fill_diagonal(A, b)
    np.fill_diagonal(A[:,1:], c)
    return np.linalg.solve(A, x)
```

This function works fine for small problems, but, if it needs to be called frequently, a more specialized algorithm could provide major improvements in both speed and accuracy. An ideal candidate for this sort of optimization is LAPACK's [LAPACK] routine `dgtsv`. That routine can be used within Cython to solve the same problem more quickly and with fewer numerical errors.

```
# cython: wraparound = False
# cython: boundscheck = False

cdef extern from "lapacke.h" nogil:
    void dgtsv "LAPACK_dgtsv"(int *n, int *nrhs,
                              double *dl, double *d,
                              double *du, double *b,
                              int *ldb, int *info)

cpdef tridiag(double[::1] a, double[::1] b,
              double[::1] c, double[::1] x):
    cdef int n=b.shape[0], nrhs=1, info
    # Solution is written over the values in x.
    dgtsv(&n, &nrhs, &a[0], &b[0], &c[0], &x[0],
          &n, &info)
```

Though this process for calling an external function from a library is not particularly difficult, the setup file for the Python module now must find a proper LAPACK installation. If there are several different versions of LAPACK present, a suitable one must be chosen. The proper headers and libraries must be found, and, if at all possible, binary incompatibilities between compilers must be avoided. If the desired routine is not a part of one of the existing C interfaces, then it must be called via the Fortran ABI and the name mangling schemes used by different Fortran compilers must be taken into account. All of the code needed to do this must also be maintained so that it continues to work with new versions of the

* Corresponding author: iandh@byu.edu
‡ Brigham Young University Math Department

different operating systems, compilers, and BLAS and LAPACK libraries.

An effective solution to this unusually painful problem is to have existing Python modules provide access to the low-level libraries that they use. NumPy has provided some of this sort of functionality for BLAS and LAPACK by making it so that the locations of the system's BLAS and LAPACK libraries can be found using NumPy's distutils module. Unfortunately, the existing functionality is only usable at build time, and does little to help users that do not compile NumPy and SciPy from source. It also does not include the various patches used by SciPy to account for bugs in different BLAS and LAPACK versions and incompatibilities between compilers.

Cython has provided similar functionality that allows C-level APIs to be exported between Cython modules without linking. In the past, these importing systems have been used primarily to share Cython-defined variables, functions and classes between Cython modules. If used carefully, however, the existing machinery in Cython can be used to expose functions and variables from existing libraries to other extension modules. This makes it so that other Python extension modules can use the functions it wraps without having to build, find, or link against the original library.

The Cython API for BLAS and LAPACK

Over the last year, a significant amount of work has been devoted to exposing the BLAS and LAPACK libraries within SciPy at the Cython level. The primary goals of providing such an interface are twofold: first, making the low-level routines in BLAS and LAPACK more readily available to users, and, second, reducing the dependency burden on third party packages.

Using the new Cython API, users can now dynamically load the BLAS and LAPACK libraries used to compile SciPy without having to actually link against the original BLAS and LAPACK libraries or include the corresponding headers. Modules that use the new API also no longer need to worry about which BLAS or LAPACK library is used. If the correct versions of BLAS and LAPACK were used to compile SciPy, the correct versions will be used by the extension module. Furthermore, since Cython uses Python capsule objects internally, C and C++ modules can easily access the needed function pointers.

BLAS and LAPACK proved to be particularly good candidates for a Cython API, resulting in several additional benefits:

- Python modules that use the Cython BLAS/LAPACK API no longer need to link statically to provide binary installers.
- The custom ABI wrappers and patches used in SciPy to provide a more stable and uniform interface across different BLAS/LAPACK libraries and Fortran compilers are no longer needed for third party extensions.
- The naming schemes used within BLAS and LAPACK make it easy to write type-dispatching versions of BLAS and LAPACK routines using Cython's fused types.

In providing these low-level wrappers, it was simplest to follow the calling conventions of BLAS and LAPACK as closely as possible, so all arguments are passed as pointers. Using the new Cython wrappers, the tridiagonal solve example shown above can be implemented in Cython in nearly the same way as before, except that all the needed library dependencies have already been resolved within SciPy.

```
# cython: wraparound = False
# cython: boundscheck = False

from scipy.linalg.cython_lapack cimport dgtsv

cpdef tridiag(double[::1] a, double[::1] b,
              double[::1] c, double[::1] x):
    cdef int n=b.shape[0], nrhs=1, info
    # Solution is written over the values in x.
    dgtsv(&n, &nrhs, &a[0], &b[0], &c[0], &x[0],
          &n, &info)
```

Since Cython uses Python's capsule objects internally for the cimport mechanism, it is also possible to extract function pointers directly from the module's `__pyx_capi__` dictionary and cast them to the needed type without writing the extra shim.

Exporting Cython APIs for Existing C Libraries

The process of exposing a Cython binding for a function or variable in an existing library is relatively simple. First, as an example, consider the following C file and the corresponding header.

```
// myfunc.c
double f(double x, double y){
    return x * x - x * y + 3 * y;
}

// myfunc.h
double f(double x, double y);
```

This library can be compiled by running `clang -c myfunc.c -o myfunc.o`.

This can be exposed at the Cython level and exported as a part of the resulting Python module by including the header in the pyx file, using the function from the C file to create a Cython shim with the proper signature, and then declaring the function in the corresponding pxd file without including the header file. A similar approach using function pointers is also possible. Here's a minimal example that demonstrates this process:

```
# cy_myfunc.pyx
# Use a file-level directive to link
# against the compiled object.
# distutils: extra_link_args = ['myfunc.o']
cdef extern from 'myfunc.h':
    double f(double x, double y) nogil
# Declare both the external function and
# the Cython function as nogil so they can be
# used without any Python operations
# (other than loading the module).
cdef double cy_f(double x, double y) nogil:
    return f(x, y)

# cy_myfunc.pxd
# Don't include the header here.
# Only give the signature for the
# Cython-exposed version of the function.
cdef double cy_f(double x, double y) nogil

# cy_myfunc_setup.py
from distutils.core import setup
from Cython.Build import cythonize
setup(ext_modules=cythonize('cy_myfunc.pyx'))
```

From here, once the module is built, the Cython wrapper for the C-level function can be used in other modules without linking against the original library.

Exporting a Cython API for an existing Fortran library

When working with a Fortran library, the name mangling scheme used by the compiler must be taken into account. The simplest

way to work around this would be to use Fortran 2003's ISO C binding module. Since, for the sake of platform/compiler independence, such a recent version of Fortran cannot be used in SciPy, an existing header with a small macro was used to imitate the name mangling scheme used by the various Fortran compilers. In addition, for this approach to work properly, all the Fortran functions in BLAS and LAPACK were first wrapped as subroutines (functions without return values) at the Fortran level.

```
!     myffunc.f
!     The function to be exported.
      double precision function f(x, y)
        double precision x, y
        f = x * x - x * y + 3 * y
      end function f

!     myffuncwrap.f
!     A subroutine wrapper for the function.
      subroutine fwrp(out, x, y)
        external f
        double precision f
        double precision out, x, y
        out = f(x, y)
      end
```

```
// fortran_defs.h
// Define a macro to handle different
// Fortran naming conventions.
// Copied verbatim from SciPy.
#if defined(NO_APPEND_FORTRAN)
#if defined(UPPERCASE_FORTRAN)
#define F_FUNC(f,F) F
#else
#define F_FUNC(f,F) f
#endif
#else
#if defined(UPPERCASE_FORTRAN)
#define F_FUNC(f,F) F##_
#else
#define F_FUNC(f,F) f##_
#endif
#endif
```

```
// myffuncwrap.h
#include "fortran_defs.h"
void F_FUNC(fwrp, FWRP)(double *out, double *x,
                        double *y);
```

```
# cyffunc.pyx
cdef extern from 'myffuncwrap.h':
    void fort_f "F_FUNC(fwrp, FWRP)"(double *out,
                                     double *x,
                                     double *y) nogil

cdef double f(double *x, double *y) nogil:
    cdef double out
    fort_f(&out, x, y)
    return out
```

```
# cyffunc.pxd
cdef double f(double *x, double *y) nogil
```

Numpy's distutils package can be used to build the Fortran libraries and compile the final extension module. The interoperability between NumPy's distutils package and Cython is limited, but the C file resulting from the Cython compilation can still be used to create the final extension module.

```
# cyffunc_setup.py
from numpy.distutils.core import setup
from numpy.distutils.misc_util import Configuration
from Cython.Build import cythonize
def configuration():
    config = Configuration()
    config.add_library('myffunc',
                       sources=['myffunc.f',
                                'myffuncwrap.f'])
    config.add_extension('cyffunc',
                         sources=['cyffunc.c'],
                         libraries=['myffunc'])
    return config
# Run Cython to get the needed C files.
# Doing this separately from the setup process
# causes any Cython file-specific distutils
# directives to be ignored.
cythonize('cyffunc.pyx')
setup(configuration=configuration)
```

There are many routines in BLAS and LAPACK, and creating these wrappers currently still requires a large amount of boilerplate code. When creating these wrappers, it was easiest to write Python scripts that used F2PY's existing functionality for parsing Fortran files to generate a set of function signatures that could, in turn, be used to generate the needed code.

Since SciPy supports several versions of LAPACK, it was also necessary to determine which routines should be included as a part of the new Cython API. In order to support all currently used versions of LAPACK, we limited the functions in the Cython API to include only those that had a uniform interface from version 3.1 through version 3.5.

Conclusion

The new Cython API for BLAS and LAPACK in SciPy helps to alleviate the substantial packaging burden imposed on Python packages that use BLAS and LAPACK. It provides a model for including access to lower-level libraries used within a Python package. It makes BLAS and LAPACK much easier to use for new and expert users alike and makes it much easier for smaller modules to write platform and compiler independent code. It also provides a model that can be extended to other packages to help fight dependency creep and reduce the burden of package maintenance. Though it is certainly not trivial, it is still fairly easy to add new Cython bindings to an existing library. Doing so makes the lower-level libraries vastly easier to use.

Going forward, there is a great need for similar APIs for a wider variety of libraries. Possible future directions for the work within SciPy include using Cython's fused types to expose a more type-generic interface to BLAS and LAPACK, writing better automated tools for generating wrappers that expose C, C++, and Fortran functions automatically, and making similar interfaces available in ctypes and CFFI.

References

[SciPy] 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), DOI:10.1109/MCSE.2011.37

[Cython] Stefan Behnel, Robert Bradshaw, Craig Citro, Lisandro Dalcin, Dag Sverre Seljebotn and Kurt Smith. Cython: The Best of Both Worlds, Computing in Science and Engineering, 13, 31-39 (2011), DOI:10.1109/MCSE.2010.118

[F2PY] Pearu Peterson. F2PY: a tool for connecting Fortran and Python programs, International Journal of Computational Science and Engineering, 4 (4), 296-305 (2009), DOI:10.1504/IJCSE.2009.029165

[LAPACK] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, D. Sorensen. LAPACK Users' Guide Third Edition, Society for Industrial and Applied Mathematics, 1999.

Mesa: An Agent-Based Modeling Framework

David Masad[‡*], Jacqueline Kazil[‡]

https://www.youtube.com/watch?v=lcySLoprPMc

Abstract—Agent-based modeling is a computational methodology used in social science, biology, and other fields, which involves simulating the behavior and interaction of many autonomous entities, or agents, over time. There is currently a hole in this area in Python's robust and growing scientific ecosystem. Mesa is a new open-source, Apache 2.0 licensed package meant to fill that gap. It allows users to quickly create agent-based models using built-in core components (such as agent schedulers and spatial grids) or customized implementations; visualize them using a browser-based interface; and analyze their results using Python's data analysis tools. Its goal is to be a Python 3-based alternative to other popular frameworks based in other languages such as NetLogo, Repast, or MASON. Since the framework is being built from scratch it is able to incorporate lessons from other tools. In this paper, we present Mesa's core features and demonstrate them with a simple example model.[1]

Index Terms—agent-based modeling, multi-agent systems, cellular automata, complexity, modeling, simulation

Introduction

Agent-based modeling involves simulating the behavior and interaction of many autonomous entities, or agents, over time. Agents are objects that have rules and states, and act accordingly with each step of the simulation [Axtell2000]. These agents may represent individual organisms, humans, entire organizations, or abstract entities. Robert Axtell, one of the early scholars of agent-based models (ABMs), identified the following advantages [Axtell2000]:

1) Unlike other modeling approaches, ABMs capture the path as well as the solution, so one can analyze the system's dynamic history.
2) Most social processes involve spatial or network attributes, which ABMs can incorporate explicitly.
3) When a model (A) produces a result (R), one has established a sufficiency theorem, meaning *R* if *A*.

To understand the utility of agent-based modeling, consider one of the earliest and best-known models, created by Thomas Schelling. Schelling wanted to test the theory that segregated neighborhoods can arise not just by active racism, but due to only a mild preference for neighbors of the same ethnicity [Schelling1971]. The model consists of majority-group and minority-group agents living on a grid, who have a preference for only several neighbors of the same group. When that preference is not met, they move to a different grid cell. The model demonstrates that even a mild preference for same-group neighbors leads to a dramatic degree of segregation. This is an example of the *emergence* of a higher-order phenomena from the interactions of lower-level entities, and demonstrates the link between agent-based modeling and complexity theory, and complex adaptive systems in particular [Miller2009].

There are currently several tools and frameworks in wide use for agent-based modeling[2], particularly NetLogo [Wilensky1999], Repast [North2013], and MASON [Luke2005]. From our perspective, all of these share a key weakness: they do not use Python. This is not just a matter of parochial preference. In recent years, Python has become an increasingly popular language for scientific computing [Perez2011], supported by a mature and growing ecosystem of tools for analysis and modeling. Python is widely considered a more natural, easy-to-use language than Java, which is used for Repast and MASON; and unlike NetLogo's custom scripting language, Python is a general purpose programming language. Furthermore, unlike the other frameworks, Python allows interactive analysis of model output data, through the IPython Notebook [Perez2007] or similar tools. Despite these advantages, and despite several partial efforts (e.g. [Zvoleff2013], [Sayama2013]), a Python agent-based modeling framework does not yet exist. Mesa is intended to fill this gap.

Mesa is a new open-source, Apache 2.0 licensed Python package that allows users to quickly create agent-based models using built-in core components (such as agent schedulers and spatial grids) or customized implementations; visualize them using a browser-based interface; and analyze their results using Python's data analysis tools.

Designing a new framework from the ground up also allows us to implement features not found in existing frameworks. For example, as we explain in more detail below, other ABM frameworks tend to use a single agent activation regime by default; in Mesa, we implement several agent schedulers and require the modeler to specify which one is being used. We also implement several useful tools to accelerate common model analysis tasks: a data collector (present only in Repast) and a batch runner (available in Repast and NetLogo only via menu-driven systems), both of which can export their results directly to *pandas* [McKinney2011] data frame format for immediate analysis.

* *Corresponding author: david.masad@gmail.com, jackiekazil@gmail.com*
‡ *Department of Computational Social Science, George Mason University*

1. Mesa is available on GitHub at https://github.com/projectmesa/mesa

2. Throughout this paper, and in Mesa's documentation more broadly, we use the term 'agent-based model' to encompass a wide range of related computational models as well, such as multi-agent systems, cellular automata and individual-based model.

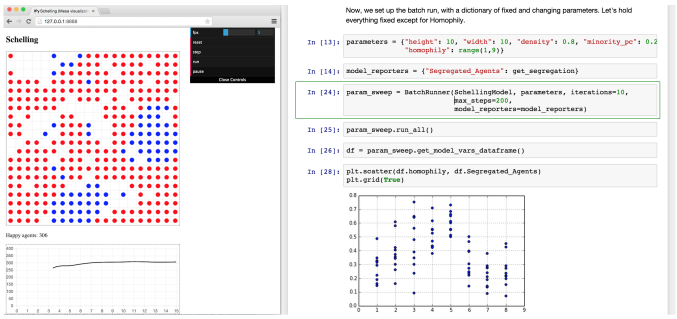


Fig. 1: A Mesa implementation of the Schelling segregation model, being visualized in a browser window and analyzed in an IPython notebook.

While interactive data analysis is important, direct visualization of every model step is also a key part of agent-based modeling, both for debugging, and for developing an intuition of the dynamics that emerge from the model. Mesa facilitates such live visualization as well. It avoids issues of system-specific GUI dependencies by using the browser as a front-end, giving framework and model developers access to the full range of modern JavaScript data visualization tools.

In the remainder of this paper, we will present Mesa's architecture and core features. To illustrate their use, we will describe and build a simple agent-based model, drawn from econophysics and presenting a statistical mechanics approach to wealth distribution [Dragulescu2002]. The core of the model is as follows: *there are some number of agents, all of whom begin with 1 unit of money. At every step of the model, an agent gives 1 unit of money (if they have it) to some other agent.* Despite its simplicity, this model yields results that are often unexpected to those not familiar with it. For our purposes, it also easily demonstrates Mesa's core features.

Architecture

Overview

The guiding principle of Mesa's architecture is modularity. Mesa makes minimal assumptions about the form a model will take. For example, while many models have spatial components, many others do not, while some may involve multiple separate spaces. Similarly, visualizations which display each step of a model may be a critical component of some models and completely unnecessary for others. Thus Mesa aims to offer a set of components that can be easily combined and extended to build different kinds of models.

We divide the modules into three overall categories: modeling, analysis and visualization. The modeling components are the core of what's needed to build a model: a **Model** class to store model-level parameters and serve as a container for the rest of the components; one or more **Agent** classes which describe the model agents; most likely a **scheduler** which controls the agent activation regime, and handles time in the model in general, and components describing the **space** and/or **network** the agents are situated in. The analysis components are the **data collectors** used to record data from each model run, and **batch runners** for automating multiple runs and parameter sweeps. Finally, the visualization components are used to map from a model object to one or more visual representations via a server interface to a browser window. Figure 2 shows a simple UML diagram of a typical Mesa model.

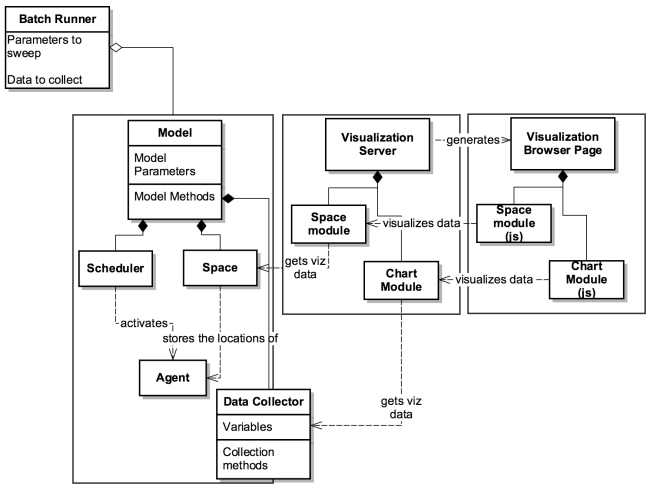


Fig. 2: Simplified UML diagram of Mesa architecture.

To begin building the example model described above, we first create two classes: one for the model object itself, and one the model agents. The model's one parameter is the number of agents, and each agent has a single variable: how much money it currently has. Each agent also has only a single action: give a unit of money to another agent. (The numbers in comments of the code below correspond to notes under the code block).

```
from mesa import Model, Agent

class MoneyAgent(Agent):
  """ An agent with fixed initial wealth."""
  def __init__(self, unique_id):
    self.unique_id = unique_id                    # 1.
    self.wealth = 1

class MoneyModel(Model):
  """A model with some number of agents."""
  def __init__(self, N):
    self.num_agents = N
    # The scheduler will be added here
    self.create_agents()

  def create_agents(self):
    """Method to create all the agents."""
    for i in range(self.num_agents):
      a = MoneyAgent(i)
      # Now what? See below.
```

1) Each agent should have a unique identifier, stored in the `unique_id` field.

Scheduler

The scheduler is a model component which deserves special attention. Unlike systems dynamics models, and dynamical systems more generally, time in agent-based models is almost never continuous; ABMs are, at bottom, discrete-event simulations. Thus, scheduling the agents' activation is particularly important, and the activation regime can have a substantial effect on the behavior of a simulation [Comer2014]. Many ABM frameworks do not make this easy to change. For example, NetLogo defaults to a random activation system, while MASON's scheduler is uniform by default. By separating out the scheduler into a separate, extensible class, Mesa both requires modelers to specify their choice of activation regime, and makes it easy to change and

observe the results. Additionally, the scheduler object serves as the model's storage structure for active agents.

Many models distinguish between a step (sometimes called a tick) of the model, and an activation of a single agent. A step of the model generally involves the activation of one or more agents, and frequently of all of the agents. There are numerous possible scheduling regimes used in agent-based modeling, including:

- Synchronous or simultaneous activation, where all agents act simultaneously. In practice, this is generally implemented by recording each agent's decision one at a time, but not altering the state of the model until all agents have decided.
- Uniform activation, where all agents are activated in the same order each step of the model.
- Random activation, where each agent is activated each step of the model, but the order in which they are activated is randomized for each step.
- Random interval activation, where the interval between each activation is drawn from a random distribution (most often Poisson). In this regime, there is no set model step; instead, the model maintains an internal 'clock' and schedule which determines which agent will be activated at which time on the internal clock.
- More exotic activation regimes may be used as well, such as agents needing to spend resources to activate more frequently.

All scheduler classes share a few standard method conventions, in order to make them both simple to use and seamlessly interchangeable. Schedulers are instantiated with the model object they belong to. Agents are added to the schedule using the `add` method, and removed using `remove`. Agents can be added at the very beginning of a simulation, or any time during its run -- e.g. as they are born from other agents' reproduction.

The `step` method runs one step of the *model*, activating agents accordingly. It is here that the schedulers primarily differ from one another. For example, the uniform `BaseScheduler` simply loops through the agents in the order they were added, while `RandomActivation` shuffles their order prior to looping.

Each agent is assumed to have a `step` method of its own, which receives the model state as its sole argument. This is the method that the scheduler calls in order to activate each agent.

The scheduler maintains two variables determining the model clock. `steps` counts how many steps of the model have occurred, while `time` tracks the model's simulated clock time. Many models will only utilize `steps`, but a model using Poisson activation, for example, will track both separately, with steps counting individual agent activations and `time` the scheduled model time of the most recent activation. Some models may implement particular schedules simulating real time: for example, `time` may attempt to simulate real-world time, where agent activations simulate them as they engage in different activities of different durations based on the time of day.

Now, let's implement a schedule in our example model. We add a `RandomActivation` scheduler to the model, and add each created agent to it. We also need to implement the agents' `step` method, which the scheduler calls by default. With these additions, the new code looks like this:

```
from mesa.time import RandomActivation

class MoneyAgent(Agent):
  # ...

  def step(self, model):
    """Give money to another agent."""
    if self.wealth > 0:
      # Pick a random agent
      other = random.choice(model.schedule.agents)
      # Give them 1 unit money
      other.wealth += 1
      self.wealth -= 1

class MoneyModel(Model):

  def __init__(self, N):
    self.num_agents = N
    # Adding the scheduler:
    self.schedule = RandomActivation(self)        # 1.
    self.create_agents()

  def create_agents(self):
    """Method to create all the agents."""
    for i in range(self.num_agents):
      a = MoneyAgent(i)
      self.schedule.add(a)

  def step(self):
    self.schedule.step()                          # 2.

  def run_model(self, steps):
    for _ in range(steps):                        # 3.
      self.step()
```

1) Scheduler objects are instantiated with their Model object, which they then pass to the agents at each step.
2) The scheduler's `step` method activates the `step` methods of all the agents that have been added to it, in this case in random order.
3) Because the model has no inherent end conditions, the user must specify how many steps to run it for.

Space

Many agent-based models have a spatial element. In spatial models, agents may have fixed positions or move around, and interact with their immediate neighbors or with agents and other objects nearby. The space may be abstract (as in many cellular automata), or represent many possible scales, from a single building to a region to the entire world. The majority of models use two-dimensional spaces, which is how Mesa's current space modules are implemented. Many abstract model spaces are toroidal (doughnut-shaped), meaning that the edges 'wrap around' to the opposite edge. This prevents model artifacts from arising at the edges, which have fewer neighbors than other locations.

Mesa currently implements two broad classes of space: grid, and continuous. Grids are discrete spaces, consisting of rectangular cells; agents and other objects may only be in a particular cell (or, with some additional coding, potentially span multiple cells), but not between cells. In continuous space, in contrast, agents can have any arbitrary coordinates. Both types of space assume by default that agents store their location as an (x, y) tuple named `pos`.

There are several specific grid classes, all of which inherit from a root *Grid* class. At its core, a grid is a two-dimensional array with methods for getting the neighbors of particular cells, adding and removing agents, etc. The default `Grid` class does not enforce what each cell may contain. However, `SingleGrid` ensures that each cell contains at most one object, while `MultiGrid`

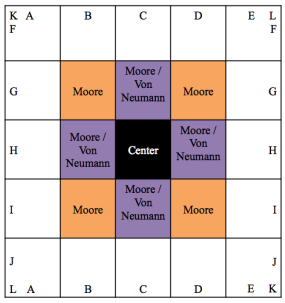


Fig. 3: *Grid topology. Moore and Von Neumann neighborhoods of radius 1; in a torus, lettered edges connect to one another.*

explicitly makes each cell be a set of 0 or more objects. There are two kinds of cell neighborhoods: The first is a cell's *Moore* neighborhood that is the 8 cells surrounding it, including the diagonals; the second is the *Von Neumann* neighborhood which is only the 4 cells immediately above, below, and to its left and right. Which neighborhood type to use will vary based on the specifics of each model, and are specified in Mesa by an argument to the various neighborhood methods.

The `ContinuousSpace` class also inherits from `Grid`, and uses the grid as a way of speeding up neighborhood lookups; the number of cells and the arbitrary limits of the space are provided when the space is created, and are used internally to map between spatial coordinates and grid cells. Neighbors here are defined as all agents within an arbitrary distance of a given point. To find the neighbors of a given point, `ContinuousSpace` only measures the distance for agents in cells intersecting with a circle of the given radius.

To add space to our example model, we can have the agents wander around a grid; instead of giving a unit of money to any random agent, they pick an agent in the same cell as themselves. This means that multiple agents are allowed in each cell, requiring a `MultiGrid`.

```
from mesa.space import MultiGrid

class MoneyModel(Model):
  def __init__(self, N, width, height, torus):
    self.grid = MultiGrid(height, width, torus)   # 1.
    # ... everything else

  def create_agents(self):
    for i in range(self.num_agents):
      # ... everything above
      x = random.randrange(self.grid.width)
      y = random.randrange(self.grid.width)
      self.grid.place_agent(a, (x, y))            # 2.

  class MoneyAgent(Agent):
    # ...
    def move(self, model):
      """Take a random step."""
      grid = model.grid
      x, y = self.pos
      possible_steps = grid.get_neighborhood(x, y,
        moore=True, include_center=True)          # 3.
      choice = random.choice(possible_steps)
      grid.move_agent(self, choice)               # 4.

    def give_money(self, model):
      grid = model.grid
      pos = [self.pos]
      others = grid.get_cell_list_contents(pos)   # 5.
      if len(others) > 1:
        other = random.choice(others)
        other.wealth += 1
        self.wealth -= 1

  def step(self, model):
    self.move(model)
    if self.wealth > 0:
      self.give_money(model)
```

1) The arguments needed to create a new grid are its width, height, and a boolean for whether it is a torus or not.
2) The `place_agent` method places the given object in the grid cell specified by the `(x, y)` tuple, and assigns that tuple to the agent's `pos` property.
3) The `get_neighborhood` method returns a list of coordinate tuples for the appropriate neighbors of the given coordinates. In this case, it's getting the Moore neighborhood (including diagonals) and includes the center cell. The agent decides where to move by choosing one of those tuples at random. This is a good way of handling random moves, since it still works for agents on an edge of a non-toroidal grid, or if the grid itself is hexagonal.
4) the `move_agent` method works like `place_agent`, but removes the agent from its current location before placing it in its new one.
5) This is a helper method which returns the contents of the entire list of cell tuples provided. It's not strictly necessary here; the alternative would be: `x, y = self.pos; others = grid[y][x]` (note that grids are indexed y-first).

Once the model has been run, we can create a static visualization of the distribution of wealth across the grid using the `coord_iter` iterator, which allows us to loop over the contents and coordinates of all cells in the grid, with output shown in figure 4.

```
wealth_grid = np.zeroes(model.grid.width,
                        model.grid.height)
for cell in model.grid.coord_iter():
  cell_content, x, y = cell
  cell_wealth = sum(a.wealth for a in cell_content)
  wealth_grid[y][x] = cell_wealth
plt.imshow(wealth_grid, interpolation='nearest')
```

Data Collection

An agent-based model is not particularly useful if there is no way to see the behaviors and outputs it produces. Generally speaking, there are two ways of extracting these: visualization, which allows for observation and qualitative examination (and which we will discuss later in this paper), and quantitative data collection. In order to facilitate the latter option, we provide a generic `DataCollector` class, which can store and export data from most models without needing to be subclassed.

The data collector stores three categories of data: *model-level* variables, *agent-level variables*, and *tables* which are a catch-all for everything else. Model- and agent-level variables are added to the data collector along with a function for collecting them. Model-level collection functions take a model object as an input, while agent-level collection functions take an agent object as an input. Both then return a value computed from the model or each agent at their current state. When the data collector's `collect` method is called, with a model object as its argument, it applies each model-level collection function to the model, and

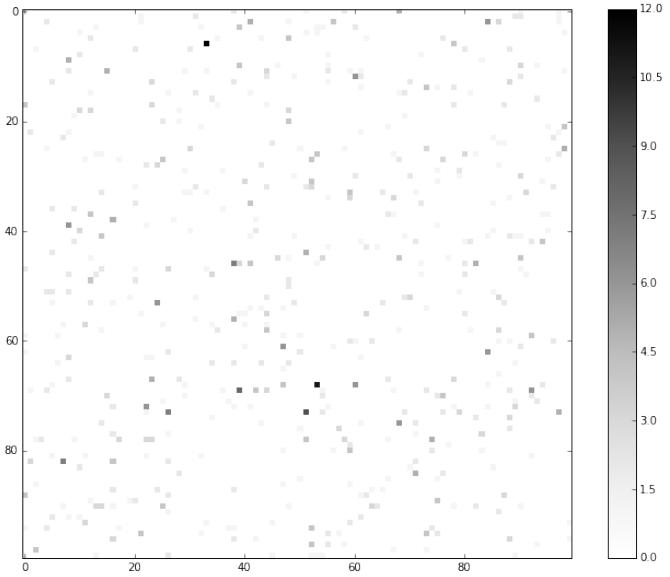


Fig. 4: *Example of spatial wealth distribution across the grid.*

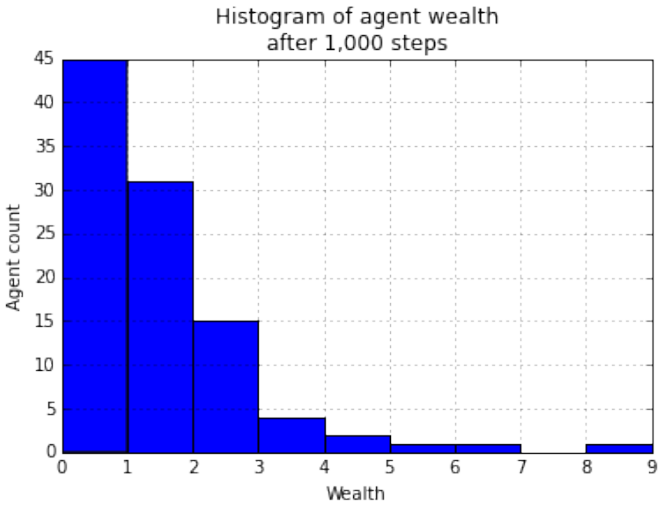


Fig. 5: *Example of model output histogram, with labels added.*

stores the results in a dictionary, associating the current value with the current step of the model. Similarly, the method applies each agent-level collection function to each agent currently in the schedule, associating the resulting value with the step of the model, and the agent's unique ID. The data collector may be placed within the model class itself, with the collect method running as part of the model step; or externally, with additional code calling it every step or every *N* steps of the model.

The third category, *tables*, is used for logging by the model or the agents rather than fixed collection by the data collector itself. Each table consists of a set of columns. The model or agents can then append records to a table according to their own internal logic. This can be used to log specific events (e.g. every time an agent is killed), and data associated with them (e.g. agent lifespan at destruction), particularly when these events do not necessarily occur every step.

Internally, the data collector stores all variables and tables in Python's standard dictionaries and lists. This reduces the need for external dependencies, and allows the data to be easily exported to JSON or CSV. However, one of the goals of Mesa is facilitating integration with Python's larger scientific and data-analysis ecosystems, and thus the data collector also includes methods for exporting the collected data to *pandas* data frames. This allows rapid, interactive processing of the data, easy charting, and access to the full range of statistical and machine-learning tools that are compatible with pandas.

To continue our example, we use a data collector to collect the wealth of each agent at the end of every step. The additional code this requires can look like this:

```
from mesa.datacollection import DataCollector

class MoneyModel(Model):

  def __init__(self, N):
    # ... everything above
    ar = {"Wealth": lambda a: a.wealth}
    self.dc = DataCollector(agent_reporters=ar)

  def step(self):
    self.dc.collect(self)
    self.schedule.step()
```

We now have enough code to run the model, get some data out of it, and analyze it.

```
# Create a model with 100 agents on a torus 10x10 grid
model = MoneyModel(100, 10, 10, True)
# Run it for 1,000 steps:
model.run_model(1000)
# Get the data as a DataFrame
wealth_history = model.dc.get_agent_vars_dataframe()
# wealth_history indexed on Step and AgentID, and...
# ...has Wealth as one data column
wealth_history.reset_index(inplace=True)
# Plot a histogram of final wealth
wealth_history[wealth_history.Step==999].\
  Wealth.hist(bins=range(10))
```

An example of the output of this code is shown in Figure 5. Notice that this simple rule, where agents give one another 1 unit of money at random, produces an extremely skewed wealth distribution -- in fact, this is approximately a Boltzmann distribution, which characterizes at least some real-world wealth distributions [Dragulescu2001].

Batch Runner

Since most ABMs are stochastic, a single model run gives us only one particular realization of the process the model describes. Furthermore, the questions we want to use ABMs to answer are often about how a particular parameter drives the behavior of the entire system -- requiring multiple model runs with different parameter values. In order to facilitate this, Mesa provides the `BatchRunner` class. Like the DataCollector, it does not need to be subclassed in order to conduct parameter sweeps on most models.

`BatchRunner` is instantiated with a model class, and a dictionary mapping names of model parameters to either a single value, or a list or range of values. Like the data collector, it is also instantiated with dictionaries mapping model- and agent-level variable names to functions used to collect them. The batch runner uses the `product` combination generator included in Python's `itertools` library to generate all possible combinations of the parameter values provided. For each combination, the batch collector instantiates a model instance with those parameters, and

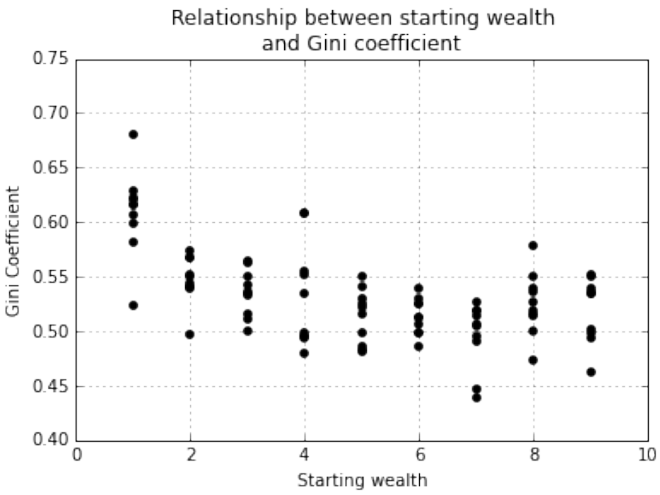


Fig. 6: *Example of batch run scatter-plot, with labels added.*

runs the model until it terminates or a set number of steps has been reached. Once the model terminates, the batch collector runs the reporter functions, collecting data on the completed model run and storing it along with the relevant parameters. Like the data collector, the batch runner can then export the resulting datasets to pandas data frames.

Suppose we want to know whether the skewed wealth distribution in our example model is dependent on initial starting wealth. To do so, we modify the model code to allow for variable starting wealth, and implement a `get_gini` method to compute the model's Gini coefficient. (In the interest of space, these modifications are left as an exercise to the reader, or are available in the full model code online). The following code sets up and runs a `BatchRunner` testing starting wealth values between 1 and 9, with 10 runs at each. Each run continues for 1,000 steps, as above.

```
param_values = {"N": 100,
                "starting_wealth": range(1,10)}
model_reporter={"Gini": compute_gini}
batch = BatchRunner(MoneyModel, param_values,
                    10, 1000, model_reporter)
batch.run_all()
out = batch.get_model_vars_dataframe()
plt.scatter(df.starting_wealth, df.Gini)
```

Output from this code is shown in Figure 6.

Visualization

Mesa uses a browser window to visualize its models. This avoids both the developers and the users needing to deal with cross-system GUI programming; more importantly, perhaps, it gives us access to the universe of advanced JavaScript-based data visualization tools. The entire visualization system is divided into two parts: the server side, and the client side. The server runs the model, and at each step extracts data from it to visualize, which it sends to the client as JSON via a WebSocket connection. The client receives the data, and uses JavaScript to actually draw the data onto the screen for the user. The client front-end also includes a GUI controller, allowing the user to start a model run, pause it, advance it by one step, reset the model, and set the desired frame-rate.

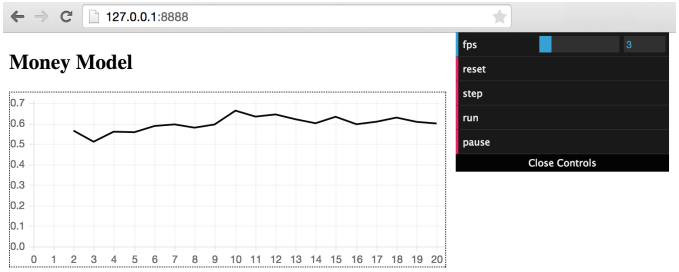


Fig. 7: *Example of the browser visualization.*

Mesa already includes a set of pre-built visualization elements which can be deployed with minimal setup. For example, to create a visualization of the example model which displays a live chart of the Gini coefficient at each step, we can use the included `ChartModule`.

```
from mesa.visualization.ModularVisualization \
  import ModularServer
from mesa.visualization.modules import ChartModule

# The Chart Module gets a model-level variable
# from the model's data collector
chart_element = ChartModule([{"Label": "Gini",
                      "Color": "Black"}],
                      data_collector_name='dc')   # 1.
# Create a server to visualize MoneyModel
server = ModularServer(MoneyModel,                # 2.
                      [chart_element],
                      "Money Model", 100)
server.launch()
```

1) We instantiate a visualization element object: ChartModule, which plots model-level variables being collected by the model's data collector as specified by the "Labels" provided. `data_collector_name` is the name of the actual DataCollector variable, so the module knows where to find the values.
2) The server is instantiated with the model class; a list of visualization elements (in this case, there's only the one element), a model name, and model arguments (in this case, just the agent count).

Running this code launches the server. To access the actual visualization, open your favorite browser (ideally Chrome) to http://127.0.0.1:8888/ . This displays the visualization, along with the controls used to reset the model, advance it by one step, or run it at the designated frame-rate. After several ticks, the browser window will look something like Figure 7.

The actual visualization is done by the visualization modules. Conceptually, each module consists of a server-side and a client-side element. The server-side element is a Python object implementing a `render` method, which takes a model instance as an argument and returns a JSON-ready object with the information needed to visualize some part of the model. This might be as simple as a single number representing some model-level statistic, or as complicated as a list of JSON objects, each encoding the position, shape, color and size of an agent on a grid.

The client-side element is a JavaScript class, which implements a `render` method of its own. This method receives the JSON data created by the Python element, and renders it in the browser. This can be as simple as updating the text in a particular HTML paragraph, or as complicated as drawing all the shapes

described in the aforementioned list. The object also implements a `reset` method, used to reset the visualization element when the model is reset. Finally, the object creates the actual necessary HTML elements in its constructor, and does any other initial setup necessary.

Obviously, the two sides of each visualization must be designed in tandem. They result in one Python class, and one JavaScript `.js` file. The path to the JavaScript file is a property of the Python class, meaning that a particular object does not need to include it separately. Mesa includes a variety of pre-built elements, and they are easy to extend or add to.

The `ModularServer` class manages the various visualization modules, and is meant to be generic to most models and modules. A visualization is created by instantiating a `ModularServer` object with a model class, one or more `VisualizationElement` objects, and model parameters (if necessary). The `launch()` method then launches a Tornado server, using templates to insert the JavaScript code specified by the modules to create the client page. The application uses Tornado's coroutines to run the model in parallel with the server itself, so that the model running does not block the serving of the page and the WebSocket data. For each step of the model, each module's `render` method extracts the visualization data and stores it in a list. That list item is then sent to the client via WebSocket when the request for that step number is received.

Let us create a simple histogram, with a fixed set of bins, for visualizing the distribution of wealth as the model runs. It requires JavaScript code, in *HistogramModule.js* and a Python class. Below is an abbreviated version of both.

```
var HistogramModule = function(bins) {
  // Create the appropriate tag, stored in canvas
  $("body").append(canvas);                          // 1.
  // ... Chart.js boilerplate removed
  var chart = new Chart(context).Bar(data, options);

  this.render = function(data) {                     // 2.
    for (var i in data)
      chart.datasets[0].bars[i].value = data[i];
    chart.update();
  };

  this.reset = function() {                          // 3.
    chart.destroy();
    chart = new Chart(context).Bar(data, options);
    };
  };
```

1) This block of code functions as the object's constructor. It adds and saves a `canvas` element to the HTML page body, and creates a *Chart.js* bar chart inside of it.
2) The `render` method takes a list of numbers as an input, and assigns each to the corresponding bar of the histogram.
3) To `reset` the histogram, this code destroys the chart and creates a new one with the same parameters.

Next, the Python class tells the front-end to include `Chart.min.js` (included with the Mesa package) and the new `HistogramModule.js` file we created above, which is located in the same directory as the Python code[3]. In this case, our module's `render` method is extremely specific for this model alone. The code looks like this.

```
class HistogramModule(VisualizationElement):
  package_includes = ["Chart.min.js"]
  local_includes = ["HistogramModule.js"]

  def __init__(self, bins):
    self.bins = bins
    new_element = "new HistogramModule({})"       # 1.
    new_element = new_element.format(bins)
    self.js_code = "elements.push("               # 2.
    self.js_code += new_element +");"

def render(self, model):
  wealth_vals = [a.wealth
                 for a in model.schedule.agents]
  hist = np.histogram(wealth_vals,
                      bins=self.bins)[0]
  return [int(x) for x in hist]
```

1) This line, and the line below it, prepare the code for actually inserting the visualization element; creating a new element, with the bins as an argument.
2) `js_code` is a string of JavaScript code to be run by the front-end. In this case, it takes the code for creating a visualization element and inserts it into the front-end's `elements` list of visualization elements.

Finally, we can add the element to our visualization server object:

```
histogram_element = HistogramModule(range(10))
server = ModularServer(MoneyModel,
                       [histogram_element],
                       "MoneyModel", 100)
server.launch()
```

Conclusions and Future Work

Mesa provides a versatile framework for building, analyzing and visualizing agent-based models. It seeks to fill the ABM-shaped hole in the scientific Python ecosystem, while bringing together powerful features found in other modeling frameworks and introducing some of its own. Both Mesa's schedule architecture and in-browser visualization are, to the best of our knowledge, unique among major ABM frameworks.

Despite this, Mesa is very much a work in progress. We intend to implement several key features in the near future, including inter-agent networks and the corresponding visualization, a better system to set model runs' random seed, and tools for reading and writing model states to disk. The server-side visualization is also structured so as to allow video-style scrubbing forwards and backwards through a model run, and we hope to implement this feature soon as well. In the longer term, we hope to add tools for geospatial simulations, and for easier distribution of a batch run or even a single model run across multiple cores or in a cluster. We also intend to iteratively continue to add to Mesa's documentation, increase its efficiency, and improve the visualization quality.

We also hope to continue to leverage Mesa's open-source nature. As more researchers utilize Mesa, they will identify opportunities for improvement and additional features, hopefully

3. While the best practice in web development is to host static files (e.g. JavaScript) separately, Mesa is not set up to this way, as the models are currently small and run only locally. As we scale the Mesa framework, we expect that the ability to pull in external javascript files to be part of the optimization process.

contribute them to the main repository. More models will generate reference code or additional stand-alone modules, which in turn will help provide a larger library of reusable modeling components that have been validated both in terms of their code and scientific assumptions.

We are happy to introduce Mesa to the world with this paper; it marks not the end of a research effort, but the beginning of an open, collaborative process to develop and expand a new tool in Python's scientific ecosystem.

Acknowledgements

Mesa is an open-source project, and we are happy to acknowledge major code contributors Kim Furuya, Daniel Weitzenfeld, and Eugene Callahan.

REFERENCES

[Axtell2000] Axtell, Robert. "Why agents?: on the varied motivations for agent computing in the social sciences." Center on Social and Economic Dynamics. The Brookings Institution. (2000).

[Comer2014] Comer, Kenneth W. "Who Goes First? An Examination of the Impact of Activation on Outcome Behavior in Agent-Based Models." George Mason University, 2014. http://gradworks.umi.com/36/23/3623940.html.

[Dragulescu2001] Drăgulescu, Adrian, and Victor M. Yakovenko. "Exponential and Power-Law Probability Distributions of Wealth and Income in the United Kingdom and the United States." Physica A: Statistical Mechanics and Its Applications 299, no. 1 (2001): 213–21.

[Dragulescu2002] Drăgulescu, Adrian A., and Victor M. Yakovenko. "Statistical Mechanics of Money, Income, and Wealth: A Short Survey." arXiv Preprint Cond-mat/0211175, 2002. http://arxiv.org/abs/cond-mat/0211175.

[Luke2005] Luke, Sean, Claudio Cioffi-Revilla, Liviu Panait, Keith Sullivan, and Gabriel Balan. "Mason: A Multiagent Simulation Environment." Simulation 81, no. 7 (2005): 517–27.

[McKinney2011] McKinney, Wes. "Pandas: A Foundational Python Library for Data Analysis and Statistics." Python for High Performance and Scientific Computing, 2011, 1–9.

[Miller2009] Miller, John H., and Scott E. Page. "Complex Adaptive Systems: An Introduction to Computational Models of Social Life." Princeton University Press, 2009.

[North2013] North, Michael J., Nicholson T. Collier, Jonathan Ozik, Eric R. Tatara, Charles M. Macal, Mark Bragen, and Pam Sydelko. "Complex Adaptive Systems Modeling with Repast Simphony." Complex Adaptive Systems Modeling 1, no. 1 (March 13, 2013): 3. doi:10.1186/2194-3206-1-3.

[Perez2007] Fernando Pérez, Brian E. Granger. "IPython: A System for Interactive Scientific Computing." Computing in Science and Engineering, vol. 9, no. 3, pp. 21-29, May/June 2007, doi:10.1109/MCSE.2007.53. URL: http://ipython.org

[Perez2011] Pérez, Fernando, Brian E. Granger, and John D. Hunter. "Python: An Ecosystem for Scientific Computing." Computing in Science & Engineering 13, no. 2 (March 1, 2011): 13–21. doi:10.1109/MCSE.2010.119.

[Sayama2013] Sayama, Hiroki. "PyCX: A Python-Based Simulation Code Repository for Complex Systems Education." Complex Adaptive Systems Modeling 1, no. 1 (March 13, 2013): 1–10. doi:10.1186/2194-3206-1-2.

[Schelling1971] Schelling, Thomas C. "Dynamic models of segregation." Journal of Mathematical Sociology 1.2 (1971): 143-186.

[Wilensky1999] Wilensky, Uri. "NetLogo." Evanston, IL: Center for Connected Learning and Computer-Based Modeling, Northwestern University, 1999.

[Zvoleff2013] Zvoleff, Alex. "PyABM Toolkit." http://azvoleff.com/pyabm.html.

HoloViews: Building Complex Visualizations Easily for Reproducible Science

Jean-Luc R. Stevens[‡†*], Philipp Rudiger[‡†], James A. Bednar[‡]

https://www.youtube.com/watch?v=hNsR2H7Lrg0

Abstract—Scientific visualization typically requires large amounts of custom coding that obscures the underlying principles of the work and makes it difficult to reproduce the results. Here we describe how the new HoloViews Python package, when combined with the IPython Notebook and a plotting library, provides a rich, interactive interface for flexible and nearly code-free visualization of your results while storing a full record of the process for later reproduction.

HoloViews provides a set of general-purpose data structures that allow you to pair your data with a small amount of metadata. These data structures are then used by a separate plotting system to render your data interactively, e.g. within the IPython Notebook environment, revealing even complex data in publication-quality form without requiring custom plotting code for each figure.

HoloViews also provides powerful containers that allow you to organize this data for analysis, embedding it whatever multidimensional continuous or discrete space best characterizes it. The resulting workflow allows you to focus on exploring, analyzing, and understanding your data and results, while leading directly to an exportable recipe for reproducible research.

Index Terms—reproducible, interactive, visualization, notebook

Introduction

Scientific research alternates between stretches of speculative, exploratory investigation and periods where crucial findings are distilled and disseminated as publications or reports. The exploratory phase typically involves running many different analyses with interactive plotting tools before the important aspects of the data are determined. The final results are then typically prepared as static figures for dissemination, often putting together many subfigures into a complicated figure that reveals multiple interrelated aspects of the results.

Current software tools provide relatively poor support for this dual exploring/reporting nature of scientific research, severely limiting scientific progress. On the one hand, developing new exploratory visualizations typically requires large amounts of custom software coding, which is slow, error-prone, and distracts from the actual scientific analysis. Moreover, this process typically involves a large amount of trial and error, generating transitory code and analyses that make it difficult to later reproduce the steps that led to any particular result [Cro13]. Switching to different tools for final, non-interactive, publication-quality figures exacerbates this problem, further disconnecting the reported results from the process by which they were created. This lack of reproducibility is a serious handicap both for progress within a single lab and for the community as a whole, making it nearly impossible for researchers to build on each others' work even for purely computational projects [Cro13].

Here we will describe a new Python software package built to address these problems directly, by providing simple tools for gradually building elaborate visualizations and analyses interactively yet reproducibly. HoloViews supports immediate exploration of data as it is obtained, without requiring custom coding, and then supports incrementally revealing more complex relationships between datasets, culminating in the final publication of fully reproducible scientific results.

In this paper we will focus on the high-level design principles that allow HoloViews to achieve these goals and we encourage the reader to visit holoviews.org for concrete examples. As detailed below, we show how this is achieved by enforcing a strict separation in the declaration of the semantic properties of the data and the specification of plotting options, allowing the user to declaratively specify their intent and let HoloViews handle the visualization.

The interactive interpreter

To understand this approach, we need to consider the history of how we interact with computational data. The idea of an interactive programming session originated with the earliest LISP interpreters in the late 1950s and remains a popular way to interact with dynamic languages such as Python.

However, like most such command prompts, the standard Python prompt is a text-only environment. Commands are entered by the user, parsed, and executed, with results displayed as text. This offers immediate feedback and works well for data that is naturally expressed in a concise textual form. Unfortunately, this approach begins to fail when the data cannot be usefully visualized as text, as is typical for the large datasets now commonplace. In such instances, a separate plotting package offering a rich graphical display would normally be used to present the results outside the environment of the interpreter, via a graphical user interface.

This disjointed approach reflects history: text-only environments, where interactive interpreters were first employed, appeared long before any graphical interfaces. To this day, text-only interpreters are standard due to the relative simplicity of working with text. Proprietary attempts to overcome these limitations,

† *These authors contributed equally.*
∗ *Corresponding author: jlstevens@ed.ac.uk*
‡ *Institute for Adaptive and Neural Computation, University of Edinburgh*

such as the Mathematica Notebook [Wol03], have remained constrained by limited interoperability and a lack of standardized open formats. Other approaches focusing explicitly on reproducibility involve building a recipe for reproducing results only at the end of the scientific project [knitr], when it is often too late to capture the important steps involved. Here we consider how graphical output can be integrated fully into an interactive workflow, addressing both exploration and reproducibility simultaneously.

Fixing the disconnect between data and representation

At the same time as text-based interpreters have failed to overcome the inherent limitations of working with rich data, the web browser has emerged as a ubiquitous means of interactively working with rich media documents. In addition to being universally available, web browsers have the benefit of being based on open standards that remain supported almost indefinitely. Although early versions of the HTML standard only allowed passive page viewing, the widespread adoption of HTML5 has made it possible for anyone to interact with complex, dynamic documents in a bi-directional manner.

The emergence of the web browser as a platform has been exploited by the Python community and the scientific community at large with tools such as the IPython Notebook [Per07] and SAGE MathCloud [Ste05]. These projects offer interactive computation sessions in a notebook format instead of a traditional text prompt. Although similar in design to the traditional text-only interpreters, these notebooks allow embedded graphics or other media (such as video) while maintaining a record of useful commands in a rich document that supports the gradual development of a document with interleaved code, results, and exposition.

Yet despite the greatly improved interactive capabilities of these tools, the spirit of the original interpreter has not yet been restored: there is still an ongoing disconnect between data and its representation. This artificial distinction is a lingering consequence of text-only displays, forcing a strict split between how we conceptualize "simple" and "complex" data. Although the IPython notebook now offers the means to give objects rich media representations, few packages have so far embraced this and none have supported easy composition of related figures. As a result the most common way to visualize complex data remains for the user to specify a detailed list of steps to get subfigures using an external plotting package such as Matplotlib [Hun07], then often combining subfigures using a GUI-based image editor.

Here we introduce HoloViews, a library of simple classes designed to provide an immediately available representation for even complex data in notebooks, analogous to the way simple datatypes are displayed in interactive sessions. HoloViews is not a plotting package; instead, it offers a set of useful data structures paired with rich, customizable visual representations that display effortlessly in the IPython Notebook environment. The result is research that is more interactive, concise, declarative, and reproducible. Figure 1 shows a self-contained example of building a complex visualization showing the declaration of an `Image` object followed by an example of how to compose HoloViews objects together.

Design principles

The core design principle of HoloViews is to *automatically* and *transparently* return and display declarative data structures to the user for immediate feedback without requiring additional code. Although this concept is familiar and intuitive when interactively working with simple data types, it is worth reviewing explicitly what is going on so that the appropriate graphical extension of these ideas is clear.

When executing an addition operation like `1 + 2.5` at a Python prompt, the expression is parsed, converted into bytecode, and then executed, resulting in the float value `3.5`. This floating-point value is immediately returned to the user in the appropriate displayable representation, giving the user immediate feedback. Of course, this representation is not the float itself, but the string `"3.5"`. Such strings are automatically generated by the interpreter, via the displayed object's `__repr__` method.

The Python interpreter also provides such automatic, immediate feedback for more complex data types like large NumPy arrays, but for such data the displayed string has very little utility because it is either incomplete or impractical. In a terminal, this restriction is a result of the `__repr__` method only supporting a text-based display value. Using HoloViews in the IPython Notebook, you can give your array a more useful, interpretable default visual representation as an image, curve, or similar plot according to the following principles:

- It must be easy to assign a useful and understandable default representation to your data. The goal is to keep the initial barrier to productivity as low as possible -- data should simply reveal itself.
- These atomic data objects (elements) should be almost trivially simple wrappers around your data, acting as proxies for the contained arrays along with a small amount of semantic metadata (such as whether the user thinks of some particular set of data as a continuous curve or as a discrete set of points).
- Any metadata included in the element must address issues of *content* and not be concerned with *display* issues -- elements should hold essential information only.
- There are always numerous aesthetic alternatives associated with rich visual representations, but such option settings should be stored and implemented entirely separately from the content elements, so that elements can be generated, archived, and distributed without any dependencies on the visualization code.
- As the principles above force the atomic elements to be simple, they must then be *compositional* in order to build complex data structures that reflect the interrelated plots typical of publication figures.

The outcome of these principles is a set of compositional data structures that contain only the essential information underlying potentially complex, publication-quality figures. These data structures have an understandable, default visualization that transparently reveals their contents, making them a useful proxy for the data itself, just as the text `3.5` is a proxy for the underlying floating-point value. This default visualization may then be customized declaratively to achieve the desired aesthetics, without complicating the objects themselves.

In the next section we will discuss the data structures that hold the important content. Starting with the simple primitive elements, we examine how they can be composed into complex figures and embedded in high-dimensional spaces for exploration. Along the way we will discover how our implementation realizes the design principles outlined and manages to keep the state of the data separate from its visual representation.

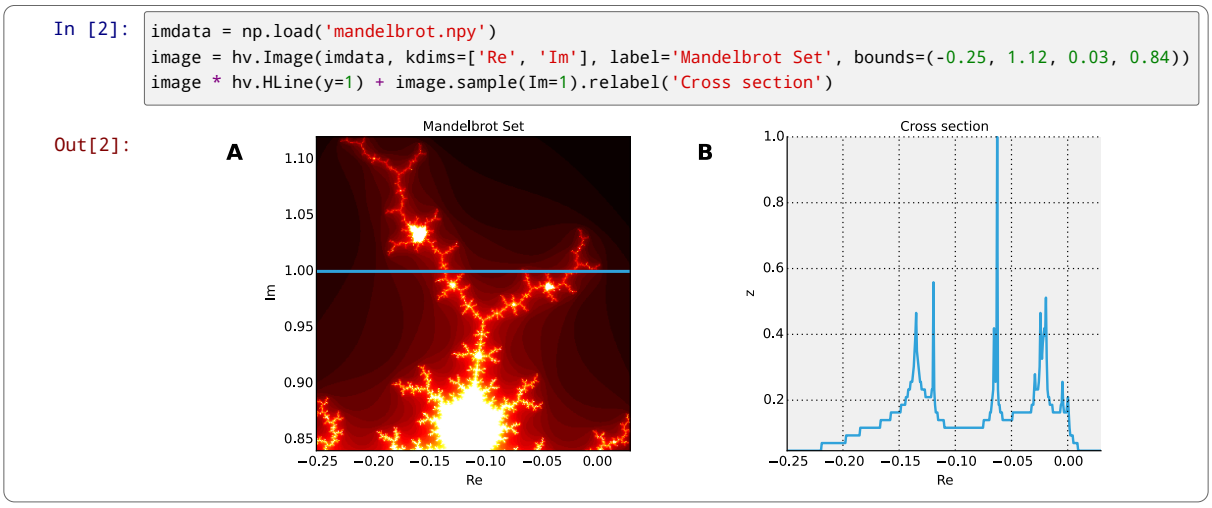


Fig. 1: *Example of a composite HoloViews data structure and how it is displayed in an IPython Notebook session. The* `imdata` *array loaded using Numpy corresponds to the displayed portion of the Mandelbrot set.* ***A.*** *The* `Image` *element displays* `imdata` *overlaid via the* `*` *operator with a horizontal line element (*`HLine`*).* ***B.*** *A* `Curve` *element generated via the* `.sample()` *method of the image, showing a cross-section of the fractal along the indicated blue horizontal line. The curve is concatenated with the* `Overlay` *in* ***A*** *via the* `+` *operation.*

Data Structures

In this section we discuss the data structures that hold the raw data and the essential semantic content of interest. The Elements section introduces each of the primitives, and the Collections section explains how they can be combined. Finally, we will discuss working with Elements embedded in high-dimensional continuous or discrete spaces.

Elements

The atomic classes that wrap raw data are the `Element` primitives. These classes are named by the natural representation they suggest for the supplied data, with `Image`, `Curve`, and `Scatter` being some simple examples. These elements are easily constructed as they only require the raw data (such as a NumPy array) to display.

In Figure 1, we have some examples of the Element primitives. On the left, in subfigure **A**, we see the `Image` primitive containing a two-dimensional NumPy array. This `Image` is declared by supplying the NumPy array `imdata` along with the optional metadata, including a suitable label and a declaration of the bounding region in the complex plane. The visual output is automatically generated and shows that the array is a part of the Mandelbrot set. Our object merely holds the supplied NumPy array, which remains easily accessed via the `.data` attribute. In part **B** of Figure 1 we have an example of a `Curve` containing a horizontal cross section of the image, as computed by the `sample` method.

Although the names of the `Elements` suggest that these objects are about visualization, they are primarily concerned with content and *not* display. The visually meaningful class names offer a convenient way to intuitively understand the dimensionality of the data in terms of an appropriate visual representation. For instance, in Figure 1 **A**, the name `Image` conveys the notion that the contained data is in the form of a two-dimensional NumPy array that can be meaningfully displayed as an image.

The particular `Image` shown in Figure 1 **A** was constructed as a visualization of the Mandelbrot Set, defined in the complex plane. In particular, the `kdims` argument declares that the *x*-axis is along the real axis and that the *y*-axis is along the imaginary axis. This information is then reflected in the visual output by assigning the appropriate axis labels. This semantic information is also passed to the `Curve` object generated by sampling the image using `image.sample(Im=1)`.

This `Curve` object is also able to pass on this semantic information to other Elements with different visual representations so that they faithfully reflect the space in which the Mandelbrot Set is defined. For instance, you can pass the curve directly to the constructor of the `Scatter` or `Histogram` elements and a new visual representation of the resulting object will retain the original semantic dimension labels. This type of operation merely changes the representation associated with the supplied data.

Note that in the declarations of `Image`, the dimensions of the axes are declared as key dimensions (`kdims`). Key dimensions correspond to the independent dimensions used to index or slice the element, with the remaining dimensions called value dimensions (`vdims`). In the case of this image, there is a single value dimension, for the values in the supplied NumPy array, which are then visualized using the default colormap of the `Image` elements (the 'hot' color map).

As key dimensions are indexable and sliceable, we can slice the `Image` to select a different subregion of the Mandelbrot Set. Continuous values are supported when slicing an `Image` and the result is then a new `Image` containing the portion of the original NumPy array appropriate to the specified slice. The mapping between continuous space and the discrete array samples is specified by the bounds, allowing us to apply the slice `[-0.2:0,0.85:1.05]` to select the corresponding part of the complex plane. The first component of this slice selects the first key dimension (the real axis `'Re'`) from *-0.2* to *0.0* while the second component of the slice selects the second key dimension (the imaginary axis `'Im'`) from *0.85* to *1.05*. You can apply a similar slice along the real axis to select a portion of the curve object shown in Figure 1 **B**.

There are many additional element classes, one for each of the common visual representations for data. These elements form an extensible library of primitives that allow the composition of data structures with complex, meaningful visualizations. Within the set

of all elements, you can cast your data between representations so long as the number of key and value dimensions is consistent. You can then index and slice your elements along their respective key dimensions to get new elements holding the appropriately sliced data of interest.

Collections

The elements are simple wrappers that hold the supplied data and allow a rich, meaningful default representation. An individual element is therefore a data structure holding the semantic contents corresponding to a simple visual element of the sort you may see in a publication. Although the elements are sufficient to cover simple cases such as individual graphs, raster images, or histogram, they are not sufficient to represent more complex figures.

A typical published figure does not present data using a single representation, but allows comparison between related data items in order to illustrate similarities or differences. In other words, a typical figure is an object composed of many visual representations combined together. HoloViews makes it trivial to compose elements in the two most common ways: concatenating representations into a single figure, or overlaying visual elements within the same set of axes.

These types of composition are so common that both have already been used in Figure 1 as our very first example. The + operation implements concatenation, and * implements overlaying elements together. When you compose an object using the + operator, a default four-column layout is used but you can specify the desired number of columns using the `.cols` method. Layouts are easily specified but also support multiple options for customizing the position and sizing of elements.

When we refer to subfigures 1 **A** and 1 **B**, we are making use of labels generated by HoloViews for representing a composite data structure called a `Layout`. Similarly, subfigure 1 **A** is itself a composite data structure called an `Overlay` which, in this particular case, consists of an `Image` element overlaid by the `HLine` element.

The overall data structure that corresponds to Figure 1 is therefore a `Layout` which itself contains another composite collection in the form of an `Overlay`. The object in Figure 1 is in fact a highly flexible, compositional tree-based data structure: intermediate nodes correspond either to `Layout` nodes (+) or `Overlay` nodes (*), with element primitives at the leaf nodes. Even in this potentially complex tree, all the raw data corresponding to every visual element is conveniently accessible via key or attribute access by selecting a leaf element using its path through the tree, and then inspecting the `.data` attribute, making it simple to declare which part of a complex dataset you want to work with at a given time.

As any element may be a leaf of such a tree, there needs to be an easy way to select subtrees or leaf elements. This is achieved with a semantic, two-level labeling system using "group" and "label" strings supported throughout HoloViews. We have seen an example of a label string in Figure 1, where it was used to title the image "Mandelbrot Set". The textual representation of the layout in Figure 1 (see Out[6] of Figure 4) shows how the supplied label is used in the attribute-based indexing scheme of the layout. The strings "Image", "Overlay", "HLine" and "Curve" are default group names, but you can supply your own names to define semantic groupings for your data. To illustrate this system, you can access the sampled data (a NumPy array) in Figure 4 using `content.Curve.Cross_Section.data`.

With the ability to overlay or concatenate any element with any other, there is great flexibility to declare complex relationships between elements. Whereas a single element primitive holds semantic information about a particular piece of data, trees encode semantic information between elements. The composition of visual elements into a single visual representation expresses some underlying semantic value in grouping these particular chunks of data together. This is what composite trees capture; they represent the overall *semantic content* of a figure in a highly composable and flexible way that always preserves both the raw data and associated metadata for further interactive analysis and reproduction.

Spaces

A single plot can represent at most a few dimensions before it becomes visually cluttered. Since real-world datasets often have higher dimensionality, we face a tradeoff between representing the full dimensionality of our data, and keeping the visual representation intelligible and therefore effective. In practice we are limited to two or at most three spatial axes, in addition to attributes such as the color, angle, and size of the visual elements. To effectively explore higher dimensional spaces we therefore have to find other solutions.

One way of dealing with this problem is to lay out multiple plots spatially. Plotting packages like ggplot [Wic09] and seaborn [Was14] have shown how this can be done easily using various grid-based layouts. Another solution is to present the data sequentially over time as an animation. A third solution is to provide interactive control, allowing the user to reveal further dimensionality by interacting with the plots using various widgets.

HoloViews provides support for all three of these approaches, via composable data structures that embed collections of `Element` objects in any arbitrarily dimensioned space. Fundamentally, this set of data structures (subclasses of `NdMapping`) are multi-dimensional dictionaries that allow the user to declare the dimensionality of the space via a list of key dimensions (`kdims`).

The list of supported `NdMapping` classes includes:

- `HoloMaps`: The most flexible high-dimensional data structure in HoloViews, allowing `Element` instances to be embedded in an arbitrarily high-dimensional space, to be rendered either as a video animation or as an interactive plot that allows exploration via a set of widgets.
- `GridSpaces`: A data structure for generating spatial layouts with either a single row (1D) or a two-dimensional grid. Each overall grid axis corresponds to a key dimension.
- `NdLayouts`/`NdOverlays`: Similar to `Layout` or `Overlay` objects, where the contained objects vary over one or more dimensions.

To explore a high-dimensional space of height as a function of age across different countries and years, you could declare `space=HoloMap(kdims=['Country', 'Year'])`. Now we can treat `space` as a dictionary and insert instances of classes such as `Curve` or `Scatter` with the appropriate `(country, year)` keys. For instance, the age and height `Curve` for the USA in 1988 (`usa`) can be inserted using `space['USA', 1988] = usa`. Note that the order of the indexing corresponds to the order of the declared key dimensions.

All of the above classes are simply different ways to package and view a high-dimensional dataset. Just as with `Elements`, it is

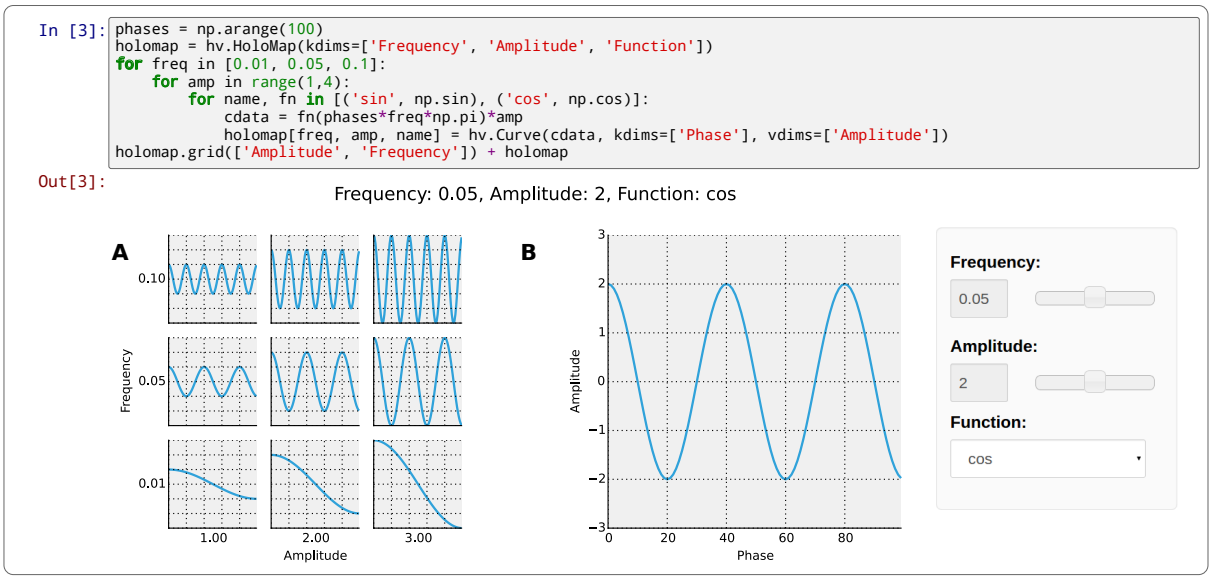


Fig. 2: *Example of a Layout object containing two different representations of a multi-dimensional space. Both representations contain* `Curve` *objects embedded in three dimensions (*`Frequency`*,* `Amplitude`*,* `Function`*), but not all of these dimensions can be visualized at once. In* ***A****, two of the dimensions are mapped onto the rows and columns of a grid, and the remaining* `Function` *dimension can be selected using the widget at the right. In* ***B****, only a single curve is shown, with the three sliders at the right together selecting the appropriate curve from the 3D HoloMap space. When two* `HoloMaps` *are joined in a* `Layout` *like this, it will automatically find the joint set of dimensions the HoloMaps can be varied over. In this way HoloMaps allow users to explore data naturally and conveniently even when its dimensionality exceeds what can be sensibly displayed on the screen at once.*

possible to cast between these different spaces via the constructor. In addition, they can all be tabularized into a HoloViews `Table` element or a pandas `DataFrame` [McK10], a feature that is also supported by the `Element` primitives.

To get a sense of how composing data and generating complex figures works within this framework, we explore some artificial data in Figure 2. Here we vary the frequency and amplitude of sine and cosine waves, demonstrating how we can quickly embed this data into a multi-dimensional space. First, we declare the dimensions of the space we want to explore as the key dimensions (`kdims`) of the HoloMap. Next, we populate the space iterating over the frequencies, amplitudes, and the two trigonometric functions, generating each `Curve` element individually and assigning to the HoloMap at the correct position in the space.

We can immediately go ahead and display this HoloMap either as an animation or using the default widgets, as in Figure 2 **B**. Visualizing individual curves in isolation is not very useful, of course; instead we probably want to see how the curves vary across `Frequency` and `Amplitude` in a single plot. A `GridSpace` provides such a representation and by using the space conversion method `.grid()` we can easily transform our three-dimensional HoloMap into a two-dimensional GridSpace (which then allows the remaining dimension, the choice of trigonometric function, to be varied via the drop-down menu). Finally, after composing a `Layout` together with the original `HoloMap`, we let the display system handle the plotting and rendering.

If we decide that a different representation of the data would be more appropriate, it is trivial to rearrange the dimensions without needing to write new plotting code. Even very high-dimensional spaces can be condensed into an individual plot or expressed as an interactive plot or animation, by simply specifying which part of the data we are interested in rather than writing new brittle and error-prone custom plotting code.

Customizing the visual representation

In this section we show how HoloViews achieves a total separation of concerns, keeping the composable data structures introduced above completely separate from both customization options and the plotting code. This design is much like the separation of content and presentation in HTML and CSS, and provides the same benefits of making the content easily maintainable while the presentation is easily controllable.

The only required connection between the above data structures and the custom display options is a single, automatically managed integer. Using this integer attribute we can make the data structures behave as if they were rich, stateful, and individually customizable objects, without actually storing anything to do with visualization on the objects. We will show how this separation is useful and extensible so that the user can quickly and easily customize almost every aspect of their plot. For instance, it is easy to change the font size of text, change the subfigure label format, change the output format (e.g. switch from PNG to SVG) and even alter the plotting backend (currently defaulting to Matplotlib) without changing any part of the underlying object being rendered.

Figure 3 provides an overall summary of how the different components in the display system interact. The declarative data structures define what will be plotted, specifying the arrangements of the plots, via Layouts, Overlays, and spaces. The connection between the data structure and the rendered representation is made according to the object type, the aforementioned integer attribute, and optionally specified group and label strings. By collecting the display options together and associating them with particular objects via these attributes, the visual representation of the content may be easily customized, e.g. to tweak aesthetic details such as tick marks, colors and normalization options. Once the user has specified both content and optionally customized the display the rendering system looks up the appropriate plot type for the object

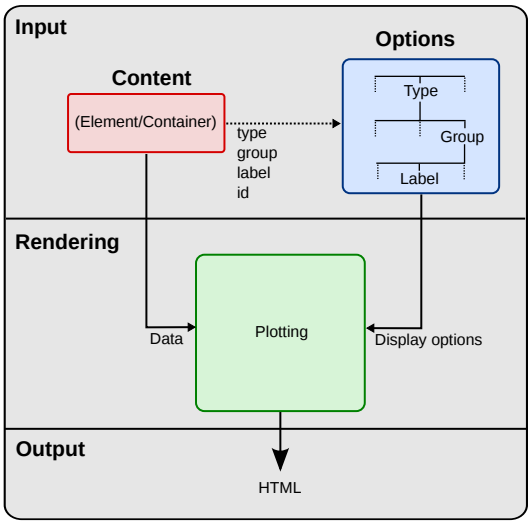


Fig. 3: *This view of the HoloViews display and customization systems illustrates the complete separation between the content (data) to be displayed, the display options, and the rendering/plotting system. The display options are stored entirely separately from the content as a tree structure, with the appropriate options being selected with user-controllable levels of specificity: general options for all objects of a given type, more specific options controlled by user-definable* `group` *and* `label` *strings, or arbitrarily specific options based on the integer* `id` *assigned to each content object. Plotting and rendering happens automatically through the use of IPython display formatters. These combine the content with the specified display options, call an external plotting library, which returns an HTML representation that can then be rendered in the notebook.*

in a global registry, which then processes the object and looks up the specified options in order to display it appropriately. This happens transparently without any input from the user. Once the plotting backend has rendered the plot in the appropriate format, it will be wrapped in HTML for display in the notebook.

The default display options are held on a global tree structure similar in structure to the composite trees described in the previous section, but with nodes holding custom display options in the form of arbitrary keywords. In fact, these option trees also use labels and groups the same way as composite trees except they additionally support type-specific customization. For instance, you may specify colormap options on the `Image` node of the tree that will then be applied to all `Images`. If this chosen colormap is not always suitable, you can declare that all `Image` elements belonging to a group (e.g. `group='Fractal'`) should use a different colormap by overriding it on the `Image.Fractal` node of the tree. This form of inheritance allow you to specify complex yet succinct style specifications, applying to all objects of a particular type or just to specific subsets of them.

To explore how option setting works in practice, Figure 4 shows an example of customizing Figure 1 with some basic display options. Here we use an optional but highly succinct method for setting the options, an IPython cell magic `%%opts`, to specify aspect ratios, line widths, colormaps, and sublabel formats. By printing the string representation of the content (`Out[6]`) and the options (`Out[7]`), we can see immediately that each entry in the options tree matches a corresponding object type. Finally, in the actual rendered output, we can see that all these display options have taken effect, even though the actual data structure differs from the object rendered in Figure 1 only by a single integer attribute.

A major benefit of separating data and customization options in this way is that all the options can be gathered in one place. There is no longer any need to dig deep into the documentation of a particular plotting package for a particular option, as all the options are easily accessible via a tab-completable IPython magic and are documented via the `help` function. This ease of discovery enables a workflow where the visualization details of a plot can be easily and quickly iteratively refined once the user has found data of interest.

The options system is also inherently extendable. New options may be added at any time, and will immediately become available for tab-completion. In fact, the plotting code for each element and container type may be switched out completely and independently, and the options system will automatically reflect the changes in the available customization options. This approach lets the user work with a variety of plotting backends at the same time, without even having to worry about the different plotting APIs.

The separation between content, options and plotting explicitly supports the workflows that are common in science, repeatedly switching between phases of exploration and periods of writing up. Interesting data can be collected and curated over time, where each step is instantly and transparently visualizable without any custom code cluttering up the notebook. Visualizations of data that are worth keeping can be customized through an interactive and iterative process, and the final set of plotting options can then be expressed as a single data structure separate from the actual displayed data, ready to be applied to the next batch of data from a subsequent measurement or experiment. Throughout, the scientist curates the data of interest, as revealed in associated visual representations, along with the visualization options and a separate codebase of general-purpose plots (mostly included in HoloViews, but potentially extended locally for specific domains). Each of these three aspects of the process (data, options, and code) can be developed, maintained, archived, and improved independently, providing comprehensive support for the natural process of exploration and dissemination common to all scientific disciplines.

Discussion

This paper demonstrates a succinct, flexible, and interactive approach for data exploration, analysis, and visualization. HoloViews restores the immediate feedback cycle that is characteristic of working with simple data in an interpreter. This is achieved by having declarative objects display themselves with good defaults allowing the user to immediately understand their data. In the majority of cases this eliminates the need to write plotting code and allows the user to keep a concise and reproducible recipe of their work, from exploration to the final publication. HoloViews thus allows scientists to capture the entire workflow involved in a research project.

Without a strictly enforced separation of concerns, workflow stages often end up mixing both data processing and visualization. Although a displayed representation is always necessary for understanding, it has been a dead end for further data processing. Because HoloViews objects represent themselves visually but also contain the raw data, the ability to continue processing is never terminated and exploration can continue. Furthermore, the chosen representation can easily be changed, turning what used to be a highly disjointed workflow into a open-ended process

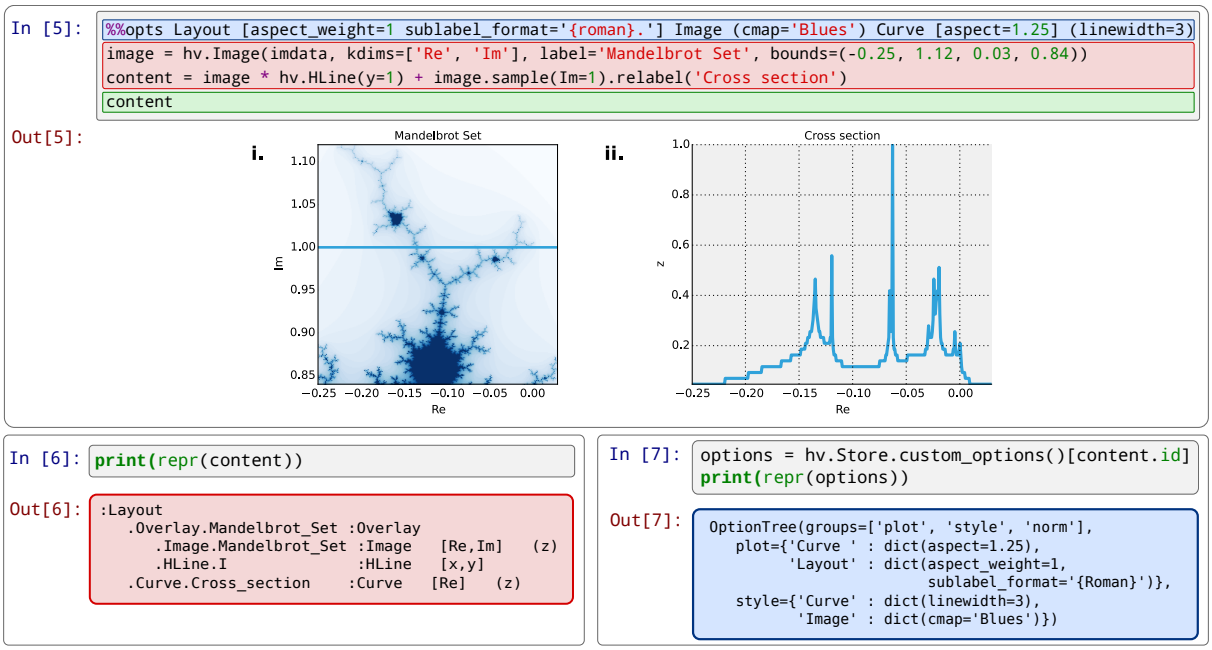


Fig. 4: *An example of customizing the display of Figure 1's data using the default Matplotlib backend.* `In[5]` *is color coded according to the components in Figure 3, where red is the content, blue is the display options (using an optional IPython-specific succinct syntax), and green is what triggers the the rendering.* `Out[5]` *shows how the supplied options have affected the final plots, compared to Figure 1. Finally,* `Out[6]` *and* `Out[7]` *show the textual representations of the content and the style specification respectively, demonstrating how the two are separate yet linked.*

concerned with the semantics of the data. Only once results worth disseminating are attained does it become necessary to consider the details of visualization.

The compositionality of HoloViews is superficially reminiscent of systems such as the Grammar of Graphics [Wil05] for the R language, but the aim of HoloViews is quite different. Instead of expressing all the complexities of graphics, the declarative data structures in HoloViews define a language for the semantics of the actual data. This language focuses on how the researcher conceptualizes it, *independent* of the exact details of plotting. The need for an automatic and useful visual representation is driven by the need to immediately present the data in a meaningful format.

HoloViews is one of many packages designed for working with large, multidimensional datasets, but it differs from each of these in important ways. For instance, Python's `seaborn` [Was14] and R's `ggplot2` [Wic09] library support laying out high-dimensional data into subplots and grids, while Python's Bokeh library and R's shiny [shiny] web application framework provide widgets for interactive data exploration. While each of these packages can provide extremely polished interactive graphics, getting them set up for specific sets of data requires significant additional effort and custom code, placing a barrier to their primary use case, the interactive exploration of data. HoloViews instead tries to avoid custom coding altogether as far as possible, with users instead supplying metadata to declare the properties of the data and option settings to control its visual appearance.

Although HoloViews is a general purpose library for working with data at every stage, it actually represents a significant advance over previous approaches focused only on achieving reproducibility of the final result. Simply by keeping specifications for figures succinct, HoloViews allows the entire recipe to be preserved in the notebook, not scattered over separately imported plotting code files. Secondly, because HoloViews can directly express the complex relationships between different bits of data as subfigures, it can capture entire figures within notebooks that would previously have required unreproducible work in external drawing programs. Lastly, HoloViews exports the actual data alongside published figures, allowing it to be tested automatically (as is done for the project web site) without conflating it with arbitrary display choices. HoloViews makes it possible to reproduce results from every step of the project, up to and including the final published figures, in a way that has not previously been practical.

Although HoloViews aims to provide good default behavior, scientific work often requires highly specialized visualizations. For that reason we have made it easy to extend the defaults and integrate new visualizations. Firstly, as many plotting and styling options as possible are exposed in an easily accessible manner, while providing a powerful, inheritance-based system for changing these options when required. Secondly, the options system has been designed to work well with the compositional data structures provided by HoloViews. Thirdly, HoloViews makes it trivial to add completely novel types of Elements with corresponding plots (or to override specific code in existing plots) using custom code when needed, and these custom plots will then combine seamlessly with other objects to make composite figures. Finally, not only is it possibly to implement new plot classes but entire plotting backends may be added and exposed to the user, such as the prototype Bokeh backend, which is well suited to live interaction and large datasets. Thus default plots are simple and straightforward, but even complex figures are easily achievable. Many such examples, ranging from simple to complex, can be found in the Tutorials and Examples sections of holoviews.org.

In this paper, we have focused on how a user can quickly build data structures for their content of interest. An even more powerful approach is for a developer to integrate HoloViews directly into a library, analysis tool, or simulator. By returning HoloViews

objects (which do not depend on any plotting library), any Python package can immediately have access to flexible, compositional data structures that automatically double as a visualization system. This is exactly the approach taken by the ImaGen image generation library and the Topographica neural simulator, two very different projects that both output data wrapped in HoloViews data structures.

Conclusion

Based on the key principles of: (1) making data immediately and transparently visualizable, (2) associating data directly with its semantic description, (3) keeping display option settings separate from the data, (4) keeping display code separate from both data and display options, (5) explicitly expressing the relationships between data elements compositionally, and (6) keeping the original data accessible even in complex visualizations, Holoviews supports the entire life cycle of scientific research, from initial exploration, to dissemination and publication, to eventual reproduction of the work and new extensions. Existing approaches for achieving some of these goals individually have been very limiting and only partially successful, each adding significant new costs along with the benefits they offer. HoloViews instead addresses the underlying problems fundamental to current methods for scientific research, solving seemingly intractable issues like reproducibility almost as a side effect of properly supporting the basic process of doing science.

Acknowledgments

This work was funded in part by grant 1R01-MH66991 to the University of Texas at Austin from the USA National Institute of Mental Health, by grant EP/F500385/1 from the UK EPSRC and MRC research councils, and by the Institute for Adaptive and Neural Computation at the University of Edinburgh.

References

[Cro13] Crook et al., "Learning from the Past: Approaches for Reproducibility in Computational Neuroscience", *20 Years of Computational Neuroscience*, J.M. Bower, ed., Springer, 9:73-102, 2013.

[Wol03] Stephen Wolfram, *The Mathematica Book*, Fifth Edition, Wolfram Media/Cambridge University Press, 2003.

[knitr] Foundation for Open Access Statistics, *knitr*, http://yihui.name/knitr, 2015.

[Per07] Fernando Perez and Brian E. Granger, IPython: a System for Interactive Scientific Computing, *Computing in Science and Engineering*, 9:21-19, 2007.

[Ste05] William Stein and David Joyner. SAGE: System for Algebra and Geometry Experimentation. *ACM SIGSAM Bulletin*, 39:61-64, 2005.

[Hun07] John D. Hunter, *Matplotlib: A 2D graphics environment*, Computing In Science & Engineering, 9(3):90-95, 2007.

[Wic09] Hadley Wickham, *ggplot2: elegant graphics for data analysis*, Springer New York, 2009.

[Was14] Michael Waskom et al.. *seaborn: v0.5.0*, Zenodo. 10.5281/zenodo.12710, November 2014.

[McK10] Wes McKinney, *Data Structures for Statistical Computing in Python*, Proceedings of the 9th Python in Science Conference, 51-56, 2010.

[Wil05] Leland Wilkinson, *The Grammar of Graphics*, Springer-Verlag New York, 2005.

[shiny] RStudio, Inc, *shiny: Easy web applications in R.*, http://shiny.rstudio.com, 2014.

Structural Cohesion: Visualization and Heuristics for Fast Computation with NetworkX and matplotlib

Jordi Torrents[§*], Fabrizio Ferraro[‡]

https://www.youtube.com/watch?v=K8RFIdG3g9Y

Abstract—The structural cohesion model is a powerful sociological conception of cohesion in social groups, but its diffusion in empirical literature has been hampered by computational problems. We present useful heuristics for computing structural cohesion that allow a speed-up of one order of magnitude over the algorithms currently available. Both the heuristics and the exact algorithm have been implemented on NetworkX by the first author. Using as examples three large collaboration networks (co-maintenance of Debian packages, co-authorship in Nuclear Theory, and co-authorship in High-Energy Theory) we illustrate our approach to measure structural cohesion in relatively large networks. We also introduce a novel graphical representation of the structural cohesion analysis to quickly spot differences across networks. It is implemented using matplotlib.

Index Terms—Network Analysis, Sociology, Structural Cohesion, NetworkX, matplotlib

Introduction

Group cohesion is a central concept that has a long and illustrious history in sociology and organization theory, although its precise characterization has remained elusive. Its use in most sociological research has been ambiguous at best. This is largely because, as [moody2003] argued, it is often based on sloppy definitions of cohesion, grounded mostly in intuition and common sense. Network analysis has provided a large number of solutions to this problem. From classical work in the graph-theoretic sociological tradition on cliques, clans, clubs, *k*-plexes, *k*-cores and lambda sets [wasserman1994], to the more recent contribution of physicists and computer scientists on community analysis [fortunato2010], network theorists have provided researchers with a wide range of measures of cohesion in social networks.

However, neither the classical approaches nor new developments in community analysis are well-enough suited to address many of the common uses of group cohesion in the sociological literature, and thus fall short when used in empirical analysis. The structural cohesion model ([white2001], [moody2003]) has strong mathematical foundations, and captures many of the features of the concept group cohesion. Despite this, it has not been widely used in empirical analysis because it is not possible to perform the required computations for networks with more than a few thousands nodes and edges in a reasonable time frame. Moreover, there are very few implementations available to researchers.

In this paper we present a set of heuristics to compute the connectivity structure of a given network. We implemented them, along with the exact algorithm, on top of NetworkX [hagberg2008], a Python Library for Network Analysis. We also suggest a novel graphical representation of the results, implemented using Matplotlib [hunter2007]. The rest of the paper is organized as follows: we start by discussing the main features which a cohesive subgroup formalization should have from a sociological perspective, and then discuss in depth the structural cohesion model. We then describe the exact algorithm and introduce our proposed heuristics. We go on to report our findings from applying the structural cohesion analysis to three large collaboration networks, which allows us to illustrate the novel graphical representation of the connectivity structure. Finally we conclude with some implications for future research.

Cohesion in social networks

[doreian1998] argue that group cohesion can be divided analytically into an *ideational* component, which is based on the members' identification with a collectivity, and a *relational* component, which is based on connections among members. These connections are, at least in part, observable, and thus the relational approach seems more appropriate for theory building and empirical research. But, despite its attractiveness, the relational component has received much less attention than the ideational component in sociological literature. Social network analysis has been the exception, and since the beginning, its proponents formalized group cohesion in relational terms, that is, they defined the boundaries of subgroups in a community starting from the patterns of relations among actors.

Unfortunately most of the existing formalizations of cohesive subgroups do not capture some key properties of the concept of cohesive groups. First, a cohesive subgroup should be *robust*, in the sense that its qualification as a group should not be dependent on the actions of a single individual, or any small set of individuals that belong to the group. This implies, on the one hand, that no actor, or small set of actors, should be able to dissolve the cohesive subgroup by abandoning it; while, on the other hand, all actors in a group should be related to all other actors by multiple direct or indirect connections in order to pull it together [moody2003]. Therefore, cohesive subgroups should also be relatively invariant to changes outside the group [brandes2005].

* Corresponding author: jordi.t21@gmail.com
§ University of Barcelona
‡ IESE Business School

Second, actual social groups tend to *overlap* in the sense that some actors are likely to be part of more than one cohesive subgroup. As [freeman1992] notes, formalizations of subgroups that overlap a lot are not well suited to capturing the concept of groups because their sociological use is not focused on individuals but on contexts, such as productive relations, friendship relations, or family ties, to name a few. Thus if groups are defined around a highly specific context the overlap is likely to be small. Therefore the formalization of subgroups often assumed non-overlapping subgroups. However, there is always overlap among cohesive subgroups in actual social groups; and this overlap might be both empirically and theoretically relevant.

Third, following a typical distinction in the social network literature, cohesive groups have both a *structural* and a *positional* dimension. In the former, cohesive subgroups are defined in terms of the global patterns of relations, and the focus is on the groups and the network as a whole. In the latter, the focus is on the identification of actors who, because of their network position, obtain preferential access to information or resources that flow through the network. Cohesive subgroup formalizations should help address both structural and positional questions.

Last but by no means least, cohesive subgroups are likely to display a *hierarchical structure* in the sense that highly cohesive subgroups are nested inside less cohesive ones. This notion of hierarchy is grounded on Simon's definition: *a system that is composed of interrelated subsystems, each of the latter being, in turn, hierarchic in structure until we reach some lowest level of elementary subsystem* [simon1962]. A hierarchical conception of cohesive subgroups implies that there is a relevant organization at all scales of the network, and that cohesive groups are a mesolevel structure that is not reducible to neither macro nor micro level phenomena and dynamics.

The structural cohesion model

Structural cohesion is a powerful explanatory factor for a wide variety of interesting empirical social phenomena. It can be used to explain, for instance: the likelihood of building alliances and partnerships among biotech firms [powell2005]; how positions in the connectivity structure of the Indian inter-organizational ownership network are associated with demographic features (age and industry); and differences in the extent to which firms engage in multiplex and high-value exchanges [mani2014]. Social cohesion can also help us understand degrees of school attachment and academic performance in young people, as well as the tendency of firms to enroll in similar political activity behaviors [moody2003]. It offers insight, also, into emerging trust relations among neighborhood residents or the hiring relations among top level US graduate programs [grannis2009]. In addition to social solidarity and group cohesion, the model can equally fit many relevant theoretical issues, such as conceptualizing structural differences among fields and organizations [white2004], explaining the role of highly connected subgroups in boosting diffusion in social networks without a high rate of decay [moody2004], or highlighting the complexity and diversity of the structure of real world markets beyond stylized one-dimensional characterizations of the market [mani2014].

The structural cohesion approach to subgroup cohesion ([white2001], [moody2003]) is grounded on two mathematically equivalent definitions of cohesion that are based on commonly used concepts of cohesion in the sociological literature. On the one hand, the ability of a collectivity to hold together independently of the will of any individual. As set out by the formal definition, *a group's structural cohesion is equal to the minimum number of actors who, if removed from the group, would disconnect the group*. Yet, on the other hand, a cohesive group has multiple independent relational paths among all pairs of members. According to the formal definition *a group's structural cohesion is equal to the minimum number of independent paths linking each pair of actors in the group* [moody2003]. These two definitions are mathematically equivalent in terms of the graph theoretic concept of node connectivity[1] as defined by Menger's Theorem [white2001], which can be formulated locally: *The minimum node cut set* $\kappa(u,v)$ *separating a nonadjacent* u,v *pair of nodes equals the maximum number of node-independent* $u-v$ *paths*; and globally: *A graph is k-connected if and only if any pair of nodes* u,v *is joined by at least k node-independent* $u-v$ *paths*. Thus Menger's theorem links with an equivalence relation the connectivity based on cut sets with the number of node independent paths among pairs of different nodes. This equivalence relation has a deep sociological meaning because it allows for the definition of structural cohesion in terms of the difficulty to pull a group apart by removing actors and, at the same time, in terms of multiple relations between actors that keep a group together.

The starting point of cohesion in a social group is a state where every actor can reach every other actor through at least one relational path. The emergence of a giant component --a large set of nodes in a network that have at least one path that links any two nodes-- is a minimal condition for the development of group cohesion and social solidarity. [moody2003] argue that, in this situation, the removal of only one node can affect the flow of knowledge, information and resources in a network because there is only one single path that links some parts of the network. Thus, if a network has actors who are articulation points[2], their role in keeping the network together is critical; and by extension the network can be disconnected by removing them. [moody2003] convincingly argue that biconnectivity provides a baseline threshold for strong structural cohesion in a network because its cohesion does not depend on the presence of any individual actor and the flow of information or resources does not need to pass through a single point to reach any part of the network. Therefore, the concept of robustness is at the core of the structural cohesion approach to subgroup cohesion.

Note that the bicomponent structure of a graph is an exact partition of its edges, which means that each edge belongs to one, and only one, bicomponent; but this is not the case for nodes because k-components can overlap in $k-1$ nodes. In the case of bicomponents, articulation points belong to all bicomponents that they separate. Thus, this formalization of subgroup cohesion allows limited horizontal overlapping over k-components of the same k. On the other hand, the k-component structure of a network is inherently hierarchical because k-components are nested in terms of connectivity: a connected graph can contain several 2-components, each of which can contain one or more tricomponents, and so forth.

However, one shortcoming of classifying cohesive subgroups only in terms of node connectivity is that k-components of the same k are always considered equally cohesive despite the fact

1. See http://networkx.readthedocs.org/en/latest/reference/generated/networkx.algorithms.connectivity.connectivity.node_connectivity.html .

2. See http://networkx.readthedocs.org/en/latest/reference/generated/networkx.algorithms.components.biconnected.articulation_points.html .

that one of them might be very close to the next connectivity level, while the other might barely qualify as a component of level k (i.e. removing a few edges could reduce the connectivity level to $k-1$). To deal with this shortcoming, we propose using another connectivity-based metric to obtain a continuous and more granular measure of cohesion. [beineke2002] propose the measure of *average node connectivity* of G[3], denoted $\bar{\kappa}(G)$, defined as the sum of local node connectivity between all pairs of different nodes of G divided by the number of distinct pairs of nodes. Or put more formally:

$$\bar{\kappa}(G) = \frac{\sum_{u,v} \kappa_G(u,v)}{\binom{n}{2}}$$

Where n is the number of nodes of G. In contrast to node connectivity κ, which is the minimum number of nodes whose removal disconnects some pairs of nodes, the average connectivity $\bar{\kappa}(G)$ is the expected minimal number of nodes that must be removed in order to disconnect an arbitrary pair of nodes of G. For any graph G it holds that $\bar{\kappa}(G) \geq \kappa(G)$. As [beineke2002] show, average connectivity does not increase only with the increase in the number of edges: graphs with the same number of nodes and edges, and the same degree for each node can have different average connectivity.

Despite all its merits, the structural cohesion model has not been widely applied to empirical analysis because it is not practical to compute it for networks with more than a few thousands nodes and edges due to its computational complexity. What's more, it is not implemented in most popular network analysis software packages. In the next section, we will review the existing algorithm to compute the k-component structure for a given network, before introducing our heuristics to speed up the computation.

Existing algorithms for computing k-component structure

[moody2003] provide an algorithm for identifying k-components in a network[4], which is based on the [kanevsky1993] algorithm for finding all minimum-size node cut-sets of a graph[5]; i.e. the set (or sets) of nodes of cardinality k that, if removed, would break the network into more connected components. The algorithm consists of 4 steps:

1) Identify the node connectivity, k, of the input graph using flow-based connectivity algorithms.
2) Identify all k-cutsets at the current level of connectivity using the Kanevsky's algorithm.
3) Generate new graph components based on the removal of these cutsets (nodes in the cutset belong to both sides of the induced cut).
4) If the graph is neither complete nor trivial, return to 1; otherwise end.

As the authors note, one of the main strengths of the structural cohesion approach is that it is theoretically applicable to both small and large groups, which contrasts with the historical focus of the literature on small groups when dealing with cohesion. But the fact that this concept and the algorithm proposed by the authors, are theoretically applicable to large groups does not mean that this would be a practical approach for analyzing the structural cohesion on large social networks.

The equivalence relation established by Menger's theorem between node cut sets and node independent paths can be useful to compute connectivity in practical cases but both measures are almost equally hard to compute if we want an exact solution. However, [white2001b] proposed a fast approximation algorithm for finding good lower bounds of the number of node independent paths between two nodes[6]. This algorithm is based on the idea of searching paths between two nodes, marking the nodes of the path as *used* and searching for more paths that do not include nodes already marked. But instead of trying all possible paths without order, this algorithm considers only the shortest paths: it finds node independent paths between two nodes by computing their shortest path, marking the nodes of the path found as *used* and then searching other shortest paths excluding the nodes marked as *used* until no more paths exist. Because finding the shortest paths is faster than finding other kinds of paths, this algorithm runs quite fast, but is not exact because a shortest path could use nodes that, if the path were longer, may belong to two different node independent paths [white2001b].

Heuristics for computing k-components and their average connectivity

The logic of the heuristics presented here is based on repeatedly applying fast algorithms for k-cores[7] [batagelj2011] and biconnected components[8] [tarjan1972] in order to narrow down the number of pairs of different nodes over which we have to compute their local node connectivity for building the auxiliary graph in which two nodes are linked if they have at least k node independent paths connecting them. We follow the classical insight that, *:math:'k'-cores can be regarded as seedbeds, within which we can expect highly cohesive subsets to be found* [seidman1983]. More formally, our approach is based on Whitney's theorem [white2001], which states an inclusion relation among node connectivity $\kappa(G)$, edge connectivity $\lambda(G)$ and minimum degree $\delta(G)$ for any graph G:

$$\kappa(G) \leq \lambda(G) \leq \delta(G)$$

This theorem implies that every k-component is nested inside a k-edge-component, which in turn, is contained in a k-core. This approach does not require computing node independent paths for all pairs of different nodes as a starting point, thus saving an important amount of computation. Moreover it does not require recursively applying the same procedure over each subgraph. In our approach we only have to compute node independent paths among pairs of different nodes in each biconnected part of each k-core, and repeat this procedure for each k from 3 to the maximal core number of a node in the input network.

The aim of the heuristics presented here is to provide a fast and reasonably accurate way of analyzing the cohesive structure of empirical networks of thousands of nodes and edges. As we

3. See http://networkx.readthedocs.org/en/latest/reference/generated/networkx.algorithms.connectivity.connectivity.average_node_connectivity.html .

4. See http://networkx.readthedocs.org/en/latest/reference/generated/networkx.algorithms.connectivity.kcomponents.k_components.html .

5. See http://networkx.readthedocs.org/en/latest/reference/generated/networkx.algorithms.connectivity.kcutsets.all_node_cuts.html .

6. See http://networkx.readthedocs.org/en/latest/reference/generated/networkx.algorithms.approximation.connectivity.node_connectivity.html .

7. See http://networkx.readthedocs.org/en/latest/reference/generated/networkx.algorithms.core.k_core.html .

8. See http://networkx.readthedocs.org/en/latest/reference/generated/networkx.algorithms.components.biconnected.biconnected_components.html .

have seen, k-components are the cornerstone of structural cohesion analysis. But they are very expensive to compute. Our approach consists of computing extra-cohesive blocks of level k for each biconnected component of a k-core. Extra-cohesive blocks are a relaxation of the k-component concept in which not all node independent paths among pairs of different nodes have to run entirely inside the subgraph. Thus, there is no guarantee that an extra-cohesive block of level k actually has node connectivity k. We introduce an additional constraint to the extra-cohesive block concept in order to approximate k-components: our algorithm computes extra-cohesive blocks of level k that are also k-cores by themselves in G. Furthermore, extra-cohesive blocks maintain high requirements in terms of multiconnectivity and robustness, thus conserving the most interesting properties from a sociological perspective on the structure of social groups.

Combining this logic with three observations about the auxiliary graph H allows us to design a new algorithm[9] for finding extra-cohesive blocks in each biconnected component of a k-core, that can either be exact but slow ---using flow-based algorithms for local node connectivity [brandes2005] --- or fast and approximate, giving a lower bound with certificate of the composition and the connectivity of extra-cohesive blocks ---using [white2001b] approximation for local node connectivity. Once we have a fast way to compute extra-cohesive blocks, we can approximate k-components by imposing that the induced subgraph of the nodes that form an extra-cohesive block of G have to also be a k-core in G.

Let H be the auxiliary graph in which two nodes are linked if they have at least k node independent paths connecting them in each of the biconnected components of the core of level k of original graph G (for $k > 2$). The first observation is that complete subgraphs in H (H_{clique}) have a one to one correspondence with subgraphs of G in which each node is connected to every other node in the subgraph for at least k node independent paths. Thus, we have to search for cliques in H in order to discover extra-cohesive blocks in G.

The second observation is that an H_{clique} of order n is also a core of level $n-1$ (all nodes have core number $n-1$), and the degree of all nodes is also $n-1$. The auxiliary graph H is usually very dense, because we build a different H for each biconnected part of the core subgraph of level k of the input graph G. In this kind of network big clusters of almost fully connected nodes are very common. Thus, in order to search for cliques in H we can do the following:

1) For each core number value c_{value} in each biconnected component of H:
2) Build a subgraph $H_{candidate}$ of H induced by the nodes that have *exactly* core number c_{value}. Note that this is different than building a k-core, which is a subgraph induced by all nodes with core number *greater or equal than* c_{value}.
3) If $H_{candidate}$ has order $c_{value}+1$ then it is a clique and all nodes will have degree $n-1$. Return the clique and continue with the following candidate.
4) If this is not the case, then some nodes will have degree $< n-1$. Remove all nodes with minimum degree from $H_{candidate}$.
5) If the graph is trivial or empty, continue with the following candidate. Or otherwise recompute the core number for each node and go to 3.

Finally, the third observation is that if two k-components of different order overlap, the nodes that overlap belong to both cliques in H and will have core numbers equal to all other nodes in the bigger clique. Thus, we can account for possible overlap when building subgraphs $H_{candidate}$ (induced by the nodes that have *exactly* core number c_{value}) by also adding to the candidate subgraph the nodes in H that are connected to all nodes that have *exactly* core number c_{value}. Also, if we sort the subgraphs $H_{candidate}$ in reverse order (starting from the biggest), we can skip checking for possible overlap for the biggest.

Based on these three observations, our heuristics for approximating the cohesive structure of a network and the average connectivity of each individual block, consists of:

Let G be the input graph. Compute the core number of each node in G. For each k from 3 to the maximum core number build a k-core subgraph G_{k-core} with all nodes in G with core level $\geq k$.

For each biconnected component of G_{k-core}:

1) Compute local node connectivity $\kappa(u,v)$ between all pairs of different nodes. Optionally store the result for each pair. Either use a flow-based algorithm (exact but slow) or White and Newman's approximation for local node connectivity (approximate but a lot faster).
2) Build an auxiliary graph H with all nodes in this bicomponent of G_{k-core} with edges between two nodes if $\kappa(u,v) \geq k$. For each biconnected component of H:
3) Compute the core number of each node in $H_{bicomponent}$, sort the values in reverse order (biggest first), and for each value c_{value}:
    a) Build a subgraph $H_{candidate}$ induced by nodes with core number *exactly* equal to c_{value} plus nodes in H that are conected with all nodes with core number equal to c_{value}.
        i) If $H_{candidate}$ has order $c_{value}+1$ then it is a clique and all nodes will have degree $n-1$. Build a core subgraph $G_{candidate}$ of level k of G induced by all nodes in $H_{candidate}$ that have core number $\geq k$ in G.
        ii) If this is not the case, then some nodes will have degree $< n-1$. Remove all nodes with minimum degree from $H_{candidate}$. Build a core subgraph $G_{candidate}$ of level k of G induced by the remaining nodes of $H_{candidate}$ that have core number $\geq k$ in G.
            A) If the resultant graph is trivial or empty, continue with the following candidate.
            B) Else recompute the core number for each node in the new $H_{candidate}$ and go to (i).
    b) The nodes of each biconnected component of $G_{candidate}$ are assumed to be a k-component of the input graph if the number of nodes is greater than k.
    c) Compute the average connectivity of each detected k-component. Either use the value of

9. See http://networkx.readthedocs.org/en/latest/reference/generated/networkx.algorithms.approximation.kcomponents.k_components.html .

$\kappa(u,v)$ computed in step 1 or recalcualte $\kappa(u,v)$ in the induced subgraph of candidate nodes.

Notice that because our approach is based on computing node independent paths between pairs of different nodes, we are able to use these computations to calculate both the cohesive structure and the average node connectivity of each detected k-component. Of course, computing average connectivity comes with a cost: either more space to store $\kappa(u,v)$ in step 1, or more computation time in step 3.c if we did not store $\kappa(u,v)$. This is not possible when applying the exact algorithm for k-components proposed by [moody2003] because it is based on repeatedly finding k-cutsets and removing them, thus it does not consider node independent paths at all.

The output of these heuristics is an approximation to k-components based on extra-cohesive blocks. We find extra-cohesive blocks and not k-components because we only build the auxiliary graph H one time on each bicoennected component of a core subgraph of level k from the input graph G. Local node connectivity is computed in a subgraph that might be larger than the final $G_{candidate}$ and thus some node independent paths that shouldn't could end up being counted.

Accuracy can be improved by rebuilding H from the pairwise node connectivity in $G_{candidate}$ and following the remaining steps of the heuristics at the cost of slowing down the computation. There is a trade-off between speed and accuracy. After some tests we decided to compute H only once and lean towards the speed pole of the trade-off. Our goal is to have an usable procedure for analyzing networks of thousands of nodes and edges in which we have substantive interests. Following this goal, the use of [white2001b] approximation algorithm for local node connectivity in step 3.b is key. It is almost on order of magnitude faster than the exact flow-based algorithms. As usual, speed comes with a cost in accuracy: [white2001b] algorithm provides a strict lower bound for the local node connectivity. Thus, by using it we can miss an edge in H that should be there.

Our tests reveal that the use of [white2001b] approximation does indeed underestimate the order of some k-components, particularly in not very sparse networks. One approach to mitigate this problem is to relax the strict cohesion requirement of $H_{candidate}$ being a clique. Following the network literature on cliques, we can relax its cohesion requirements in terms of degree, coreness and density. We did some experiments and found that a good relaxation criteria is to set a density threshold of 0.95 for $H_{candidate}$.

Case study: Structural cohesion in collaboration networks

The structural cohesion model can be used to explain cooperation in different kinds of collaboration networks; for instance, co-authorship networks ([moody2004] , [white2004]) and collaboration among biotech firms [powell2005]. Most collaboration networks are modeled as bipartite graphs, in which nodes can be divided in two disjoin sets, and edges only connect nodes from opposite sets. In the case of co-authorship networks, one node set represents authors and the other papers. Each author has edges that link her to all papers she authored. The usual practice to deal with bipartite networks is focus the analysis only on unipartite projections. That is, a new network only with the nodes that represent authors from the original bipartite network, where two authors are linked by an edge if they co-authored a paper together.

However, recent literature on bipartite networks strongly suggests that it is necessary to analyze bipartite networks directly to get an accurate picture ([uzzi2007], [opsahl2011], [latapy2008]). We show that this is also the case for the k-component structure of collaboration networks. This kind of analysis has been conducted very rarely on bipartite networks, and only on very small ones [white2004]. Its limited diffusion can be readily explained by the fact that bipartite networks are usually quite a lot bigger than their unipartite counterparts, and the computational requirements, once again, stifled empirical research in this direction.

The heuristics for structural cohesion presented here allow us to analyze relatively large networks (up to tens of thousands of nodes and edges) quickly enough to be practical. To illustrate this we use data on collaboration among software developers in one organization (the Debian project) and scientists publishing papers in the arXiv.org electronic repository in two different scientific fields: High Energy Theory and Nuclear Theory. We built the Debian collaboration network by linking each software developer with the packages (i.e. programs) that she uploaded to the package repository of the Debian Operating System during a complete release cycle. We analyze the Debian Operating System version 5.0, codenamed *Lenny*, which was developed from April 8, 2007, to February 1, 2009. Scientific networks are built using all the papers uploaded to the arXiv.org preprint repository from January 1, 2006, to December 31, 2010, for High Energy Physics Theory and Nuclear Theory. In these networks each author is linked to the papers that she has authored during the time period analyzed. Unipartite projections consist of scientists linked together if they have co-authored a paper, and developers linked together if they have worked on the same program. Table 1 presents some details on those networks (which are available, see[10]).

In the remaining part of this section we perform two kinds of analysis to illustrate how the structural cohesion model can help us understand the structure and dynamics of collaboration networks. First, we present a tree representation of the k-component structure ---which is also named cohesive blocks structure in the literature ([white2001], [moody2003], [white2004], [mani2014])--- for our bipartite networks and their unipartite projections, both for actual networks and for their random counterparts. Finally, we present a novel graphic representation of the structural cohesion of a network, based on three-dimensional scatter plot, using average node connectivity as a fine-grained measure of cohesion of each k-component.

For the first analysis we do need to generate null models in order to discount the possibility that the observed structure of actual networks is just the result of randomly mixing papers and scientists or packages and developers. The null models used in this paper are based on a bipartite configuration model [newman2003], which consists of generating networks by randomly assigning papers/programs to scientists/developers but maintaining constant the distribution of papers per scientists and scientists by paper observed in the actual networks. For unipartite projections, we generated bipartite random networks, and then performed the unipartite projection.

10. You can download the networks used in this section in graphml format. Nodes have an attribute named *bipartite*, with values 0 and 1, which indicates the node set to which each node belongs. Note that this is the convention used in NetworkX's bipartite package (see https://networkx.github.io/documentation/latest/reference/algorithms.bipartite.html):

- Debian Lenny: http://dx.doi.org/10.6084/m9.figshare.1472938
- Nuclear Theory: http://dx.doi.org/10.6084/m9.figshare.1472940
- High Energy Theory: http://dx.doi.org/10.6084/m9.figshare.1472939

	Bipartite				Unipartite			
Network	# nodes	# edges	Av. degree	Time(s)	# nodes	# edges	Av. degree	Time(s)
Debian Lenny	13,121	20,220	3.08	1,105.2	1,383	5,216	7.54	204.7
High Energy (theory)	26,590	37,566	2.81	3,105.7	9,767	19,331	3.97	7,136.0
Nuclear Theory	10,371	15,969	3.08	1,205.2	4,827	14,488	6.00	3,934.1

TABLE 1: *Collaboration networks analyzed from science and from software development. See text for details on their content. Time refers to the execution of our heuristics on each network expressed in seconds.*

So let's start with the tree representation of the cohesive blocks structure. As proposed by [white2004], we can represent the k-component structure of a network by drawing a tree whose nodes are k-components; two nodes are linked if the k-component of higher level is nested inside the k-component of lower level (see pp. 1643, 1651 from [mani2014] for this kind of analysis on the Indian firm ownership network). This representation of the connectivity structure can be built during the run time of the exact algorithm. However, because our heuristics are based on finding node independent paths, we have to compute first the k-components hierarchy, and then construct the tree that represents the connectivity structure of the network.

Figures 1 (a) and 1 (c) show the connectivity structure of Nuclear Theory collaboration networks represented as a tree, the former for the bipartite network and the latter for the unipartite one. As we can see, both networks display non-trivial structure. The bipartite network has up to an 8-component, but most nodes are in k-components with $k < 6$. Up to $k = 3$ most nodes are in giant k-components, but for $k = \{4,5\}$ there are many k-components of similar order. Figure 1 (c), which corresponds to the unipartite projection, has a lot more connectivity levels. In this network, the maximum connectivity level is 46; the four long legs of the plot correspond to 4 cliques with 47, 31, 27 and 25 nodes. Notice that each one of these 4 cliques are already a separated k-component at $k = 7$ It is at this level of connectivity ($k = \{7,8\}$) where the giant k-components start to dissolve and many smaller k-components emerge.

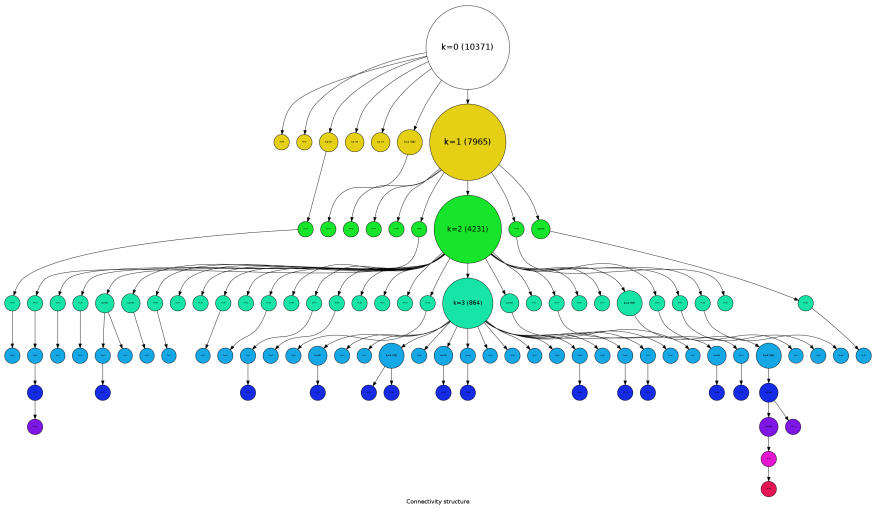
In order to be able to assess the significance of the results obtained, we have to compare the connectivity structure of actual networks with the connectivity structure of a random network that maintains some constraints observed in the empirical networks. In this case, we compare actual networks with only one random network. We obtained it by generating 1000 random networks and choosing one randomly. Figures 1 (b) and 1 (d) show the connectivity structure of the random counterparts for Nuclear Theory collaboration networks. For the bipartite network, instead of the differentiated connectivity structure displayed by the actual bipartite network, there is a flatter connectivity structure, where the higher level k-component is a tricomponent. Moreover, instead of many small k-components at high connectivity levels, the random bipartite network has only giant k-components where all nodes with component number k are. In this case, the unipartite network is also quite different from its random counterpart. There are only giant k-components up until $k = 15$, where the four cliques observed in the actual network separate from each other to form distinct k-components.

Going one step beyond classical structural cohesion analysis, as proposed above, we can deepen our analysis by also considering the average connectivity of the k-components of these networks. By analogy with the k-component number of each node, which is the maximum value k of the deepest k-component in which that node is embedded, we can establish the average k-component number of each node as the value of average connectivity of the deepest k-component in which that node is embedded. Notice that, unlike plain node connectivity, average node connectivity is a continuous measure of cohesion. Thus it provides a more granular measure of cohesion because we can rank k-components with the same k according to their average node connectivity.

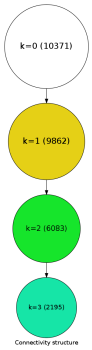
Figure 2 graphically represent the three networks with three-dimensional scatter plots produced with the powerful Matplotlib library [hunter2007]. In these graphs, each dot corresponds to a node of the network, for bipartite networks nodes represent both scientists/developers and papers/programs. The Z axis (the vertical one) is the average k-component number of each node, and the X and Y axis are the result of a 2 dimensional force-based layout algorithm implemented by the *neato* program of Graphviz. The two dimensional layout is computed by constructing a virtual physical model and then using an iterative solver procedure to obtain a low-energy configuration. Following [kamada1989], an ideal spring is placed between each pair of nodes (even if they are not connected in the network). The length of each spring corresponds to the geodesic distance between the pair of nodes that it links. The final node positioning in the layout approximates the path distance among pairs of nodes in the network.

This novel graphic representation of cohesion structure is inspired by the approximation technique developed by [moody2004] for plotting the approximate cohesion contour of large networks to which is not practical to apply Moody & White (2003) exact algorithm for k-components. Moody's technique is based on the fact that force-based layouts algorithms tend to draw nodes within highly cohesive subgroups near each other. Then we have to divide the surface of the two-dimensional plane in squares of equal areas and compute node independent paths on a sample of pairs of nodes inside each square so as to obtain an approximation for the node connectivity in that square. Then we can draw a surface plot using a smoothing probability density function. However, in order to obtain a nice smooth surface plot, we have to use heavy smoothing in the probability density function, and carefully choose the area of the squares (mostly by trial and error). Moreover, this technique strongly relies on the force-based layout algorithm to put nodes in highly cohesive subgroups near each other ---something which is not guaranteed because they are usually based in path distance and not directly on node connectivity. Because we are able to compute the k-component structure with our heuristics for large networks, the three-dimensional scatter plot only relies on the layout algorithm for setting the X and Y positions of the nodes, while the Z position (average node connectivity) is computed directly from the network. Moreover, we don't have to use a smoothed surface plot because we have a value of average connectivity for each node, and thus we can plot each node as a dot on the plot. This gives a more accurate picture of the actual cohesive structure of a network.

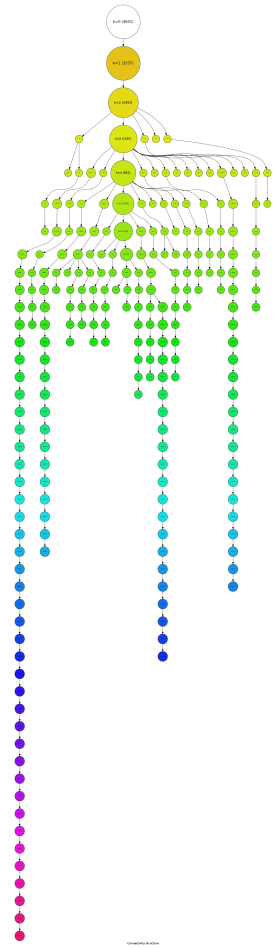
This representation of cohesive structures can help researchers visualize the presence of different organizational mechanisms in



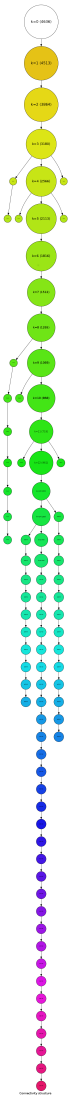
(a) Actual bipartite network



(b) Random bipartite network

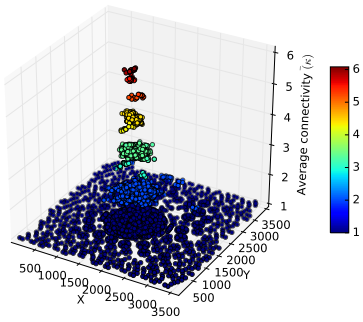


(c) Actual unipartite network

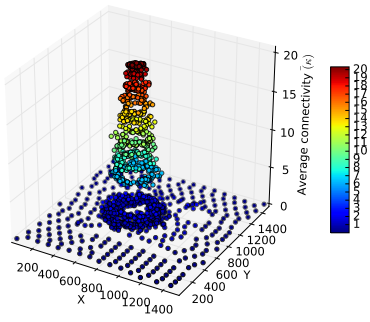


(d) Random unipartite network

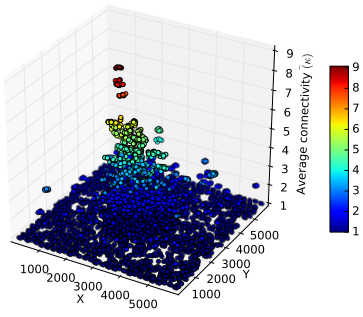
Fig. 1: *Cohesive blocks for bipartite and unipartite Nuclear Theory collaboration networks, and for their random counterparts. Random networks were generated using a bipartite configuration model. We built 1000 random networks and chose one randomly, see text for details. For lower connectivity levels we have removed some small k-components to improve the readability: we do not show 1-components with less than 20 nodes, 2-components with less than 15 nodes, or tricomponents with less than 10 nodes.*



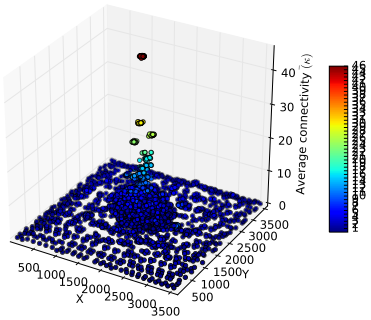
(a) Bipartite Debian Lenny network



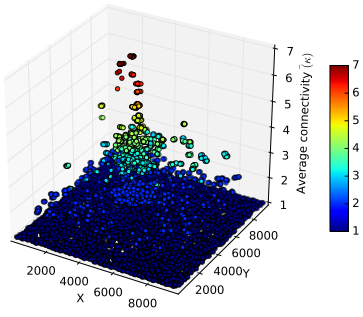
(b) Unipartite Debian Lenny network



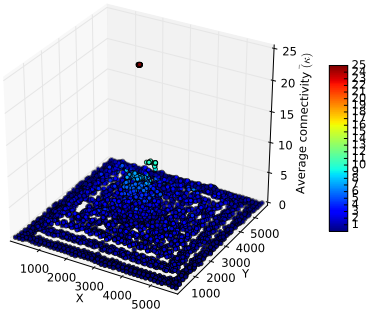
(c) Bipartite Nuclear Theory network



(d) Unipartite Nuclear Theory network



(e) Bipartite High Energy Theory network



(f) Bipartite High Energy Theory network

Fig. 2: *Average connectivity three-dimensional scatter plots. X and Y are the positions determined by the Kamada-Kawai layout algorithm. The vertical dimension is average connectivity. Each dot is a node of the network and bipartite networks contain both papers/programs and scientists/developers.*

different kinds of collaboration networks. The difference between the Debian and the scientific collaboration networks is striking. In figure 2 (a) we can see the scatter plot for a Debian bipartite network. We can observe a clear vertical separation among nodes in different connectivity levels. This is because almost all nodes in each connectivity level are in a giant k-component and thus they have the same average connectivity. In other words, developers in Debian show different levels of engagement and contribution, with a core group of developers deeply nested at the core of the community. This pattern is the result of formal and informal rules of collaboration that evolved over the years [ferraro2007] into a homogeneous hierarchical structure, where there is only one core of highly productive individuals at the center. Not surprisingly, perhaps, the Debian project has been particularly resilient to developers' turnover and splintering factions.

Scientific collaboration networks show a rather different structure of collaboration. The bipartite science collaboration networks (figures 2 (c) and 2 (e)) display a continuous hierarchical structure in which there are nodes at different levels of average connectivity for each discrete plain connectivity level. This is because science collaboration networks have a complex cohesive block structure where there are a lot of independent k-components in each plain connectivity level, for $k \geq 3$. Each small cohesive block has a different order, size and average connectivity; thus, when we display them in this three-dimensional scatter plot we observe a continuous hierarchical structure that contrasts with the almost discrete structure of Debian collaboration networks.

One explanation why we observe this heterogeneous connectivity structure is that scientific collaborations cluster around a variety of different aims, methods, projects, and institutional environments. Therefore as the most productive scientists collaborate with each other, hierarchies naturally emerge. However, we are less likely to observe one single hierarchical order as we did in the Debian network, as more than one core of highly productive scientists is likely to emerge.

If we compare the bipartite networks with their unipartite projections using this graphical representation (see figures 2 (b), 2 (d), and 2 (f)) we can see that, again, they look quite different. While bipartite average connectivity structure for the Debian network is characterized by clearly defined and almost discrete hierarchical levels, its unipartite counterpart shows a continuous hierarchical structure. However, this is not caused by the presence of many small k-components at the same level k, as in the case of bipartite science networks discussed above, but by the close succession of hierarchy levels with almost the same number of nodes in a chain-like structure.

For collaboration science networks, the three-dimensional scatter plots of unipartite projections are also quite different than their original bipartite networks. They have a lot more hierarchy levels than bipartite networks but most nodes are at lower connectivity levels. Only a few nodes are at top levels of connectivity, and they all form part of some clique, which are the groups in the long *legs* of the cohesive block structure depicted in figure 1 (c). Thus, the complex hierarchical connectivity structure of bipartite collaboration networks gets blurred when we perform unipartite projection. An important consequence of the projection is that only a few nodes embedded in big cliques appear at top connectivity levels and all other nodes are way down in the connectivity structure. This could lead the risk of overestimating the importance of those nodes in big cliques and to underestimate the importance of nodes that, despite being at high levels of the bipartite connectivity structure, appear only at lower levels of the unipartite connectivity structure.

Conclusions

We developed heuristics to compute the k-components structure, along with the average node connectivity for each k-component, based on the fast approximation to compute node independent paths [white2001b]. These heuristics allow for the computing of the approximate value of group cohesion for moderately large networks in a reasonable time frame. We showed that these heuristics can be applied to networks at least one order of magnitude bigger than the ones manageable by the exact algorithm proposed by [moody2003]. To ensure reproducibility and facilitate diffusion of these heuristics we provided an implementation of both the exact algorithm and the heuristics on top of NetworkX [hagberg2008]. These implementations are included in the recently released 1.10 version of NetworkX.

We analyzed three large collaboration networks and showed that the heuristics and the novel visualization technique for cohesive network structure help us capture important differences in the way collaboration is structured. Future research could leverage the tools we provide to systematically measure those structures. For instance, sociologists of science often compare scientific disciplines in terms of their collaborative structures [moody2004] and their level of controversies [bearman2010]. The measures and the visualization technique we proposed could nicely capture these features and compare them across scientific disciplines. This would make it possible to further our understanding of the social structure of science, and its impact in terms of productivity, novelty and impact. Social network researchers interested in organizational robustness would also benefit from leveraging the structural cohesion measures to detect sub-groups that are more critical to the organization's resilience, and thus prevent factionalization. Exploring the consequences of different forms of cohesive structures will eventually help us further our understanding of collaboration and the role that cohesive groups play in linking micro-level dynamics with macro-level social structures.

References

[batagelj2011] Batagelj, V. and M. Zaveršnik (2011). Fast algorithms for determining (generalized) core groups in social networks. Advances in Data Analysis and Classification 5(2), 129–145.

[bearman2010] Shwed, U. and P. Bearman (2010). The temporal structure of scientific consensus formation. American sociological review 75(6), 817–840.

[beineke2002] Beineke, L., O. Oellermann, and R. Pippert (2002). The average connectivity of a graph. Discrete mathematics 252(1-3), 31–45.

[brandes2005] Brandes, U. and T. Erlebach (2005). Network analysis: methodological foundations, Volume 3418. Springer Verlag.

[doreian1998] Doreian, P. and T. Fararo (1998). The problem of solidarity: theories and models. Routledge.

[freeman1992] Freeman, L. (1992). The sociological concept of "group": An empirical test of two models. American Journal of Sociology, 152–166.

[fortunato2010] Fortunato, S. (2010). Community detection in graphs. Physics Reports, 486(3), 75-174.

[grannis2009] Grannis, R. (2009). Paths and semipaths: reconceptualizing structural cohesion in terms of directed relations. Sociological Methodology 39(1), 117–150.

[hagberg2008] Hagberg, A., Schult, D. A., & Swart, P. (2008). Exploring network structure, dynamics, and function using NetworkX. In Proceedings of the 7th Python in Science Conferences (SciPy 2008) (Vol. 2008, pp. 11-16).

[hunter2007] Hunter, J. D. (2007). Matplotlib: A 2d graphics environment. Computing In Science & Engineering 9(3), 90–95.

[kamada1989] Kamada, T. and S. Kawai (1989). An algorithm for drawing general undirected graphs. Information processing letters 31(1), 7–15.

[kanevsky1993] Kanevsky, A. (1993). Finding all minimum-size separating vertex sets in a graph. Networks 23(6), 533–541.

[latapy2008] Latapy, M., C. Magnien, and N. Vecchio (2008). Basic notions for the analysis of large two mode networks. Social Networks 30(1), 31–48.

[mani2014] Mani, D. and J. Moody (2014). Moving beyond stylized economic network models: The hybrid world of the indian firm ownership network. American Journal of Sociology 119(6), pp. 1629–1669.

[moody2004] Moody, J. (2004). The structure of a social science collaboration network: Disciplinary cohesion from 1963 to 1999. American Sociological Review 69(2), 213–238.

[moody2003] Moody, J., & White, D. R. (2003). Structural cohesion and embeddedness: A hierarchical concept of social groups. American Sociological Review, 103-127.

[newman2003] Newman, M. (2003). The structure and function of complex networks. SIAM Review 45, 167.

[ferraro2007] O'Mahony, S. and F. Ferraro (2007). The emergence of governance in an open source community. The Academy of Management Journal 50(5), 1079–1106.

[opsahl2011] Opsahl, T. (2011). Triadic closure in two-mode networks: Redefining the global and local clustering coefficients. Social Networks 34.

[powell2005] Powell, W., D. White, K. Koput, and J. Owen-Smith (2005). Network dynamics and field evolution: The growth of interorganizational collaboration in the life sciences. American Journal of Sociology 110(4), 1132–1205.

[simon1962] Simon, H. A. (1962). The architecture of complexity. Proceedings of the American philosophical society 106(6), 467–482.

[seidman1983] Seidman, S. (1983). Network structure and minimum degree. Social networks 5(3), 269–287.

[tarjan1972] Tarjan, R. (1972). Depth-first search and linear graph algorithms. In Switching and Automata Theory, 1971., 12th Annual Symposium on, pp. 114–121. IEEE.

[uzzi2007] Uzzi, B., L. Amaral, and F. Reed-Tsochas (2007). Small-world networks and management science research: a review. European Management Review 4(2), 77–91.

[wasserman1994] Wasserman, S., & Faust, K. (1994). Social network analysis: Methods and applications (Vol. 8). Cambridge university press.

[white2004] White, D., J. Owen-Smith, J. Moody, and W. Powell (2004). Networks, fields and organizations: micro-dynamics, scale and cohesive embeddings. Computational & Mathematical Organization Theory 10(1), 95–117.

[white2001b] White, D. and M. Newman (2001). Fast approximation algorithms for finding node-independent paths in networks. Santa Fe Institute Working Papers Series.

[white2001] White, D. R., & Harary, F. (2001). The cohesiveness of blocks in social networks: Node connectivity and conditional density. Sociological Methodology, 31(1), 305-359.

Automated Image Quality Monitoring with IQMon

Josh Walawender‡*

https://www.youtube.com/watch?v=dGLkDOvYOHA

Abstract—Automated telescopes are capable of generating images more quickly than they can be inspected by a human, but detailed information on the performance of the telescope is valuable for monitoring and tuning of their operation. The IQMon (Image Quality Monitor) package[1] was developed to provide basic image quality metrics of automated telescopes in near real time.

Index Terms—astronomy, automated telescopes, image quality

Introduction

Using existing tools such as astropy [Astropy2013], astrometry.net [Lang2010], source extractor [Bertin1996], [Bertin2010a], SCAMP [Bertin2006], [Bertin2010b], and SWARP [Bertin2010c], IQMon analyzes images and provides the user with a quick way to determine whether the telescope is performing at the required level.

For projects which need to monitor the operation of an imaging telescope, IQMon is meant to provide a middle ground solution between simply examining the operations logs (e.g. those output by the control system) and a full data analysis pipeline. IQMon provides more information than typical operations logs while also giving a "ground truth" analysis since it looks at the actual data and not just what the system intended to do. While not as powerful as a full data pipeline, it is designed to provide operational information instead of scientific data products and thus its output is tuned to the task of examining the quality of the data and evaluating it for common problems.

IQMon can provide a determination of whether the telescope is focused (from the typical Full Width at Half Maximum, or FWHM, of stars in the image), whether it is pointing accurately (obtained from a comparison of the target coordinates with the astrometrically solved coordinates), whether the tracking or guiding is adequate (from the typical ellipticity of stars in the image), and whether the night is photometric (obtained from the typical photometric zero point of stars in the image). For wide field systems which detect many stars in each image, these metrics can be spatially resolved allowing for more detailed analysis such as differentiating between tracking error, focus error, and optical aberration or determining if the dome is partially obscuring the telescope aperture.

* *Corresponding author: joshwalawender@me.com*
‡ *Subaru Telescope, National Astronomical Observatory of Japan*

1. Source code at https://github.com/joshwalawender/IQMon

To date, IQMon has been deployed on three disparate optical systems. Two for the VYSOS Project which performs photometric monitoring of young stars: a 735mm focal length wide field imager with a monochrome CCD camera which undersamples the point spread function (PSF) and an 0.5 meter f/8 telescope with a monochrome CCD camera with well sampled PSF. It has also been deployed on the prototype unit for the PANOPTES[2] Project: an 85mm focal length camera lens and DSLR camera (with Bayer color array) designed for very wide field photometry. PANOPTES aims to create a global network of low-cost, robotic observatories for citizen science projects. IQMon has provided valuable diagnostic information about system performance in all cases.

Structure and Example Use

IQMon operates by using `Telescope` and `Image` classes. The `Telescope` object contains basic information about the telescope which took the data. When a `Telescope` object is instantiated, a configuration file is read which contains information on the telescope and controls various user-configurable parameters and preferences for IQMon. The configuration file is a YAML document and is read using the pyyaml[3] module.

An `Image` object is instantiated with a path to a file with one of the supported image formats and with a reference to a `Telescope` object. The image analysis process is simply a series of calls to methods on the `Image` object.

At the most basic level, IQMon is a sequencing tool which calls other programs (e.g. SExtractor, Astronometry.net) and tracks their output. These calls are all made using the `subprocess32` module, so all of these dependencies need to be installed and visible in the path for IQMon to function properly.

The IQMon philosophy is to never operate on the raw file itself, but instead to create a "working file" (using the `read_image` method) and store it in a temporary directory. If the raw image file is a FITS file, then `read_image` simply copies the raw file to the temporary directory and records this file name and path in the `working_file` property. If the file is a raw image file from a DSLR (e.g. `.CR2` or `.dng` format), then `read_image` will call `dcraw`[4] using the subprocess32 module[5] to convert the file to `.ppm`. The file is then converted to FITS format using either `pamtofits` or `pnmtofits` tools from the `netpbm`[6] package. IQMon then operates on the green channel of that resulting FITS file. For full functionality, the user should populate the header of this FITS file with appropriate FITS

2. http://projectpanoptes.org/v1/
3. http://pyyaml.org

keywords (e.g. RA, DEC, EXPTIME, DATE-OBS, etc.). To date, IQMon has only been tested with FITS and .CR2 files, but should in principle work with numerous DSLR raw format images.

IQMon has been tested with Python 2.7.X, testing with Python 3.X is pending. Python 3.X compatibility notes will be posted to the readme file on the git repository. IQMon runs successfully on both Mac OS X and linux. Windows compatibility is untested, but will be limited by the availability of dependencies (astrometry.net, SExtractor, etc.).

Because the system is designed to do quick evaluations of image quality, the primary concept is an object representing a **single** image. IQMon does not do any image stacking or other processing which would be applied to more than one image at a time nor is it built around other organizational concepts such as targets or visits. It is not intended to supplant a full data reduction and analysis package. The output of IQMon, however, can be stored in a MongoDB[7] database making it potentially useful for collecting information on observing concepts which span multiple images such as targets, nights, or visits. It might also be useful as a preprocessing step for a more complex data pipeline.

The time to process an image varies depending on many factors. It has been well studied for two of the systems mentioned in the Introduction. Both of these systems are analyzed by the same computer (a 2.3GHz Quad-Core Intel Core i7 with 8GB of RAM), so they share the system resources during the night.

In both cases the full image analysis takes tens of seconds per image, but depends on the number of stars in the image. The total analysis time for these systems is dominated by the SCAMP solve (roughly one third of the total time) and the generation of two JPEG images (also roughly one third of the total time). IQMon itself is single threaded, but many of the programs it calls, such as SCAMP, are multi threaded and so will take advantage of multiple cores.

In the following sections, I will describe a simple example of evaluating image quality for a single image. A more complex example which is updated in concert with IQMon can be found in the measure_image.py script at the git repository for the VYSOS project[8]. That process can then be wrapped in a simple program to monitor a directory for images and analyze them as they are written to disk (see the watch_directory.py script in the same VYSOS repository for an example). This enables automatic near real time analysis.

Configuration and Reading the Image In

After importing IQMon, the first step would be to instantiate the Telescope object which takes a configuration file as its input. The next step is to instantiate an Image object with the path to the image file and the Telescope object representing the telescope which took that image.

```
tel = IQMon.Telescope('~/MyTelescope.yaml')
im = IQMon.Image('~/MyImage.fits', tel)
```

IQMon writes a log which is intended to provide useful information to the user (not just the developer) and shows the progress of the analysis. We can either pass in a logger object from Python's logging module, or ask IQMon to create one:

```
# create a new logger object
im.make_logger(verbose=False)
print('Logging to file {}'.format(im.logfile))
im.logger.info('This is a log entry')
```

The first step for any image analysis is likely to be to call the read_image method. After calling read_image, the FITS header is read and various Image object properties are populated by calling the read_header method.

```
# Generate working file copy of the raw image
im.read_image()
# Read the fits header
im.read_header()
```

Once the image has been read in and a working file created, IQMon uses various third party tools to perform image analysis. The following sections describe some of the analysis steps which are available.

PSF Size Measurements with Source Extractor

Source Extractor (SExtractor) [Bertin1996], [Bertin2010a] is a program which builds a catalog of sources (stars, galaxies, etc.) detected in an image. SExtractor is called using the run_SExtractor method which invokes the command using the subprocess32 module. Customization parameters can be passed to Source Extractor using the telescope configuration file.

The output file of SExtractor is read in and stored as an astropy table object. Stars with SExtractor generated flags are removed from the table and the table is stored as a property of the image object.

Determining the PSF size from the SExtractor results is done with the determine_FWHM method. The full width at half maximum (FWHM) and ellipticity values for the image are a weighted average of the FWHM and ellipticity values for the individual stars.

These steps not only provide the typical FWHM (which can indicate if the image is in focus), they can also be used to guess at whether the image is "blank" (i.e. very few stars are visible either because of cloud cover or other system failure). For example:

```
im.run_SExtractor()
# Consider the image to be blank if <10 stars
if im.n_stars_SExtracted < 10:
    im.logger.warning('Only {} stars found.'
                      .format(im.n_stars_SExtracted))
    im.logger.warning('Image may be blank.')
else:
    im.determine_FWHM()
```

Pointing Determination and Pointing Error

IQMon also contains a solve_astrometry method to invoke the solve-field command which is part of the astrometry.net software. The call to solve-field is only intended to determine basic pointing and orientation and so IQMon does not use the SIP polynomial fit of distortion in the image.

Once a world coordinate system (WCS) is present in the image header, then the determine_pointing_error method can be called which compares the right ascension (RA) and declination (DEC) values read from the RA and DEC keywords in the header (which are presumed to be the telescope's intended pointing) to the RA and DEC values of the center pixel which are calculated using the astropy.wcs module. The separation between the two

4. http://www.cybercom.net/~dcoffin/dcraw/

5. The subprocess32 module "is a backport of the subprocess standard library module from Python 3.2 & 3.3 for use on Python 2.4, 2.5, 2.6 and 2.7" (from https://pypi.python.org/pypi/subprocess32). It is used instead of the standard subprocess module due to its support for timeout functionality.

6. http://netpbm.sourceforge.net

7. http://www.mongodb.org

8. https://github.com/joshwalawender/VYSOStools

coordinates is determined using the `separation` method available in the `SkyCoord` object of the `astropy.coordinates` module. The magnitude of the separation between the two is reported as the pointing error.

```
# If WCS is not present, solve with astrometry.net,
if not im.image_WCS:
    im.solve_astrometry()
# Determine pointing error by comparing telescope
# pointing coordinates from the header with WCS.
im.determine_pointing_error()
```

Astrometric Distortion Correction

In order to make an accurate comparison of the photometry of stars detected in the image and stars present in a chosen stellar catalog, many optical systems require distortion coefficients to be fitted as part of the astrometric solution. IQMon uses the SCAMP software to fit distortions.

SCAMP is invoked with the `run_SCAMP` method. Once a SCAMP solution has been determined, the image can be remapped to new pixels without distortions using the SWARP tool with the `run_SWARP` method.

```
# If the image has a WCS and a SExtractor catalog,
# run SCAMP to determine a WCS with distortions.
if im.image_WCS and im.SExtractor_results:
    im.run_SCAMP()
    if im.SCAMP_successful:
        # Remap the pixels to a rectilinear grid
        im.run_SWarp()
```

A Note on Astrometry.net and SCAMP

In principle, Astrometry.net can solve for distortions. The `-t` option on `solve-field` allows the user to specify the order of the SIP polynomial which the program should fit. This is available in IQMon by calling the `solve_astrometry` method with the `SIP` keyword set to the polynomial order to pass to `solve-field`.

In my experience working with the first two systems IQMon was used on, I found that high order solves were not necessarily reliable or timely. The `solve-field` operation would sometimes fail to solve or would process for a very long time which would cause the analysis system to fail to keep up with the data rate from the two telescopes.

This is why SCAMP is also available in IQMon and is the recommended astrometric solution if you want full distortion correction. By defining a SCAMP "ahead" file, you can incorporate previous knowledge of the optical system's distortion characteristics rather then solving blindly. With a proper ahead file, SCAMP was a more reliable solution.

SWarp is used because (at the time) `astropy.wcs` did not handle the distortion coefficients as written by SCAMP. To solve this, SWarp remaps the pixels to de-distort the image which means that the WCS is properly described by a very basic set of header keywords (`CRPIXn`, `CRVALn`, `PCn_m`, etc.) which almost every analysis program supports.

Estimating the Photometric Zero Point

With a full astrometric solution, SExtractor photometry, and a catalog of stellar magnitude values, we can estimate the zero point for the image and use that as an indicator of clouds or other aperture obscurations.

The `get_catalog` method can be used to download a catalog of stars from VizieR using the `astroquery`[9] module. Alternatively, support for a local copy of the UCAC4 catalog is available using the `get_local_UCAC4` method.

Once a catalog is obtained, the `run_SExtractor` method is invoked again, this time with the `assoc` keyword set to `True`. This will limit the resulting catalog of detected stars to stars which **both** exist in the catalog and also are detected in the image. This may significantly decrease the number of stars used for the FWHM and ellipticity calculation, but may also remove spurious detections of image artifacts which would improve the reliability of the measured values.

```
# Retrieve catalog defined in config file
im.get_catalog()
im.run_SExtractor(assoc=True)
im.determine_FWHM()
im.measure_zero_point()
```

In the above example code, `determine_FWHM` is invoked again in order to use the new SExtractor catalog for the calculation.

The `measure_zero_point` method determines the zero point by taking the weighted average of the difference between the measured instrumental magnitude from SExtractor and the catalog magnitude in the same filter.

It should be noted that unless custom code is added to handle reduction steps such as dark/bias subtraction and flat fielding, the zero point result will be influenced by systematics due to those effects. In addition, the choice of catalog and the relative response curve of the filter in use and the filter defined by the catalog's photometric system will also introduce systematic offsets. For many systems (especially typical visible light CCDs), the zero point value from IQMon can be used to compare throughput from image to image, but should not be used to compare different equipment configurations.

Analysis Results and Mongo Database Integration

Results of the IQMon measurements for each image are stored as properties of the `Image` object as `astropy.units.Quantity`. For example, the `FWHM` value is in units of pixels, but can be converted to arcseconds using the equivalency which is automatically defined by the `Telescope` object (`tel.pixel_scale_equivalency`) for this purpose.

```
## Results are typically astropy.units quantities
## and can be manipulated as such.  For example:
print('Image FWHM = {:.1f}'.format(im.FWHM))
print('Image FWHM = {:.1f}'.format(\
      im.FWHM.to(u.arcsec, equivalencies=\
      im.tel.pixel_scale_equivalency)))
print('Zero Point = {:.2f}'.format(im.zero_point))
print('Pointing Error = {:.1f}'.format(\
      im.pointing_error.to(u.arcmin)))
```

These results can also be stored for later use. Methods exist to write them to an `astropy.Table` (the `add_summary_entry` method) and to a YAML document (the `add_yaml_entry` method), but the preferred storage solution is to use a mongo database as that is compatible with the `tornado` web application included with IQMon (see below).

The address, port number, database name, and collection name to use with `pyMongo` to add the results to an existing mongo database are set by the Telescope configuration file. The `add_mongo_entry` method adds a dictionary of values with the results of the IQMon analysis.

9. http://dx.doi.org/10.6084/m9.figshare.805208

Flags

For the four primary measurements (FWHM, ellipticity, pointing error, and zero point), the configuration file may contain a threshold value. If the measured value exceeds the threshold (or is below the threshold in the case of zero point), then the image is "flagged" as an indication that there may be a potential problem with the data. The flags property of an `Image` object stores a dictionary with the flag name and a boolean value as the dictionary elements.

This can be useful when summarizing results. For example, the Tornado web application provided with IQMon (see the Tornado Web Application section) lists images and will color code a field red if that field is flagged. In this way, a user can easily see when and where problems might have occurred.

Images and Plots

In addition to generating single values for FWHM, ellipticity, and zero point to represent the image, IQMon can also generate more detailed plots with additional information.

A plot with PSF quality information can be generated when `determine_FWHM` is called by setting the `plot=True` keyword. This generates a .png file (see Fig. 1) using matplotlib [matplotlib] which shows detailed information about the point spread function (FWHM and ellipticity metrics) including histograms of individual values, a spatial map of FWHM and ellipticity over the image, and plots showing the ellipticity vs. radius within the image (which can be used to show whether off axis aberrations influence the ellipticity measure) and the correlation between the measured PSF position angle and the position angle of the star within the image (which can be used to differentiate between tracking error and off axis aberrations).

In the example plot (Fig. 1), we can see several different effects. First, from the spatial distribution of FWHM and ellipticity, as well as the ellipticity vs. radius plot, we see that image quality is falling off at large radii. This image is from a wide field imaging system and we are seeing the signature of off axis aberrations. This is also suggested in the plot of the correlation between the measured PSF position angle and the position angle of the star within the image which shows strong diagonal components indicating that position within the image influences the PSF. There is also, however, a vertical component in that plot at $PA \sim 0$ which is suggestive of image drift perhaps due to slight polar misalignment or flexure.

A plot with additional information on the zero point can be generated when calling `measure_zero_point` by setting the `plot` keyword to `True`. This generates a .png file (see Fig. 2) using matplotlib which shows plots of instrumental magnitude vs. catalog magnitude, a histogram of zero point values, a plot of magnitude residuals vs. catalog magnitude, and a a spatial map of zero point over the image.

JPEG versions of the image can be generated using the `make_JPEG` method. The jpeg can be binned or cropped using the `binning` or `crop` keyword arguments and various overlays can be generated showing, for example, the pointing error and detected and catalog stars.

The JPEG overlays can be useful in evaluating the performance of SExtractor and SCAMP. In the example shown in Fig. 3, the stars marked as detected by SExtractor (which was run with the `assoc` keyword set to `True`) show that there are no stars detected in the very corners of the image. This indicates that the SCAMP distortion solution did not accurately fit the WCS in the corners and could be improved. Poor SCAMP solutions can also show up even more dramatically when entire radial zones of the image have no matched stars.

Tornado Web Application

IQMon comes with a tornado web application which, while it can be run stand alone, is intended to be used as a template for adding IQMon results to a more customized web page. The web application (`web_server.py`) contains two `tornado` web handlers: `ListOfNights` and `ListOfImages`. The first generates a page which lists UT dates and if there are image results associated with a date, then it provides a link to a page with the list of image results for that date. The second handler (see Fig. 4) produces the page which lists the images for a particular UT date (or target name) and provides a table formatted list of the IQMon measurement results for each image with flagged values color coded red, along with links to jpegs and plots generated for that image.

This web application is intended to be the primary interface for users. It provides three levels of interaction to the user. First, a custom plot of IQMon results over the course of a night is easy to generate from the mongo database entries and represents the highest level of interaction. Using such a plot, serious problems which affect many images can be detected at a glance. Users can then drill down to see a list of images for that UT date and see system performance as a table of IQMon results with flagged values highlighted in red. Finally an individual image can be examined as a jpeg with overlays or by using the PSF quality plots or zero point plots to examine detailed performance.

Conclusions

IQMon provides a way to evaluate the performance of automated telescopes. It allows the user to build a customized analysis for their particular application by assembling a script which includes only those steps which are required. Using the included tornado web application, a user can quickly and easily view the results and determine whether the observatory is performing acceptably or if it needs attention.

Over roughly two years of routine operation with two telescopes, it has enabled quick alerting of problems including stuck focus drives, poorly aligned dome rotation, and poor tracking model correction. Previously, some of these problems would have gone unnoticed until a spot check of the data downloaded from the site revealed them or they would have required a time consuming reading of the nightly system logs to reveal. Use of IQMon has resulted in greater uptime and improved data quality for both telescopes.

References

[Astropy2013] Astropy Collaboration, Robitaille, T.~P., Tollerud, E.~J., et al. *Astropy: A community Python package for astronomy* 2013, A&A, 558, A33

[Bertin1996] Bertin, E., & Arnouts, S. *SExtractor: Software for source extraction*, 1996, A&AS, 117, 393

[Bertin2006] Bertin, E. *Automatic Astrometric and Photometric Calibration with SCAMP*, 2006, Astronomical Data Analysis Software and Systems XV, 351, 112

[Bertin2010b] Bertin, E. *SCAMP: Automatic Astrometric and Photometric Calibration*, 2010, Astrophysics Source Code Library, 1010.063

[Bertin2010a] Bertin, E., & Arnouts, S. *SExtractor: Source Extractor*, 2010, Astrophysics Source Code Library, 1010.064

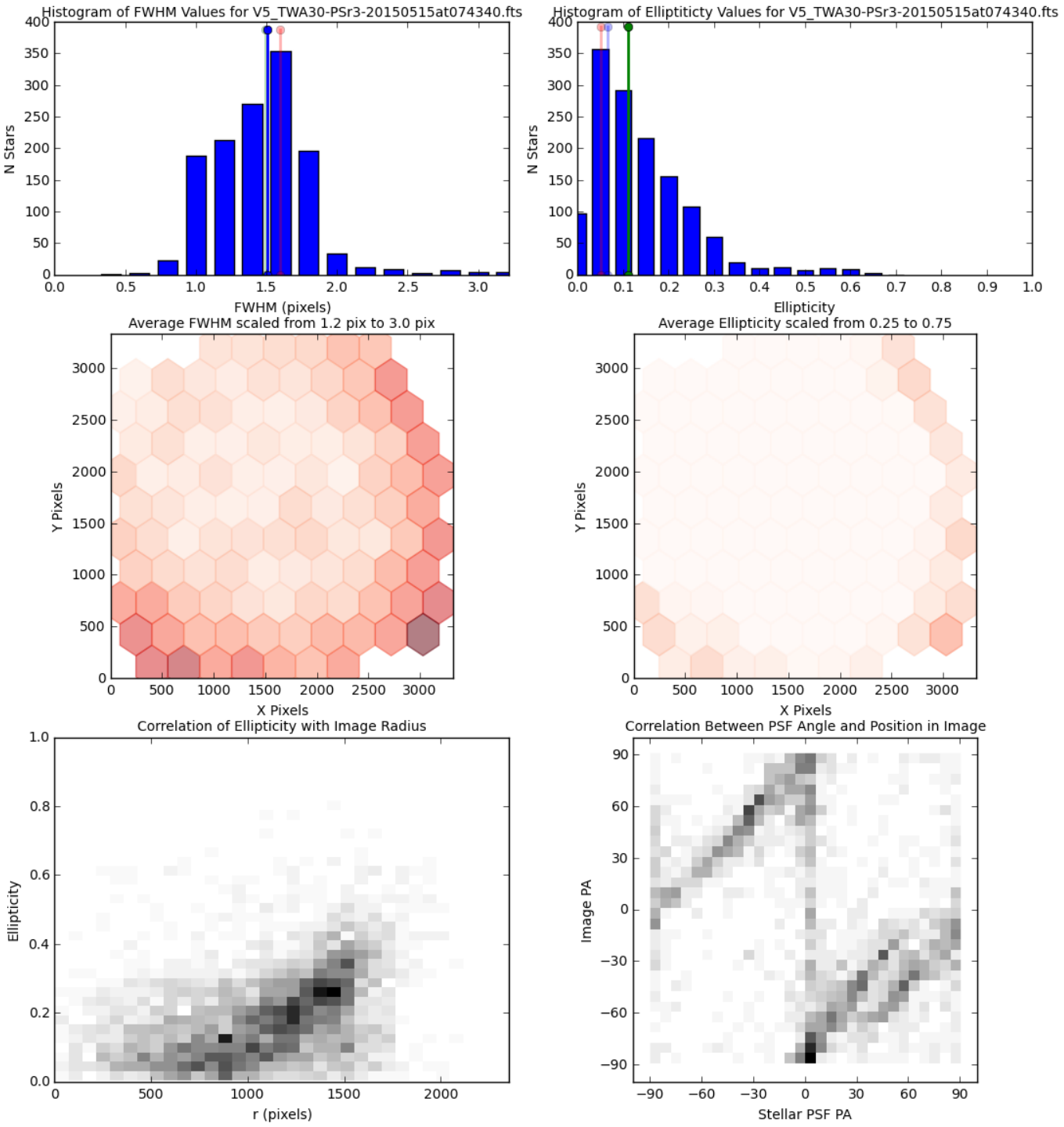


Fig. 1: *An example of the plot which can be produced using the* `determine_FWHM` *method. The plot shows histograms of the FWHM and ellipticity values (upper left and upper right respectively), the spatial distribution of FWHM and ellipticity values (middle left and middle right), ellipticity vs. radius within the image (lower left), and the correlation between the measured PSF position angle and the position angle of the star within the image (lower right).*

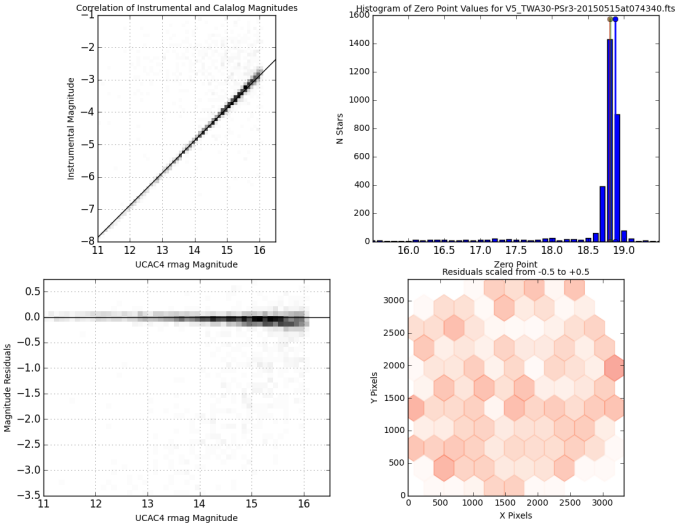


Fig. 2: *An example of the plot which can be produced using the* `measure_zero_point` *method. The plot shows the correlation between instrumental magnitude and catalog magnitude (upper left), a histogram of zero point values (upper right), a plot of the residuals vs. catalog magnitude (lower left), and a spatial distribution of the residuals (lower left).*

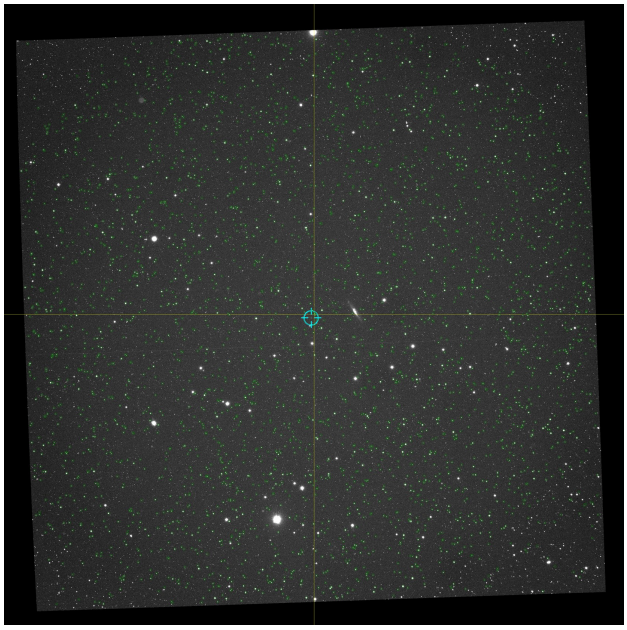


Fig. 3: *An example jpeg generated by the* `make_JPEG` *method using the* `mark_detected_stars` *and* `mark_pointing` *options. In this example, pointing error has placed the target (marked by the cyan crosshair) to the lower right (southwest) of the image center (marked by the yellow lines). Stars from the UCAC4 catalog which were detected in the image are marked with green circles.*

[Bertin2010c] Bertin, E. *SWarp: Resampling and Co-adding FITS Images Together* 2010, Astrophysics Source Code Library, 1010.068

[Lang2010] Lang, D., Hogg, D. W., Mierle, K., Blanton, M., & Roweis, S., *Astrometry.net: Blind astrometric calibration of arbitrary astronomical images* 2010, AJ, 137, 1782–1800

[matplotlib] Hunter, J. D., *Matplotlib: A 2D graphics environment* 2007, Computing In Science & Engineering, 9, 90-95

IQMon Results for 20150523UT for VYSOS-5

Exposure Start (Date and Time UT)	Image File Name	Alt (deg)	Az (deg)	Airmass	Moon Sep (deg)	Moon Illum. (%)	FWHM (pix)	Ellip.	Pointing Error (arcmin)	Zero Point (mag)	N Stars	Process Time (sec)
20150523UT 05:56:00	V5_TWA30-PSr3-20150523at055554.fts (JPEG) (PSF) (ZP)	40.1	180.8	1.56	64.8	27 %	3.62	0.61	0.39	16.03	321	63 s
20150523UT 05:58:19	V5_TWA30-PSr3-20150523at055812.fts (JPEG) (PSF) (ZP)	40.0	181.4	1.56	64.8	27 %	2.67	0.61	0.43	16.76	739	48 s
20150523UT 06:00:40	V5_TWA30-PSr3-20150523at060033.fts (JPEG) (PSF) (ZP)	40.0	182.1	1.56	64.8	27 %	2.32	0.56	0.64	17.28	536	52 s
20150523UT 06:17:30	V5_TWA30-PSr3-20150523at061724.fts (JPEG) (PSF) (ZP)	39.7	186.8	1.57	64.7	27 %	2.21	0.11	0.31	18.53	662	56 s
20150523UT 06:19:49	V5_TWA30-PSr3-20150523at061943.fts (JPEG) (PSF) (ZP)	39.7	187.5	1.57	64.6	27 %	2.10	0.10	0.49	18.66	647	55 s
20150523UT 06:22:09	V5_TWA30-PSr3-20150523at062202.fts (JPEG) (PSF) (ZP)	39.6	188.1	1.57	64.6	27 %	1.98	0.06	0.67	18.72	642	59 s
20150523UT 06:27:52	V5_TWA30-PSr3-20150523at062746.fts (JPEG) (PSF) (ZP)	39.4	189.7	1.58	64.6	27 %	1.73	0.09	2.85	18.84	630	57 s
20150523UT 06:30:18	V5_TWA30-PSr3-20150523at063011.fts (JPEG) (PSF) (ZP)	39.3	190.4	1.58	64.6	27 %	1.76	0.16	2.58	18.87	634	60 s
20150523UT 06:32:43	V5_TWA30-PSr3-20150523at063237.fts (JPEG) (PSF) (ZP)	39.2	191.0	1.59	64.6	27 %	1.91	0.10	0.31	18.88	684	59 s
20150523UT 06:36:08	V5_TWA30-PSr3-20150523at063602.fts (JPEG) (PSF) (ZP)	39.0	191.9	1.59	64.5	27 %	1.43	0.04	2.82	18.91	656	57 s
20150523UT 06:38:34	V5_TWA30-PSr3-20150523at063827.fts (JPEG) (PSF) (ZP)	38.9	192.6	1.60	64.5	27 %	1.56	0.08	0.47	18.89	704	58 s
20150523UT 06:40:53	V5_TWA30-PSr3-20150523at064047.fts (JPEG) (PSF) (ZP)	38.8	193.2	1.60	64.5	27 %	1.35	0.07	0.72	18.89	688	60 s
20150523UT 06:43:37	V5_TWA30-PSr3-20150523at064331.fts (JPEG) (PSF) (ZP)	38.6	194.0	1.61	64.5	27 %	1.67	0.08	1.00	18.88	693	58 s
20150523UT 06:45:57	V5_TWA30-PSr3-20150523at064551.fts (JPEG) (PSF) (ZP)	38.5	194.6	1.61	64.5	27 %	1.51	0.07	1.04	18.88	691	63 s
20150523UT 06:48:16	V5_TWA30-PSr3-20150523at064809.fts (JPEG) (PSF) (ZP)	38.3	195.2	1.62	64.5	27 %	1.83	0.11	1.29	18.88	693	67 s
20150523UT 06:51:00	V5_TWA30-PSr3-20150523at065054.fts (JPEG) (PSF) (ZP)	38.2	195.9	1.62	64.4	27 %	1.50	0.05	1.56	18.90	691	62 s

Fig. 4: *An example of the* `ListOfImages` *handler of the tornado web application. In this example, a user can easily determine that the first few images of the night had a problem (indicated by the red flagged values). Based on examination of the JPEGs, this turns out to have been due to the dome rotation being misaligned and partially blocking the telescope aperture leading to large FWHM and ellipticity values (image elongation due to "glints" of the dome edge) and low zero point values (due to aperture obscuration). The problem resolved itself without human intervention as can be seen by the green, un-flagged images which follow and which continued for the rest of the night.*

PyRK: A Python Package For Nuclear Reactor Kinetics

Kathryn Huff‡*

https://www.youtube.com/watch?v=2HToG61wMWI

Abstract—In this work, a new python package, PyRK (Python for Reactor Kinetics), is introduced. PyRK has been designed to simulate, in zero dimensions, the transient, coupled, thermal-hydraulics and neutronics of time-dependent behavior in nuclear reactors. PyRK is intended for analysis of many commonly studied transient scenarios including normal reactor startup and shutdown as well as abnormal scenarios including Beyond Design Basis Events (BDBEs) such as Accident Transients Without Scram (ATWS). For robustness, this package employs various tools within the scientific python ecosystem. For additional ease of use, it employs a reactor-agnostic, object-oriented data model, allowing nuclear engineers to rapidly prototype nuclear reactor control and safety systems in the context of their novel nuclear reactor designs.

Index Terms—engineering, nuclear reactor, package

Introduction

Time-dependent fluctuations in neutron population, fluid flow, and heat transfer are essential to understanding the performance and safety of a reactor. Such *transients* include normal reactor startup and shutdown as well as abnormal scenarios including Beyond Design Basis Events (BDBEs) such as Accident Transients Without Scram (ATWS). However, no open source tool currently exists for reactor transient analysis. To fill this gap, PyRK (Python for Reactor Kinetics) [Huff2015], a new python package for nuclear reactor kinetics, was created. PyRK is the first open source tool capable of:

- time-dependent,
- lumped parameter thermal-hydraulics,
- coupled with neutron kinetics,
- in 0-dimensions,
- for nuclear reactor analysis,
- of any reactor design,
- in an object-oriented context.

As background, this paper will introduce necessary concepts for understanding the PyRK model and will describe the differential equations representing the coupled physics at hand. Next, the implementation of the data model, simulation framework, and numerical solution will be described. This discussion will include the use, in PyRK [Huff2015], of many parts of the scientific python software ecosystem such as NumPy [vanderWalt2011] for array manipulation, SciPy [Milman2011] for ODE and PDE solvers, nose [Pellerin2015] for testing, Pint [Grecco2014] for unit-checking, Sphinx [Brandl2009] for documentation, and Matplotlib [Hunter2007] for plotting.

* Corresponding author: katyhuff@gmail.com
‡ University of California, Berkeley

Background

Fundamentally, nuclear reactor transient analyses must characterize the relationship between neutron population and temperature. These two characteristics are coupled together by reactivity, ρ , which characterizes the departure of the nuclear reactor from *criticality*:

$$\rho = \frac{k-1}{k} \tag{1}$$

where

$$\rho = \text{reactivity} \tag{2}$$

$$k = \text{neutron multiplication factor} \tag{3}$$

$$= \frac{\text{neutrons causing fission}}{\text{neutrons produced by fission}}. \tag{4}$$

The reactor power is stable (*critical*) when the effective multiplication factor, k, equals 1. For this reason, in all power reactors, the scalar flux of neutrons determines the power. The reactor power, in turn, affects the temperature. Reactivity feedback then results due to the temperature dependence of geometry, material densities, the neutron spectrum, and reaction probabilities [Bell1970]. This concept is captured in the feedback diagram in Figure 1.

One common method for approaching these transient simulations is a zero-dimensional approximation which results in differential equations called the Point Reactor Kinetics Equations (PRKE). PyRK provides a simulation interface that drives the solution of these equations in a modular, reactor design agnostic manner. In particular, PyRK provides an object oriented data model for generically representing a nuclear reactor system and provides the capability to exchange solution methods from one simulation to another.

The Point Reactor Kinetics Equations can only be understood in the context of neutronics, thermal-hydraulics, reactivity, delayed neutrons, and reactor control.

Neutronics

The heat produced in a nuclear reactor is due to nuclear fission reactions. In a fission reaction, a neutron collides inelastically with

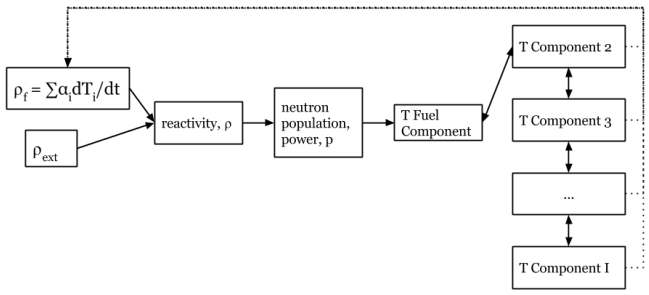


Fig. 1: *Reactivity feedback couples neutron kinetics and thermal hydraulics*

a 'fissionable' isotope, which subsequently splits. This reaction emits both heat and neutrons. When the emitted neutrons go on to collide with another isotope, this is called a nuclear chain reaction and is the basis of power production in a nuclear reactor. The study of the population, speed, direction, and energy spectrum of neutrons in a reactor as well as the related rate of fission at a particular moment is called neutronics or neutron transport. Neutronics simulations characterize the production and destruction of neutrons in a reactor and depend on many reactor material properties and component geometries (e.g., atomic densities and design configurations).

Thermal-Hydraulics

Reactor thermal hydraulics describes the mechanics of flow and heat in fluids present in the reactor core. As fluids are heated or cooled in a reactor core (e.g. due to changes in fission power) pressure, density, flow, and other parameters of the system respond accordingly. The fluid of interest in a nuclear reactor is typically the coolant. The hydraulic properties of this fluid depend primarily on its intrinsic properties and the characteristics of the cooling system. Thermal hydraulics is also concerned with the heat transfer between the various components of the reactor (e.g., heat generation in the reactor fuel heat removal by the coolant). Heat transfer behavior depends on everything from the moderator density and temperature to the neutron-driven power production in the fuel.

Reactivity

The two physics (neutronics and thermal-hydraulics) are coupled by the notion of reactivity, which is related to the probability of fission. The temperature and density of materials can increase or decrease this probability. Fission probability directly impacts the neutron production and destruction rates and therefore, the reactor power. The simplest form of the equations dictating this feedback are:

$$\rho(t) = \rho_0 + \rho_f(t) + \rho_{ext}$$

where

$$\begin{aligned}\rho(t) &= \text{total reactivity}\\ \rho_f(t) &= \text{reactivity from feedback}\\ \rho_{ext}(t) &= \text{external reactivity insertion}\end{aligned}$$

and where

$$\rho_f(t) = \sum_i \alpha_i \frac{\delta T_i}{\delta t}$$

$$\begin{aligned}T_i &= \text{temperature of component i}\\ \alpha_i &= \text{temperature reactivity coefficient of i.}\end{aligned}$$

The PRKE

The Point Reactor Kinetics Equations (PRKE) are the set of equations that capture neutronics and thermal hydraulics when the time-dependent variation of the neutron flux shape is neglected. That is, neutron population is captured as a scalar magnitude (a *point*) rather than a geometric distribution. In the PRKE, neutronics and thermal hydraulics are coupled primarily by reactivity, but have very different characteristic time scales, so the equations are quite stiff.

$$\frac{d}{dt}\begin{bmatrix} p \\ \zeta_1 \\ . \\ . \\ . \\ \zeta_j \\ . \\ . \\ . \\ \zeta_J \\ \omega_1 \\ . \\ . \\ . \\ \omega_k \\ . \\ . \\ . \\ \omega_K \\ T_i \\ . \\ . \\ . \\ T_I \end{bmatrix} = \begin{bmatrix} \frac{\rho(t,T_i,\cdots)-\beta}{\Lambda} p + \sum_{j=1}^{j=J} \lambda_{d,j}\zeta_j \\ \frac{\beta_1}{\Lambda} p - \lambda_{d,1}\zeta_1 \\ . \\ . \\ . \\ \frac{\beta_j}{\Lambda} p - \lambda_{d,j}\zeta_j \\ . \\ . \\ . \\ \frac{\beta_J}{\Lambda} p - \lambda_{d,J}\zeta_J \\ \kappa_1 p - \lambda_{FP,1}\omega_1 \\ . \\ . \\ . \\ \kappa_k p - \lambda_{FP,k}\omega_k \\ . \\ . \\ . \\ \kappa_{kp} - \lambda_{FP,k}\omega_k \\ f_i(p, C_{p,i}, T_i, \cdots) \\ . \\ . \\ . \\ f_I(p, C_{p,I}, T_I, \cdots) \end{bmatrix} \tag{5}$$

In the above matrix equation, the following variable definitions are used:

$$p = \text{ reactor power} \tag{6}$$
$$\rho(t, T_{fuel}, T_{cool}, T_{mod}, T_{refl}) = \text{ reactivity} \tag{7}$$
$$\beta = \text{ fraction of neutrons that are delayed} \tag{8}$$
$$\beta_j = \text{ fraction of delayed neutrons from precursor group j} \tag{9}$$
$$\zeta_j = \text{ concentration of precursors of group j} \tag{10}$$
$$\lambda_{d,j} = \text{ decay constant of precursor group j} \tag{11}$$
$$\Lambda = \text{ mean generation time} \tag{12}$$
$$\omega_k = \text{ decay heat from FP group k} \tag{13}$$
$$\kappa_k = \text{ heat per fission for decay FP group k} \tag{14}$$
$$\lambda_{FP,k} = \text{ decay constant for decay FP group k} \tag{15}$$
$$T_i = \text{ temperature of component i} \tag{16}$$

The PRKE in equation 5 can be solved in numerous ways, using either loose or tight coupling. Operator splitting, loosely coupled in time, is a stable technique that neglects higher order nonlinear terms in exchange for solution stability. Under this approach, the system can be split clearly into a neutronics sub-block and a thermal-hydraulics sub-block which can be solved independently at each time step, combined, and solved again for the next time step.

$$U^n = \begin{bmatrix} N^n \\ T^n \end{bmatrix} \tag{17}$$
$$N^{n+1} = N^n + kf(U^n) \tag{18}$$

$$U^* = \begin{bmatrix} N^{n+1} \\ T^n \end{bmatrix} \tag{19}$$
$$T^{n+1} = T^n + kf(U^*) \tag{20}$$

PyRK Implementation

Now that the premise of the problem is clear, the implementation of the package can be discussed. Fundamentally, PyRK is object oriented and modular. The important object classes in PyRK are:

- SimInfo: Reads the input file, manages the solution matrix, Timer, and communication between neutronics and thermal hydraulics.
- Neutronics : Calculates $\frac{dP}{dt}$, $\frac{d\zeta_j}{dt}$, and $\frac{d\omega_j}{dt}$, based on $\frac{dT_i}{dt}$ and the external reactivity insertion.
- THSystem : Manages various THComponents and facilitates their communication during the lumped parameter heat transfer calculation.
- THComponent : Represents a single thermal volume, made of a single material, (usually a volume like "fuel" or "coolant" or "reflector" with thermal or reactivity feedback behavior distinct from other components in the system.
- Material : A class for defining the intensive properties of a material (c_p, ρ, k_{th}). Currently, subclasses include FLiBe, Graphite, Sodium, SFRMetal, and Kernel.

A reactor is made of objects, so an object-oriented data model provides the most intuitive user experience for describing a reactor system, its materials, thermal bodies, neutron populations, and their attributes. In PyRK, the system, comprised by those objects is built up by the user in the input file in an intuitive fashion. Each of the classes that enable this object oriented model will be discussed in detail in this section.

SimInfo

PyRK has implemented a casual context manager pattern by encapsulating simulation information in a SimInfo object. This class keeps track of the neutronics system and its data, the thermal hydraulics system (THSystem) and its components (THComponents), as well as timing and other simulation-wide parameters.

In particular, the SimInfo object is responsible for capturing the information conveyed in the input file. The input file is a python file holding parameters specific to the reactor design and transient scenario. However, a more robust solution is anticipated for future versions of the code, relying on a json input file rather than python, for more robust validation options.

The current output is a plain text log of the input, runtime messages, and the solution matrix. The driver automatically generates a number of plots. However, a more robust solution is anticipated for v0.2, relying on an output database backend in hdf5, via the pytables package.

Neutronics

The neutronics object holds the first 1+j+k equations in the right hand side of the matrix equation in 5. In particular, it takes ownership of the vector of $1 + j + k$ independent variables and their solution. It also customizes the equations based on paramters noted in the user input file. The parameters customizing these equations for a particular reactor include α_i for each component, j, Λ, k, and the fissionable nuclide.

The Neutronics class has three attributes that are sufficiently complex as to warrant their own classes: PrecursorData, DecayHeat, and ReactivityInsertion.

A Neutronics object can own one PrecursorData object. In this class, the input parameters J and the fissionable nuclide are used to select, from a database supplied by PyRK, standardized data representing delayed neutron precursor concentrations and the effective decay constants of those precursors ($\lambda_{d,j}, \beta_j, \zeta_j$. That nuclear data is stored in the PrecursorData class, and is made available to the Neutronics class through a simple API.

A Neutronics object can also own one DecayHeat object. In this class, the input parameters K, and the fissionable nuclide are used to select, the fission product decay data ($\lambda_{FP,k}, \omega_k, \kappa_k$. The DecayHeat class provides a simple API for accessing those decay constants, fission product fractions, and weighting factors.

Finally, a Neutronics object can own one ReactivityInsertion object. This defines the external reactivity, rho_ext, resulting from control rods, external neutron sources, etc. With this ReactivityInsertion object, the Neutronics class is equipped to drive a reactivity insertion accident scenario. That is, an accident scenario can be driven by an insertion of reactivity (e.g. the removal of a control rod). In PyRK, this reactivity insertion capability is captured in the ReactivityInsertion class, from which reactivity insertions can be selected and customized as in Figure 2.

THSystem

A reactor is made up of many material structures which, in addition to their neutronic response, vary in their temperature response. These structures may include fuel, cladding, coolant, reflectors, or other components. In PyRK, a heat transfer model of the changing temperatures and material properties of those components has been implemented as a lumped capacitance model.

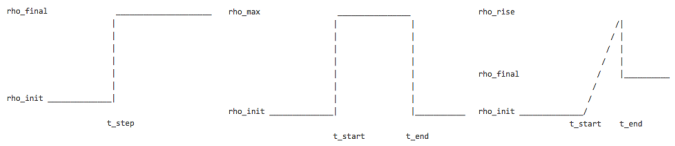


Fig. 2: *The reactivity insertion that can drive the PyRK simulator can be selected and customized from three models.*

Mode	Heat Transfer Rate	Thermal Resistance
Conduction	$\dot{Q} = \frac{T_1 - T_2}{\left(\frac{L}{kA}\right)}$	$\frac{L}{kA}$
Convection	$\dot{Q} = \frac{T_{surf} - T_{envr}}{\left(\frac{1}{h_{conv} A_{surf}}\right)}$	$\frac{1}{h_{conv} A_{surf}}$
Radiation	$\dot{Q} = \frac{T_{surf} - T_{surr}}{\left(\frac{1}{h_r A_{surf}}\right)}$	$\frac{1}{h_r A}$ $h_r = \varepsilon\sigma(T_{surf}^2 + T_{surr}^2)(T_{surf} + T_{surr})$

This model approximates heat transfer into discrete components, approximating the effects of geometry for "lumps" of material.

In this model, heat transfer through a system of components is modeled analogously to current through a resistive circuit. Table 1 describes the various canonical forms of lumped capacitance heat transfer modes.

Based on the modes in Table 1, we can formulate a model for component temperatures specific to to the geometry of a particular reactor design. This might include fuel pellets, particles, or pebbles, cladding, coolant, reflectors or other structures in the design.

Fundamentally, to determine the temperature change in a thermal body of the reactor, we rely on relations between temperature, heat capacity, and thermal resistance. As in Table 1, the heat flow out of body i is the sum of surface heat flow by conduction, convection, radiation, and other mechanisms to each adjacent body, j [Lienhard2011]:

$$\begin{aligned} Q &= Q_i + \sum_j Q_{ij} \\ &= Q_i + \sum_j \frac{T_i - T_j}{R_{th,ij}} \end{aligned}$$

where

$$\begin{aligned} \dot{Q} &= \text{total heat flow out of body i } [J \cdot s^{-1}] \\ Q_i &= \text{other heat transfer, a constant } [J \cdot s^{-1}] \\ T_i &= \text{temperature of body i } [K] \\ T_j &= \text{temperature of body j } [K] \\ j &= \text{adjacent bodies } [-] \\ R_{th} &= \text{thermal resistence of the component } [K \cdot s \cdot J^{-1}]. \end{aligned}$$

Note also that the thermal energy storage and release in the body is accordingly related to the heat flow via capacitance:

$$\frac{dT_i}{dt} = \frac{-Q + \dot{S}_i}{C_i}$$

TABLE 1: *Lumped Capacitance for various heat transfer modes [Lienhard2011]*

where

$$\begin{aligned} C &= \text{heat capacity of the object } [J \cdot K^{-1}] \\ &= (\rho c_p V)_i \\ \dot{S}_i &= \text{source term, thermal energy conversion } [J \cdot s^{-1}] \end{aligned}$$

Together, these form the equation:

$$\frac{dT_i}{dt} = \frac{-\left[Q_i + \sum_j \frac{T_i - T_j}{R_{th,ij}}\right] + \dot{S}_i}{(\rho c_p V)_i}$$

THComponent

The THSystem class is made up of THComponent objects, linked together at runtime by heat transfer interfaces selected by the user in the input file:

```
fuel = th.THComponent(name="fuel",
                      mat=Kernel(name="fuelkernel"),
                      vol=vol_fuel,
                      T0=t_fuel,
                      alpha_temp=alpha_f,
                      timer=ti,
                      heatgen=True,
                      power_tot=power_tot)

cool = th.THComponent(name="cool",
                      mat=Flibe(name="flibe"),
                      vol=vol_cool,
                      T0=t_cool,
                      alpha_temp=alpha_c,
                      timer=ti)

clad = th.THComponent(name="clad",
                      mat=Zirconium(name="zirc"),
                      vol=vol_clad,
                      T0=t_clad,
                      alpha_temp=alpha_clad,
                      timer=ti)

components = [fuel, clad, cool]

# The fuel conducts to the cladding
fuel.add_conduction('clad', area=a_fuel)
clad.add_conduction('fuel', area=a_fuel)

# The clad convects to the coolant
clad.add_convection('cool', h=h_clad, area=a_clad)
cool.add_convection('clad', h=h_clad, area=a_clad)
```

In the above example, the *mat* argument must include a Material object.

Material

The PyRK Material class allows for materials of any kind to be defined within the system. This class represents a generic material and daughter classes inheriting from the Material class describe specific types of material (water, graphite, uranium oxide, etc.). The attributes of a material object are intrinsic material properties (such as thermal conductivity, $k_i h$) as well as material-specific behaviors.

Given these object classes, the burden of the user is then confined to:

- defining the simulation information (such as duration or preferred solver)
- defining the neutronic parameters associated with each thermal component
- defining the materials of each component
- identifying the thermal components
- and connecting those components together by their dominant heat transfer mode.

Quality Assurance

For robustness, a number of tools were used to improve robustness and reproducibility in this package. These include:

- GitHub : for version control hosting [GitHub2015]
- Matplotlib : for plotting [Hunter2007]
- Nose : for unit testing [Pellerin2015]
- NumPy : for holding and manipulating arrays of floats [vanderWalt2011]
- Pint : for dimensional analysis and unit conversions [Grecco2014]
- SciPy : for ode solvers [Oliphant2007], [Milman2011]
- Sphinx : for automated documentation [Brandl2009]
- Travis-CI : for continuous integration [Travis2015]

Together, these tools create a functional framework for distribution and reuse.

Unit Validation

Of particular note, the Pint package[Grecco2014]_ is used for keeping track of units, converting between them, and throwing errors when unit conversions are not sane. For example, in the code below, the user is able to initialize the material object with k_{th} and c_p in any valid unit for those quantities. Upon initialization of those member variables, the input values are converted to SI using Pint.

```
def __init__(self, name=None,
             k=0*units.watt/units.meter/units.kelvin,
             cp=0*units.joule/units.kg/units.kelvin,
             dm=DensityModel()):
    """Initalizes a material

    :param name: The name of the component
    :type name: str.
    :param k: thermal conductivity, :math:`k_{th}`
    :type k: float, pint.unit.Quantity
    :param cp: specific heat capacity, :math:`c_p`
    :type cp: float, pint.unit.Quantity
    :param dm: The density of the material
    :type dm: DensityModel object
    """
    self.name = name
    self.k = k.to('watt/meter/kelvin')
    validation.validate_ge("k", k,
        0*units.watt/units.meter/units.kelvin)
    self.cp = cp.to('joule/kg/kelvin')
    validation.validate_ge("cp", cp,
        0*units.joule/units.kg/units.kelvin)
    self.dm = dm
```

The above code employs a validation utility written for PyRK and used throughout the code to confirm (at runtime) types, units, and valid ranges for parameters of questionable validity. Those validators are simple, but versatile, and in combination with the Pint package, provide a robust environment for users to experiment with parameters in the safe confines of dimensional accuracy.

Minimal Example : SFR Reactivity Insertion

To demonstrate the use of this simulation framework, we give a minimal example. This example approximates a 1-second impulse-reactivity insertion in a sodium cooled fast reactor. This type of simulation is common, as it represents the instantaneous removal and reinsertion of a control rod. The change in reactivity results in a slightly delayed change in power and corresponding increases in temperatures throughout the system. For simplicity, the heat exchanger outside of the reactor core is assumed to be perfectly efficient and the inlet coolant temperature is accordingly held constant throughout the transient.

Minimal Example: Input Parameters

The parameters used to configure the simulation were retrieved from **??** and **??**. The detailed input is listed in the full input file with illuminating comments as follows:

```
import math
from ur import units
import th_component as th
from timer import Timer
from sfrmetal import SFRMetal
from sodium import Sodium

##########################################
#
# User Workspace
#
##########################################

# Timing: t0=initial, dt=step, tf=final
t0 = 0.00*units.seconds
dt = 0.005*units.seconds
tf = 5.0*units.seconds

# Temperature feedbacks of reactivity (Ragusa2009)
# Fuel: Note Doppler model not implemented
alpha_f = (-0.8841*units.pcm/units.kelvin)
# Coolant
alpha_c = (0.1263*units.pcm/units.kelvin)

# Initial Temperatures
t_fuel = 737.033*units.kelvin
t_cool = 721.105*units.kelvin
t_inlet = units.Quantity(400.0, units.degC)
t_inlet.ito(units.kelvin)

# Neglect decay heating
kappa = 0.00

# Geometry
# fuel pin radius
r_fuel = 0.00348*units.meter
# active core height
h_core = 0.8*units.meter
# surface area of fuel pin
a_fuel = 2*math.pi*r_fuel*h_core
# volume of a fuel pin
vol_fuel = math.pi*pow(r_fuel, 2)*h_core
# hydraulic area per fuel pin
a_flow = 5.281e-5*pow(units.meter, 2)
# volume of coolant per pin
vol_cool = a_flow*h_core
# velocity of coolant
v_cool = 5.0*units.meter/units.second

# constant heat transfer approximation
h_cool = 1.0e5*(units.watt/
                units.kelvin/
                pow(units.meter, 2))
# power density
omega = 4.77E8*units.watt/pow(units.meter, 3)
# total power, watts, thermal, per 1 fuel pin
power_tot = omega*vol_fuel

##########################################
#
# Required Input
#
##########################################

# maximum number of ode solver internal steps
nsteps = 1000

# Timer instance, based on t0, tf, dt
ti = Timer(t0=t0, tf=tf, dt=dt)

# Number of precursor groups
n_pg = 6
```

```
# Number of decay heat groups
n_dg = 0

# Fissioning Isotope
fission_iso = "sfr"

# Spectrum
spectrum = "fast"

# False to turn reactivity feedback off.
feedback = True

# External Reactivity
from reactivity_insertion \
    import ImpulseReactivityInsertion as pulse
rho_ext = pulse(timer=ti,
                t_start=1.0*units.seconds,
                t_end=2.0*units.seconds,
                rho_init=0.0*units.delta_k,
                rho_max=0.05*units.delta_k)


fuel = th.THComponent(name="fuel",
                      mat=SFRMetal(name="sfrfuel"),
                      vol=vol_fuel,
                      T0=t_fuel,
                      alpha_temp=alpha_f,
                      timer=ti,
                      heatgen=True,
                      power_tot=power_tot)

cool = th.THComponent(name="cool",
                      mat=Sodium(name="sodiumcoolant"),
                      vol=vol_cool,
                      T0=t_cool,
                      alpha_temp=alpha_c,
                      timer=ti)

inlet = th.THComponent(name="inlet",
                      mat=Sodium(name="sodiumcoolant"),
                      vol=vol_cool,
                      T0=t_inlet,
                      alpha_temp=0.0*units.pcm/units.K,
                      timer=ti)

# The clad convects with the coolant
fuel.add_convection('cool', h=h_cool, area=a_fuel)
cool.add_convection('fuel', h=h_cool, area=a_fuel)

# The coolant flows
cool.add_mass_trans('inlet', H=h_core, u=v_cool)

components = [fuel, cool, inlet]
```

Minimal Example Results

The results of this simulation are a set of plots, the creation and labelling of which are enabled by matplotlib. In the first of these plots, the transient, beginning at time $t = 1s$, is driven by a step reactivity insertion of 0.5 "dollars" of reactivity as in Figure 3.

The power responsds accordingly as in Figure 4.

Finally, the temperatures in the key components of the system follow the trends in Figure 5.

These are typical of the kinds of results nuclear engineers seek from this kind of analysis and can be quickly re-parameterized in the process of prototyping nuclear reactor designs. This particular simulation is not sufficiently detailed to represent a benchmark, as the effect of the cladding on heat transfer is neglected, as is the Doppler model controlling fuel temperature feedback. However, it presents a sufficiently interesting case to demonstrate the use of the PyRK tool.

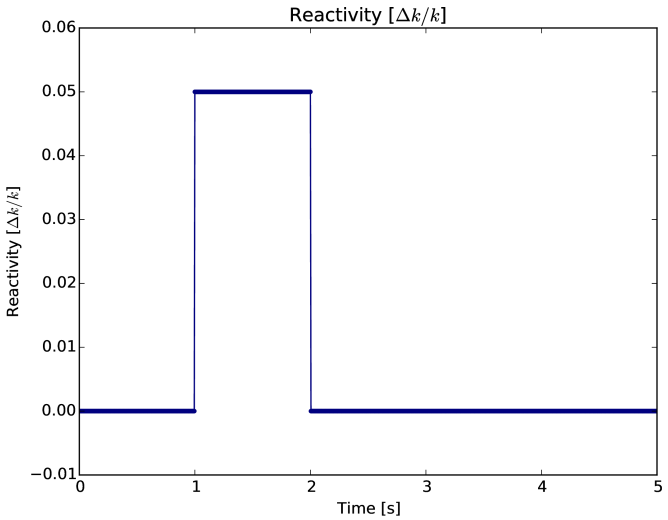


Fig. 3: *A prompt reactivity insertion, with a duration of 1 second and a magnitude of* $0.05\delta k/k$ *drives the simulation. It represents the prompt partial removal and reinsertion of a control rod.*

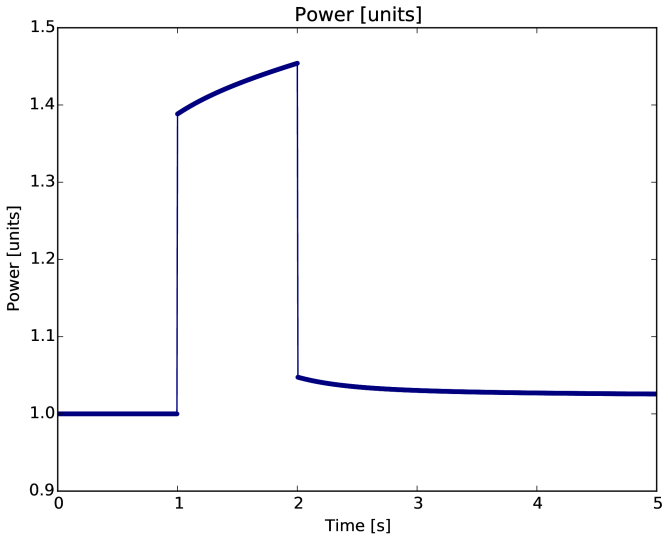


Fig. 4: *The power in the reactor closely follows the reactivity insertion, but is magnified as expected.*

Conclusions and Future Work

The PyRK library provides a modular simulation environment for a common and essential calculation in nuclear engineering. PyRK is the first freely distributed tool for neutron kinetics. By supplying and API for ANSI standard precursor data, a modular material definition framework, and coupled lumped parameter thermal hydraulics with zero-dimensional neutron kinetics in an object-oriented modeling paradigm, PyRK provides design-agnostic toolkit for accident analysis potentially useful to all nuclear reactor designers and analysts.

Acknowledgements

The author would like to thank the contributions of collaborators Xin Wang, Per Peterson, Ehud Greenspan, and Massimiliano Fratoni at the University of California Berkeley. This research was performed using funding received from the U.S. Department

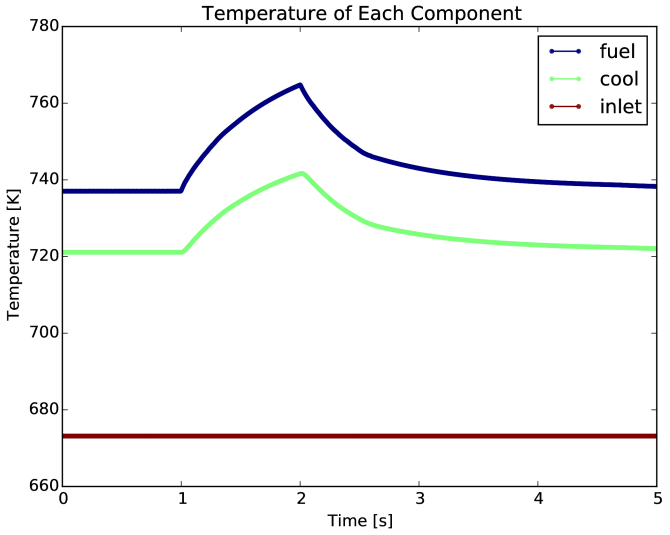


Fig. 5: *While the inlet temperature remains constant as a boundary condition, the temperatures of fuel and coolant respond to the reactivity insertion event.*

of Energy Office of Nuclear Energy's Nuclear Energy University Programs through the FHR IRP. Additionally, this material is based upon work supported by the Department of Energy National Nuclear Security Administration under Award Number: DE-NA0000979 through the Nuclear Science and Security Consortium.

REFERENCES

[Andreades2014] C. Andreades, A. T. Cisneros, J. K. Choi, A. Y. . Chong, D. L. Krumwiede, L. Huddar, K. D. Huff, M. D. Laufer, M. Munk, R. O. Scarlat, J. E. Seifried, N. Zwiebaum, E. Greenspan, and P. F. Peterson, "Technical Description of the 'Mark 1' Pebble-Bed, Fluoride-Salt-Cooled, High-Temperature Reactor Power Plant," University of California, Berkeley, Department of Nuclear Engineering, Berkeley, CA, Thermal Hydraulics Group UCBTH-14-002, Sep. 2014.

[Bell1970] G. I. Bell and S. Glasstone, Nuclear Reactor Theory. New York: Van Nostrand Reinhold Company, 1970.

[Brandl2009] G. Brandl, Sphinx: Python Documentation Generator. URL: http://sphinx. pocoo. org/index. html (13.8. 2012), 2009.

[GitHub2015] GitHub, "GitHub: Build software better, together," GitHub, 2015. [Online]. Available: https://github.com. [Accessed: 17-Jun-2015].

[Grecco2014] H. E. Grecco, Pint: a Python Units Library. https://github.com/hgrecco/pint. 2014.

[Huff2015] K. Huff, PyRK: Python for Reactor Kinetics. https://pyrk.github.io. 2015.

[Hunter2007] J. D. Hunter, "Matplotlib: A 2D Graphics Environment," Computing in Science & Engineering, vol. 9, no. 3, pp. 90–95, 2007.

[Lienhard2011] Lienhard V and J. H. Lienhard IV, A Heat Transfer Textbook: Fourth Edition, Fourth Edition edition. Mineola, N.Y: Dover Publications, 2011.

[Milman2011] K. J. Millman and M. Aivazis, "Python for Scientists and Engineers," Computing in Science & Engineering, vol. 13, no. 2, pp. 9–12, Mar. 2011.

[Oliphant2007] T. E. Oliphant, "Python for Scientific Computing," Computing in Science & Engineering, vol. 9, no. 3, pp. 10–20, 2007.

[Pellerin2015] J. Pellerin, nose. https://pypi.python.org/pypi/nose/1.3.7. 2015.

[Ragusa2009] J. C. Ragusa and V. S. Mahadevan, "Consistent and accurate schemes for coupled neutronics thermal-hydraulics reactor analysis," Nuclear Engineering and Design, vol. 239, no. 3, pp. 566–579, Mar. 2009.

[Sofu2011] T. Sofu, "A review of inherent safety characteristics of metal alloy sodium-cooled fast reactor fuel against postulated accidents," Nuclear Engineering and Technology, vol. 47, no. 3, pp. 227–239, Apr. 2015.

[Travis2015] Travis, "travis-ci/travis-api," GitHub repository. Available: https://github.com/travis-ci/travis-api. Accessed: 04-Jul-2015.

[vanderWalt2011] S. van der Walt, S. C. Colbert, and G. Varoquaux, "The NumPy Array: A Structure for Efficient Numerical Computation," Computing in Science & Engineering, vol. 13, no. 2, pp. 22–30, Mar. 2011.

VisPy: Harnessing The GPU For Fast, High-Level Visualization

Luke Campagnola[§*], Almar Klein[‡], Eric Larson[¶], Cyrille Rossant[‖], Nicolas Rougier[**]

https://www.youtube.com/watch?v=_3YoaeoiIFI

Abstract—The growing availability of large, multidimensional data sets has created demand for high-performance, interactive visualization tools. VisPy leverages the GPU to provide fast, interactive, and beautiful visualizations in a high-level API. Here we introduce the main features, architecture, and techniques used in VisPy.

Index Terms—graphics, visualization, plotting, performance, interactive, opengl

Motivation

Despite the rapid growth of the scientific Python stack, one conspicuously absent element is a standard package for high-performance visualization. The de facto standard for plotting is Matplotlib [matplotlib]; however, this package is designed for publication graphics and is not optimized for visualizations that require real-time, interactive performance, or that incorporate large data volumes. Several packages in the Python ecosystem are capable of high-performance visualization: VTK [vtk] provides an extensive set of 3D visualization tools with Python bindings, Chaco [chaco] offers efficient 2D plotting, PyQtGraph [pyqtgraph] is a scientific GUI library with fast plotting, Glumpy [glumpy] implements high-quality plotting primitives in OpenGL, and VisVis [visvis] and Galry [galry] both provide high-performance 2D/3D OpenGL visualization. Each of these packages has its particular strengths and weaknesses and, although the Python community benefits from such a rich ecosystem, at the same time it suffers from the lack of a focused, collaborative effort.

In recognition of this problem and the potential benefit to the Python community, VisPy [vispy] was created as a collaborative effort to succeed several of these projects—visvis, galry, glumpy, and the visualization components of pyqtgraph. VisPy has quickly grown an active community of developers and is approaching beta status.

What is VisPy

VisPy is a scientific visualization library based on OpenGL and NumPy [numpy]. Its primary purpose is to deliver high-performance rendering under heavy load, but at the same time we aim to provide publication-quality graphics, a high-level 2D and 3D plotting API, and portability across many platforms. VisPy's main design criteria are:

- *High-performance for large data sets.* By making use of the modern, shader-based OpenGL pipeline, most of the graphical rendering cost is offloaded to the graphics processor (GPU). This allows real-time interactivity even for data on the order of tens of millions of samples, and at the same time minimizes CPU overhead.
- *High-level visualization tools.* Most Python developers are not graphics experts. Getting from raw data to interactive visualization should require as little code as possible, and should require no knowledge of OpenGL or the underlying graphics hardware.
- *Publication quality output.* Commodity graphics hardware and the modern OpenGL shader pipeline have made it possible to render moderately large data sets without sacrificing quality in primitive shapes or antialiasing [rougier2013a], [rougier2013b]. VisPy is also designed to enable vector graphics output, although this feature is not yet implemented.
- *Flexibility.* VisPy strives to make common tasks easy—most basic plot types can be generated with just a few lines of code. At the same time, VisPy makes complex and niche tasks possible through a flexible and extensible architecture. VisPy's library of graphical components can be reconfigured and recombined to build complex, interactive scenes.
- *Portability.* VisPy's reliance on commodity graphics hardware for optimization reduces its reliance on CPU-optimized code or numerous external dependencies; VisPy is pure-Python and depends only on NumPy and a suitable GUI library. This makes VisPy easy to distribute and install across many platforms, including WebGL-enabled browsers.

VisPy's Architecture

VisPy's functionality is divided into a layered architecture, with each new layer providing higher-level primitives. The top layers provide a powerful system for quickly and easily visualizing data, whereas the lower layers provide greater flexibility and control over OpenGL's features.

* *Corresponding author: luke.campagnola@gmail.com*
§ *University of North Carolina at Chapel Hill*
‡ *Continuum Analytics*
¶ *University of Washington*
‖ *University College London*
** *French National Institute for Research in Computer Science and Control*

Layer 1: Object-Oriented GL

The OpenGL API, although very powerful, is also somewhat verbose and unwieldy. VisPy's lowest-level layer, `vispy.gloo`, provides an object-oriented OpenGL wrapper with a clean, compact, and Pythonic alternative to traditional OpenGL programming (Figure 1). Developers unfamiliar with OpenGL are encouraged to work from the scenegraph and plotting layers instead. Objects that typically require several GL calls to instantiate, such as textures, vertex buffers, frame buffers, and shader programs, are instead encapsulated in simple Python classes. The following example demonstrates creating a shader program and assigning a value to one of its uniform variables:

```
program = Program(vert_code, frag_code)
program['color'] = (1, 0.5, 0, 1)
```

The equivalent code using the OpenGL API is somewhat more verbose:

```
prg = glCreateProgram()
vsh = glCreateShader(GL_VERTEX_SHADER)
glShaderSource(vsh, vert_code)
fsh = glCreateShader(GL_FRAGMENT_SHADER)
glShaderSource(fsh, vert_code)
for shader in (vsh, fsh):
    glCompileShader(shader)
    assert glGetShaderParameter(shader,
                                GL_COMPILE_STATUS) == 1
    glAttachShader(prg, shader)
glLinkProgram(prg)
assert glGetProgramParameter(prg,GL_LINK_STATUS) == 1
nunif = glGetProgramParameter(prg,GL_ACTIVE_UNIFORMS)
uniforms = {}
for i in range(nunif):
    name, id, typ = glGetActiveAttrib(prg, i)
    uniforms[name] = id
glUseProgram(prg)
glUniform4fv(uniforms['color'], 1, (1, 0.5, 0, 1))
```

Most OpenGL commands cannot be invoked until a context, provided by the GUI toolkit, has been created and activated. This requirement imposes design limitations that can make OpenGL programs more awkward. To circumvent this restriction, `vispy.gloo` uses a context management system that queues all OpenGL commands until the appropriate context has become active. The direct benefit is that the end user is free to interact with `vispy.gloo` however makes sense for their program. Most notably, `vispy.gloo` objects can be instantiated when the program starts up, before any context is available.

The command queues used by `vispy.gloo` are also designed to be serializable such that commands generated in one process or thread can be executed in another. In this way, a stream of GL commands could be sent to a web browser such as the IPython notebook, recorded to disk to be replayed later, or shared between processes to take advantage of multi-core systems.

Another purpose of `vispy.gloo` is to hide many of the differences between various versions and implementations of OpenGL. We currently target OpenGL versions 2.1 for desktop systems and ES2.0 for embedded and WebGL systems, which are available on virtually all commodity hardware today. Systems that lack a modern GPU may still run VisPy code using a software OpenGL implementation such as Mesa [mesa3d]. Notably, this is used by Travis CI [travisci] to run our unit tests. However, OpenGL versions older than 2.1 are not supported. VisPy also supports some features from OpenGL 3+ but these currently depend on pyopengl [pyopengl].

A closely related system, `vispy.app`, abstracts the differences between the various supported GUI backends, which include

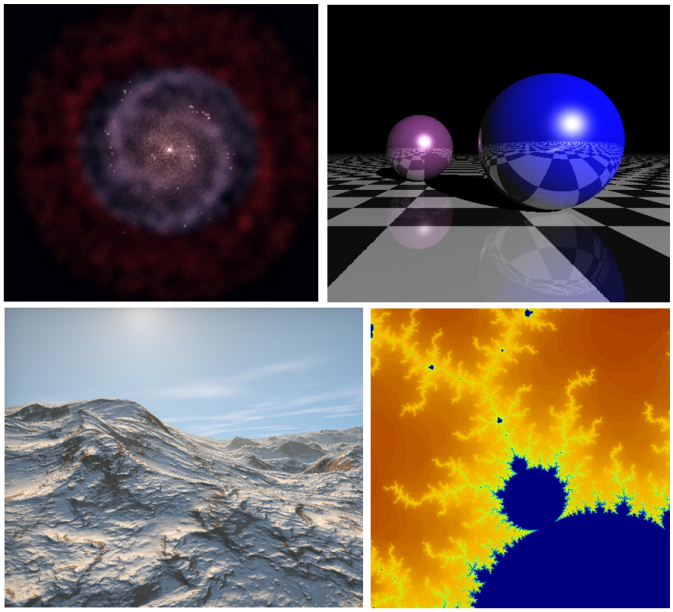


Fig. 1: *A selection of demos written with `vispy.gloo`. This layer provides low-level access to OpenGL with a simple and Pythonic API. It is primarily used to implement visual classes; however, developers who are familiar with OpenGL may find this a suitable starting point for some visualization tasks.*

PyQt4/5, PySide, IPython, SDL, GLFW, and several others. This system provides uniform access to user input, timers, and window features across all backends, and allows VisPy to be incorporated into most existing applications. VisPy can be used as a Qt widget, embedded in IPython notebook, or run on a headless server with almost no code differences. This support, combined with VisPy's pure-python and low-dependency approach, helps to ensure that VisPy will run on most platforms with minimal effort from users and developers alike.

Layer 2: Visuals

The core of VisPy is its library of `Visual` classes that provide the primitive graphical objects used to build more complex visualizations. These objects range from very simple primitives (lines, points, triangles) to more powerful primitives (text, volumes, images), to high-level visualization tools (histograms, surface plots, spectrograms, isosurfaces). Figure 2 shows several examples of visuals implemented in VisPy.

Internally, visuals upload their data to graphics memory and implement a shader program [glsl] that is executed on the GPU. Because all OpenGL implementations since 2.0 include an OpenGL shader language (GLSL) compiler, this allows the most computationally intensive operations to run in compiled, parallelized code without adding any build dependencies. Visuals can be reconfigured and updated in real time by simply uploading new data or shaders to the GPU. Before drawing, each visual also configures the necessary OpenGL global state such as blending and depth testing. These state parameters may be reconfigured for each visual to select different compositing modes.

Visuals may also be modified by applying arbitrary coordinate transformations and filters such as opacity, clipping, and lighting. To support this flexibility, it is necessary to be able to recombine smaller chunks of shader code. VisPy implements a shader management system that allows independent GLSL functions to

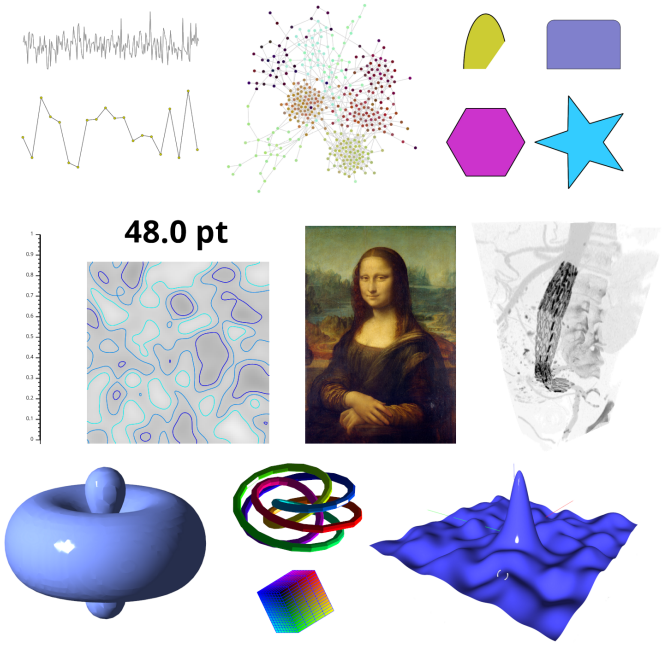


Fig. 2: A selection of VisPy's visuals. These span the range from simple 2D and 3D primitives to more advanced visualization tools like contour plots, surface plots, and volume renderings. More complex visualizations can be built from combinations of these visuals.

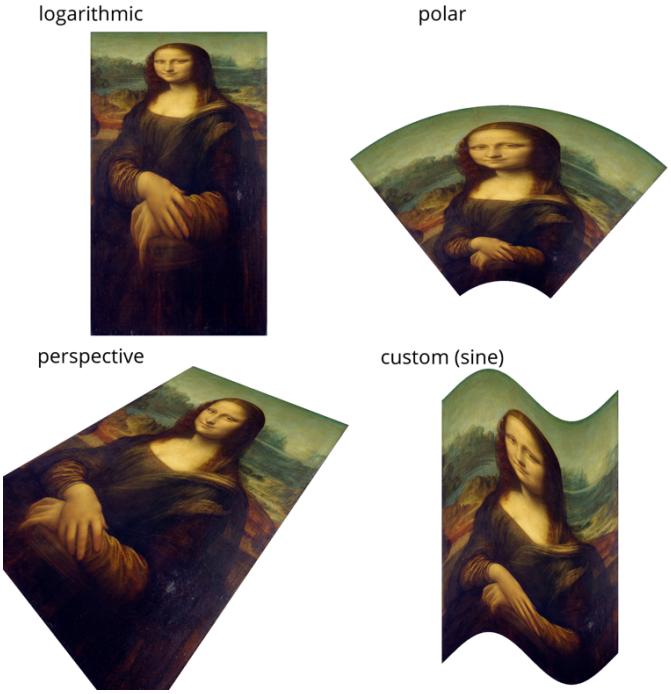


Fig. 3: One image viewed using four different coordinate transformations. VisPy supports linear transformations such as scaling, translation, and matrix multiplication (bottom left) as well as nonlinear transformations such as logarithmic (top left) and polar (top right). Custom transform classes are also easy to construct (bottom right).

be attached together in a single shader program. This enables the insertion of arbitrary coordinate transformations and color modification into each visual's shader program.

VisPy implements a collection of coordinate transformation classes that are used to map between a visual's raw data and its output coordinate system (screen, image, svg, etc.). By offloading coordinate transformations to the GPU along with drawing operations, VisPy makes it possible to stream data directly from its source to the GPU without any modification in Python. Most transforms affect the location, orientation, and scaling of visuals and can be chained together to produce more complex adjustments. Transforms may also be nonlinear, as in logarithmic, polar, and Mercator projections, and custom transforms can be implemented easily by defining the forward and inverse mapping functions in both Python and GLSL.

The following example summarizes the code that produces the logarithmically-scaled image in Figure 3. It combines a scale/translation, followed by log base 2 along the y axis, followed by a second scale/translation to set the final position on screen. The resulting chained transformation maps from the image's pixel coordinates to the window's pixel coordinates:

```
from vispy import visuals
from vispy.visuals.transforms import (STTransform,
                                      LogTransform)

# Create an image from a (h, w, 4) array
image = visuals.ImageVisual(image_data)

# Assign a chain of transforms to stretch the image
# logarithmically and set its placement in the window
tr1 = STTransform(scale=(1, -0.01),
                  translate=(-50, 1.3))
tr2 = LogTransform((0, 2, 0))
tr3 = STTransform(scale=(3, -150),
                  translate=(200, 100))
image.transform = tr3 * tr2 * tr1
```

Quality and Optimization in Visuals

One of VisPy's main challenges is to implement visuals that simultaneously satisfy three major design constraints: high performance, high quality, flexibility, and portability. In reality, no single visualization algorithm can cover all of the possible use cases for a single visual. For example, algorithms that provide the highest quality may impact performance, techniques that improve performance may not be available on all platforms, and some combinations of techniques naturally require an inflexible implementation.

In VisPy's approach, each visual implements multiple rendering algorithms that share the same API. The user may then select for different performance and quality targets and the visual will gracefully fall back to safer techniques if the platform requires it. For example, drawing a surface plot with lighting requires a normal vector to be calculated for each location on the surface. If the surface vertex positions are specified in a floating point texture, then the normal calculation can be performed on the GPU. However, older OpenGL versions (and current WebGL implementations) lack the necessary texture support. For these cases, extra effort is required to either encode the vertex positions in a different type of texture, or to perform the normal calculation on the CPU. Alternatively, the surface can be rendered with a lower quality method that does not require normal vector calculation.

More generally, optimizing for performance often requires consideration for two different targets: data *volume* and data *throughput*. In the former case, a large but static data set is uploaded to the GPU once but subsequently viewed or modified interactively. This case is typically limited by the efficiency of the shader programs, and thus it may help to pre-process the data once on the CPU to lighten the recurring load on the GPU. In

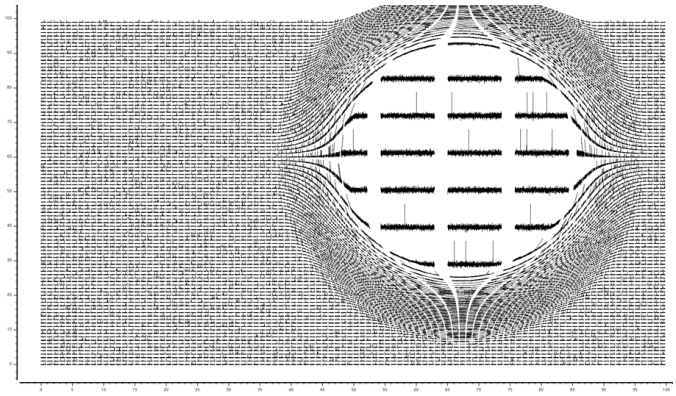


Fig. 4: *A large collection of scrolling plots rendered with a specialized visual (`examples/demo/scene/scrolling_plots.py`). There are 10,000 plots, each containing 2,000 data points for a total of 20 million points drawn per frame. The plots are scrolled continuously as new data is streamed to the GPU, and still render at 35 fps on the author's laptop. A region of the plot is enlarged using a nonlinear transform.*

the latter case, data is being rapidly streamed to the GPU and is typically displayed only once before being discarded. This case tends to be limited by the per-update CPU overhead, and thus may be optimized by offloading more effort to the GPU. Intertwined with these optimization targets are quality considerations—often performance can be improved by sacrificing rendering quality, but the true performance gain of each sacrifice can be unpredictable.

By wrapping multiple rendering techniques within a single API, the user is freed from the burden of restructuring their application for each technique. Some cases, however, are too unique to fit comfortably in a generic API. For example, Figure 4 uses a specialized visual to draw a 100x100 grid of scrolling plots, each containing 2,000 data points. This example could be implemented using the basic line visual techniques, but independently updating each of the 10,000 lines as they scroll would be prohibitively slow. The example is able to run over 30 fps by organizing the data in memory as a 2D circular buffer, which allows all plots to be updated in a single operation. The essential lines of this example are summarized below:

```
lines = ScrollingLines(n_lines=10e3, line_size=2e3,
                       columns=100, dt=4e-4,
                       cell_size=(1, 8))

def update(ev):
    # add 10 samples to each plot
    data = np.random.normal(size=(N, 10), scale=0.3)
    data[data > 1] += 4  # random spikes
    lines.roll_data(data)

timer = app.Timer(connect=update, interval=0)
timer.start()
```

Layer 3: Scenegraph

Layer 3 implements common features required for interactive visualization, and is the first layer that requires no knowledge of OpenGL. This is the main entry point for most users who build visualization applications. Although the majority of VisPy's graphical features can be accessed by working directly with its Visual classes (layer 2), it can be confusing and tedious to manage the visuals, coordinate transforms, and filters for a complex scene. To automate this process, VisPy implements a scenegraph—a standard data structure used in computer graphics that organizes visuals into a hierarchy. Each node in the hierarchy inherits coordinate transformations and filters from its parent. VisPy's scenegraph allows visuals to be easily arranged in a scene and, in automating control of the system of transformations, it is able to handle some common interactive visualization requirements:

- *Picking.* User input from the mouse and touch devices are delivered to the objects in the scene that are clicked on. This works by rendering the scene to an invisible framebuffer, using unique colors for each visual; thus the otherwise expensive ray casting computation is carried out on the GPU.
- *Interactive viewports.* These allow the user to interactively pan, scale, and rotate data within the view, and the visuals inside the view are clipped to its borders.
- *Cameras.* VisPy contains a variety of camera classes, each implementing a different mode of visual perspective or user interaction. For example, `PanZoomCamera` allows panning and scaling for 2D plot data, whereas `ArcballCamera` allows data to be rotated in 3D like a trackball.
- *Lighting.* The user may add lights to the scene and shaded objects will react automatically.
- *Export.* Any portion of the scene may be rendered to an image at any resolution. We also plan to add support for exporting a scenegraph to SVG.
- *Layouts.* These automatically partition window space into grids allowing multiple visualizations to be combined in a single window.
- *High-resolution displays.* The scenegraph automatically corrects for high-resolution displays to ensure visuals are scaled correctly on all devices.

The example below is a simple demonstration of creating a scenegraph window and adding visuals to its scene:

```
import vispy.scene as vs

# Create a window with a grid layout inside
window = vs.SceneCanvas()
grid = window.central_widget.add_grid()

# Create a view with a 2D line plot inside
view1 = grid.add_view(row=0, col=0, camera='panzoom')
plot = vs.PlotLine(data1, parent=view1.scene)

# Create a second view with a 3D surface plot
view2 = grid.add_view(row=0, col=1,
                      camera='turntable')
surf = vs.SurfacePlot(data2, parent=view2.scene)

# Adjust the position and orientation of
# the surface plot
surf.transform = vs.AffineTransform()
surf.transform.translate(2, 1, 0)
surf.transform.rotate(30, 0, 1, 0)

# start UI event loop
window.app.run()
```

Adding mouse interaction requires the ability to determine which visuals are under the mouse cursor and to map between the coordinate systems of the canvas and the visual. In the example below, the coordinate system mapping corrects for the scale and translation of a 2D interactive view box:

```
@window.connect
def on_mouse_press(event):
```

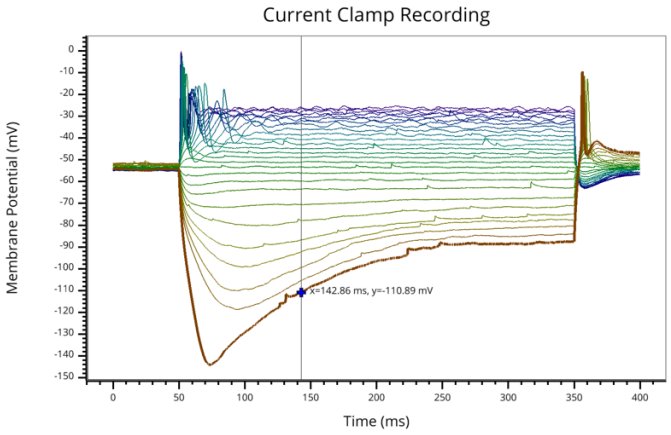


Fig. 5: *Mouse interaction example (`examples/demos/scene/picking.py`). In this example, mouse press events are captured and a list of visuals near the mouse is generated using `canvas.visuals_at(pos, radius=10)`. The list of visuals is returned in order of proximity to the mouse, allowing the nearest line to be selected. Mouse movement events are captured in a separate callback and used to update the plot cursor. The location along the plot line and the cursor placement are all determined by mapping the mouse position into the local coordinate system of the selected visual.*

```
# get the visual under the click
vis = window.visual_at(event.pos)

# map the click position to the coordinate
# system of the visual
tr = window.scene.node_transform(vis)
pos = tr.map(event.pos)

print("Clicked on %s at %s" % (vis, pos))
```

A more complete mouse interaction example is described in Figure 5.

Layer 4: Plotting

VisPy's plotting layer allows quick and easy access to advanced data visualization, such as plotting, image display, volume rendering, histograms, and spectrograms. This layer is intended for use in simple analysis scripts or in an interactive session, and is similar in principle to Matplotlib's `pyplot` API. The following example creates a window displaying a plot line and a spectrogram of the same data:

```
import numpy as np
from vispy import plot as vp

# Create a logarithmic chirp
fs = 1000.
N = 1000000
t = np.arange(N) / float(fs)
f0, f1 = 1., 500.
phase = ((t[-1] / np.log(f1 / f0)) * f0 *
         (pow(f1 / f0, t / t[-1]) - 1.0))
data = np.cos(2 * np.pi * phase)

# Create a figure in a new window and add a
# spectrogram and line in separate sub-plots.
fig = vp.Fig(size=(800, 400))
fig[0:2, 0].spectrogram(data, fs=fs, clim=(-100, -20))
fig[2, 0].plot(np.array((t, data)).T, marker_size=0)
```

The output of this code is shown in Figure 6.

Despite the large volume of data, the resulting views can be immediately panned and zoomed in real-time. As a rough performance comparison, the same plot data can be redrawn at about 0.2 Hz by Matplotlib, 2 Hz by PyQtGraph, and over 30 Hz by VisPy on the author's machine.

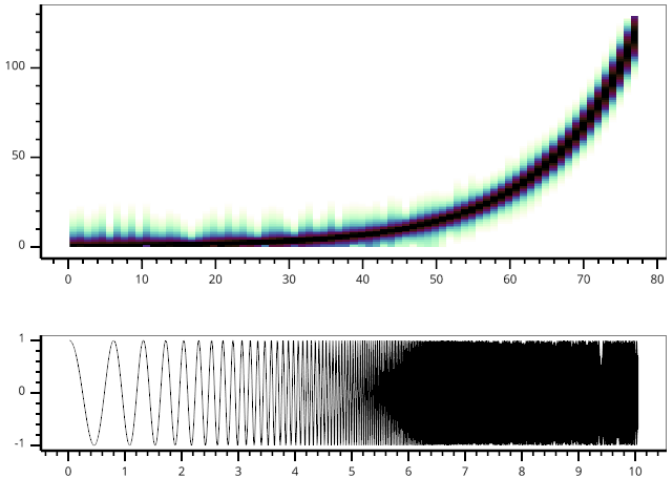


Fig. 6: *Example `vispy.plot` output (from `examples/basics/plotting/spectrogram.py`). This figure requires only three lines to generate, excluding the data generation: one to create the figure window, and one each for the spectrogram and line plots. The plot areas can be zoomed and panned with the mouse. Despite containing 1e6 samples, the plots update smoothly.*

Each function in `vispy.plot` generates scenegraph (layer 3) objects to allow lower level control over the visual output. This makes it possible to begin development with the simplest `vispy.plot` calls and iteratively refine the output as needed. VisPy also includes an experimental wrapper around mplexporter [mplexporter] that allows it to act as a drop-in replacement for Matplotlib in existing projects. This approach, however, is not always expected to have the same performance benefits as using the native `vispy.plot` API.

The `vispy.plot` interface is currently the highest-level and easiest layer VisPy offers. Consequently, it is also the least mature. We expect this layer to grow quickly in the coming months as we add more plot types and allow the API to settle.

Future Work

Our immediate goal for vispy is to stabilize the visual, scenegraph, and plotting APIs, and implement the most pressing basic features. We are continuously testing for performance under different use cases and ensuring that behavior is consistent across all platforms. In the long term, we plan to implement more advanced features:

- *Add more plot types.* The scope of `vispy.plot` encompasses a very broad range of high-level visualizations, only a few of which are currently implemented. Expanding this library of visualizations will be an ongoing process. In the future we expect to support vector fields, flow charts, parametric surfaces, bar charts, and many more.
- *Add more interactive tools.* With VisPy it should be simple to select, manipulate, and slice many different kinds of data. The scenegraph makes this easier by providing support for picking, but we would like to add a set of higher level tools such as region of interest boxes, rotation gimbals, contrast and colormap controls, etc. We also plan to allow picking individual vertices within a single visual.

- *SVG export.* This is a must-have feature for any visualization library that targets publication graphics, and a high priority for VisPy. Most 2D visuals will be simple to implement as they have direct analogs in the SVG standard. Other visuals, however, may simply be rendered as an image in the export process.
- *Backend and OpenGL support.* VisPy currently supports most desktop platforms and has preliminary support for IPython notebook. We are working to add support for mobile devices and embedded systems like the Raspberry Pi, as well as a wider range of web backends. We would also like to expand support for newer GPU features such as geometry and tesselation shaders and general purpose GPU computing libraries like Cuda [cuda] and OpenCL [opencl].
- *Collections.* This system will allow many visuals to be joined together and drawn with a single call to OpenGL. This is expected to greatly improve performance when many static visuals are displayed in the scene. This will allow efficiently drawing complex shapes such as maps,
- *Order-independent blending*. This technique will allow translucent visuals to be correctly blended without the need to sort the visuals by depth first. This will greatly improve the rendering quality of many 3D scenes.

With the base plotting API almost settled, VisPy is rapidly approaching beta status when it will become more useful to a broader audience. In the long term we hope VisPy will continue to flourish and expand its community of developers.

REFERENCES

[vispy] *VisPy: OpenGL-based interactive visualization in Python* http://vispy.org

[matplotlib] J. D. Hunter. *Matplotlib: A 2D graphics environment*, Computing In Science & Engineering, 9(3):90-95, IEEE COMPUTER SOC, 2007.

[vtk] Kitware. *VTK - The Visualization Toolkit*, http://www.vtk.org/

[chaco] Enthought, Inc. *Chaco*, http://code.enthought.com/projects/chaco/

[pyqtgraph] L. Campagnola. *PyQtGraph. Scientific Graphics and GUI Library for Python*, http://www.pyqtgraph.org/

[glumpy] N. Rougier. *Glumpy: fast, scalable and beautiful scientific visualization*, https://glumpy.github.io/

[visvis] A. Klein. *visvis - The object oriented approach to visualization.* https://code.google.com/p/visvis/

[galry] C. Rossant. *Galry: high performance interactive visualization package in Python*, https://github.com/rossant/galry

[numpy] S. van der Walt, S.C. Colbert and G. Varoquaux, *The NumPy Array: A Structure for Efficient Numerical Computation*, Computing in Science & Engineering, 13, 22-30, 2011.

[mesa3d] *The Mesa 3D Graphics Library* http://www.mesa3d.org/

[travisci] *Travis CI* https://travis-ci.org/

[pyopengl] *PyOpenGL* http://pyopengl.sourceforge.net/

[glsl] *OpenGL Shading Language* https://www.opengl.org/documentation/glsl/

[rougier2013a] Nicolas P. Rougier, *Higher Quality 2D Text Rendering*, Journal of Computer Graphics Techniques (JCGT), vol. 2, no. 1, 50-64, 2013. Available online http://jcgt.org/published/0002/01/04/

[rougier2013b] Nicolas P. Rougier, *Shader-Based Antialiased, Dashed, Stroked Polylines*, Journal of Computer Graphics Techniques (JCGT), vol. 2, no. 2, 105--121, 2013 Available online http://jcgt.org/published/0002/02/08/

[mplexporter] mpld3, *mplexporter - A general scraper/exporter for matplotlib plots* https://github.com/mpld3/mplexporter

[opencl] Khronos Group, *OpenCL - The open standard for parallel programming of heterogeneous systems*, https://www.khronos.org/opencl/

[cuda] nVidia, *CUDA - Paallel Programming and Computing Platform*, http://www.nvidia.com/object/cuda_home_new.html

White Noise Test: detecting autocorrelation and nonstationarities in long time series after ARIMA modeling

Margaret Y Mahan‡*, Chelley R Chorn‡, Apostolos P Georgopoulos‡

Abstract—Time series analysis has been a dominant technique for assessing relations within datasets collected over time and is becoming increasingly prevalent in the scientific community; for example, assessing brain networks by calculating pairwise correlations of time series generated from different areas of the brain. The assessment of these relations relies, in turn, on the proper calculation of interactions between time series, which is achieved by rendering each individual series stationary and nonautocorrelated (i.e., white noise, or to "prewhiten" the series). This ensures that the relations computed subsequently are due to the interactions between the series and do not reflect internal dependencies of the series themselves. An established method for prewhitening time series is to apply an Autoregressive (AR, *p*) Integrative (I, *d*) Moving Average (MA, *q*) model (ARIMA) and retain the residuals. To diagnostically check whether the model orders (*p*, *d*, *q*) are sufficient, both visualization and statistical tests (e.g., Ljung-Box test) of the residuals are performed. However, these tests are not robust for high-order models in long time series. Additionally, as dataset size increases (i.e., number of time series to model) it is not feasible to visually inspect each series independently. As a result, there is a need for robust alternatives to diagnostic evaluations of ARIMA modeling. Here, we demonstrate how to perform ARIMA modeling of long time series using *Statsmodels*, a library for statistical analysis in Python. Then, we present a comprehensive procedure (White Noise Test) to detect autocorrelation and nonstationarities in prewhitened time series, thereby establishing that the series does not differ significantly from white noise. This test was validated using time series collected from magnetoencephalography recordings. Overall, our White Noise Test provides a robust alternative to diagnostic checks of ARIMA modeling for long time series.

Index Terms—Time series, Statsmodels, ARIMA, statistics

Introduction

Time series are discrete, stochastic realizations of underlying data generating processes [Yaffee]. In other words, a time series is a set of consecutive samples collected over a time interval, such as temperature recordings at regular intervals. They are ubiquitous in any field where monitoring of data is involved. For example, time series can be environmental, economic, or medical. In addition, time series can provide information about trends (e.g., broad fluctuations in values) and cycles (e.g., systematic, periodic fluctuations in values). Time series analysis are also used to predict the next value in the series, given some model of its history. This is of special importance in environmental and econometric studies where forecasting the next set of values (e.g., the weather or a stock price) may have serious practical consequences. In other fields, time series provide crucial information about an evolving process (e.g., rate of spread of a disease or changing pollution levels) with implications about the effect of interventions. Finally, time series can provide fundamental information about the process that generates them, leading to a scientific understanding of that process (e.g., brain network analysis).

In time series analysis, there are two main investigative methods: frequency-domain and time-domain. In this paper, only analysis in the time-domain is considered. Within the time-domain, typically crosscorrelation analysis is utilized as a measure of the relation between two time series. Now, it is commonly the case that a time series contains some autocorrelation, meaning that values in the time series are influenced by previous values. It is also common for a time series to exhibit nonstationarities, such as drifts or trends over time. In either case, the crosscorrelation function calculated between two series containing either autocorrelation or nonstationarities will give misleading results, such as an inflated correlation between two series where there is none. To circumvent this, time series are modeled to remove such effects, as in the case of prewhitening.

Prewhitening

A white noise process is a continuous time series of random values, with a constant mean and variance, normally and independently distributed, and nonautocorrelated. If after modeling a time series the residuals are practically white noise, then we say the series has been prewhitened. An established method for prewhitening time series is to apply an Autoregressive (AR) Integrative (I) Moving Average (MA) model (ARIMA) and retain the residuals [Box]. The full specification of an ARIMA model comprises the orders of each component, (p, d, q), where p is the number of preceding values in the autoregressive component, d is the number of differencing, and q is the number of preceding values in the moving average component. An ARIMA model with orders p, d, and q, is a discrete time linear equations with noise of the form:

$$(1-\sum_{k=1}^{p}\phi_k L^k)(1-L)^d X_t = (1+\sum_{k=1}^{q}\theta_k L^k)\varepsilon_t$$

* Corresponding author: mahan027@umn.edu
‡ Brain Sciences Center, Minneapolis VA Health Care System & University of Minnesota

where L is the time lag operator, $Lx_t = x_{t-1}$.

In ARIMA modeling, the I component is addressed first, followed by jointly addressing the AR and MA components. Most importantly, the ARIMA method requires the input time series to be: (1) equally spaced over time, (2) of sufficient length, (3) continuous (i.e., no missing values), and, specifically for the ARMA portion, (4) stationary in the second or weak sense, meaning the mean and variance remain constant over time and the autocovariance is only lag-dependent.

Prewhitening using ARIMA modeling takes three main steps. First, identify and select the model, by detecting factors that influence the time series, such as nonstationarities or periodicities, and identifying the AR and MA components (i.e., model orders). Second, estimate parameter values, by using an estimation function to optimize the parameter values for the desired model. Third, evaluate the model, by checking the model's adequacy through establishing that the series has been rendered stationary and nonautocorrelated. This time series modeling is iterative, successively refining the model until stationary and nonautocorrelated residuals are obtained. Overall, a good model serves three purposes: providing the background information for further research on the process that generated the time series; enabling accurate forecasting of future values in the series; and yielding the stationary and nonautocorrelated residuals necessary to evaluate accurately associations between time series, since they are devoid of any dependencies stemming from within the series themselves.

Here, we implement two complementary tests to establish stationarity, which determines the value of the I(d) order. Using these stationary series, we use median correlation values at each lag of the autocorrelation (ACF) and partial autocorrelation (PACF) functions to identify a range of AR(p) and MA(q) orders to implement combinatorially. Then we utilize the *Statsmodels* package to find the method-solver combination that provides good metrics for long time series. Finally, we present a novel approach (White Noise Test) to diagnostic checking of ARIMA modeling for long time series, which evaluates residual series based on stationarity and nonautocorrelation. Using our approach, an investigator can perform ARIMA modeling and evaluate candidate models with ease for large datasets and datasets containing long time series.

Model Identification and Selection

There are several factors that can influence a value in a time series, which arise from previous values in the series, variability in these values, or nonstationarities (trend, drift, changing variance, or random walk). It is important to properly remove the effects of these factors by modeling the time series and taking the residuals. To identify the model orders for an ARIMA(p, d, q), the ACF and PACF are used.

First, nonstationarities need to be removed before ARMA modeling. A nonstationary process is identified by an ACF that does not tail away to zero quickly or cut-off after a finite number of steps. If the time series is nonstationary, then a first differencing of the series is computed. This process is repeated until the time series is stationary, which determines the value of d (i.e., the value of d is the number of times the derivative of the series is taken to achieve stationarity). Two of the most frequently used tests for detecting nonstationarities are the augmented Dickey-Fuller (ADF) test [Said] and the Kwiatkowski–Phillips–Schmidt–Shin (KPSS) test [Kwiatkowski]. The ADF is a unit root test for the null hypothesis that a time series is I(1) while the KPSS is a stationarity test for the null hypothesis that a time series is I(0). Since these tests are complementary, we use them together to determine whether a series is stationary. In our case, a series taken to be nonstationary, if the ADF null hypothesis is accepted and the KPSS null is rejected. We implement the ADF test using *Statsmodels* and the KPSS test using the *Arch* Python package.

Once nonstationarities have been removed, ARMA modeling can begin. To choose the p and q orders, the ACF and PACF of the stationary (differenced) series will show patterns based on which tentative ARMA model can be postulated. There are three main patterns. A pure MA(q) process will have an ACF that cuts off after q lags and a PACF that tails off with exponential or oscillating decay. A pure AR(p) process will have an ACF that tails off with exponential or oscillating decay and a PACF that cuts off after p lags. For a mixed-model ARMA(p, q) process, both the ACF and PACF will tail off with exponential or oscillating decay. Using these patterns, the model selection begins by using the minimum orders to achieve stationary and nonautocorrelated residuals.

Parameter Value Estimation

ARIMA modeling has been implemented in Python with the *Statsmodels* package [McKinney], [Seabold]. It includes parameter value estimation and model evaluation procedures. We import the *Statsmodels* and *Numpy* packages as:

```
import statsmodels.api as sm
import numpy as np
```

After the model orders have been selected, the model parameter values can be estimated with the `sm.tsa.arima_model.ARIMA.fit()` function to maximize the likelihood that these parameter values (i.e., coefficients) describe the data, as follows. First, initial estimates of the parameter values are used to get close to the desired parameter values. Second, optimization functions are applied to adjust the parameter values to maximize the likelihood by minimizing the negative log-likelihood function. If adequate initial parameter value estimates were selected, a local optimization algorithm will find the local log-likelihood minimum near the parameter value estimates, which will be the global minimum.

In *Statsmodels*, default starting parameter value estimations are calculated using the Hannan-Rissanen method [Hannan] and these parameter values are checked for stationarity and invertibility (these concepts are discussed in further detail in the next section). If `method` is set to `css-mle`, starting parameter values are estimated further with conditional sum of squares methods. However, parameter values estimated in this way are not guaranteed to be stationary; therefore, we advise specifying starting parameter values as an input variable (`start_params`) to `ARIMA.fit()`. A custom starting parameter value selection method may be built upon a copy of `sm.tsa.ARMA._fit_start_params_hr`, which forces stationarity and invertibility on the estimated `start_params` when necessary. For example,

```
if not np.all(np.abs(np.roots(np.r_
       [1, -start_params[k:k + p]])) < 1) or
   not np.all(np.abs(np.roots(np.r_
       [1, start_params[k + p:]])) < 1):
   start_params = np.array(start_params[0:k]
                  + [1./(p+1)] * p + [1./(q+1)] * q)
```

In addition, the Hannan-Rissanen method uses an initial AR model with an order selected by minimizing Bayesian Information Criterion (BIC); then it estimates ARMA using the residuals from that model. This initial AR model is required to be larger than max(p, q) of the desired ARIMA model, which is not guaranteed

with an AR selected by BIC criterion. We have implemented a method similar to Hannan-Rissanen, the long AR method, which is equivalent to Hannan-Rissanen except the initial AR model is set to be large (AR = 300). This results in an initial AR model order which is guaranteed to be larger than max(p, q), and starting parameter value selection is more time efficient since fitting multiple AR model orders to optimize BIC is not required.

To fit ARIMA models, *Statsmodels* has options for methods and solvers. The chosen method will determine the type of likelihood for estimation, where `mle` is the exact likelihood maximization (MLE), `css` is the conditional sum of squares (CSS) minimization, and `css-mle` involves first estimating the starting parameter values with CSS followed by an MLE fit. The solver variable in `ARIMA.fit()` designates the optimizer from `scipy.optimize` for minimizing the negative loglikelihood function. Optimization solvers `nm` (Nelder-Mead) and `powell` are the most time efficient because they do not require a score, gradient, or Hessian. The next fastest solvers, `lbfgs` (limited memory Broyden-Fletcher-Goldfarb-Shanno), `bfgs` (Broyden-Fletcher-Goldfarb-Shanno), `cg` (conjugate gradient), and `ncg` (Newton conjugate-gradient), require a score or gradient, but no Hessian. The `newton` (Newton-Raphson) solver requires a score, gradient, and Hessian. Lastly, a global solver `basinhopping`, displaces parameter values randomly before minimizing with another local optimizer. For more information about these solvers, see `sm.base.model.GenericLikelihoodModel`.

Model Evaluation

There are two components in evaluating an ARIMA model, namely, model stability and model adequacy. For the model to be stable, the roots of the characteristic equations

$$1 - \phi_1 L - \cdots - \phi_p L^p = 0$$

where ϕ_i are the estimated AR parameter values, L is the time lag operator, and

$$1 + \theta_1 L + \cdots + \theta_q L^q = 0$$

where θ_i are the estimated MA parameter values, should lie outside the unit circle, i.e., within bounds of stationarity (for the p parameter values) and invertibility (for the q parameter values) [Pankratz]. For the model to be adequate, the residual time series should not be significantly different from white noise; in other words, the series should have constant mean and variance, and each value in the series should be uncorrelated with other realizations up to k lags. If either model stability or adequacy have not been established, then model identification and selection should be revised, and the diagnostic cycle continued, iteratively, until established.

Inspecting the p and q parameter values for being within the bounds of stationarity and invertibility checks model stability. Typically, this will be accomplished during parameter value estimation. The model adequacy is checked by examining the time-varying mean of the residuals (should be close to zero), their variance (should not differ appreciably along time), and their autocorrelation (should not be different from chance). Finally, the ACF and PACF of the residuals should not contain statistically significant terms more than the number expected by chance. This number depends on the number of lags; for example, if k = 40 lags, one would expect 2 values (5% of 40) to exceed their standard error. Under the assumption that the process is white noise and when the length (N) of the series is long, the standard error of the sample autocorrelation (and partial autocorrelation) [Bartlett] approximates to:

$$\text{Standard Error} = 1/\sqrt{N}$$

Several statistical tests are available to detect autocorrelation. Most notable is the Ljung-Box test [Ljung], which is applied to residuals to detect whether they exhibit autocorrelation. The test statistic is calculated for each of h lags being tested. Another common test to detect autocorrelation is the Durbin-Watson test [Durbin]; however, unlike the Ljung-Box test which is calculated for h lags, the Durbin-Watson test is calculated only for lag 1. Therefore, any autocorrelation beyond lag 1 will not be detected by this test. Similar to the Ljung-Box test is the Breusch-Godfrey Lagrange multiplier test [Breusch], [Godfrey]. This test also aims to detect autocorrelation up to h lags tested. We compare our model evaluation, namely the White Noise Test, to both the Ljung-Box and Breusch-Godfrey tests.

White Noise Test

The White Noise Test (Figure 1) calculates multiple attributes on residuals. Inclusively, the attributes characterize an individual residual series by its "whiteness". To change the degree of "whiteness", the thresholds in the red boxes of Figure 1 may be made more or less conservative.

Excluded data: Channels that could not be modeled with the given model order were excluded from further analysis. Additionally, channels with extreme values beyond a threshold of 5 per channel, calculated on the residuals for each model order, were also excluded from further analysis (xVAL in Table 1 and 5). Extreme values are calculated as follows. For each raw series, the interquartile range (IQR) is calculated.

$$IQR = 75^{th}percentile - 25^{th}percentile$$

Using the IQR, Tukey's outer fences are calculated [Tukey].

$$Fence_{upper} = 75^{th}percentile + 3 \times IQR$$

$$Fence_{lower} = 25^{th}percentile - 3 \times IQR$$

Then, the values below the lower fence and above the upper fence are counted as extreme values. If this count is greater than 5, the series is removed from further consideration when selecting model orders.

Normality: Each residual series was tested for normality using the Kolmogorov–Smirnov test. Residual series not significantly different from normal ($\alpha = 0.01$) were retained.

Constant mean: Each residual series was split into 10% nonoverlapping windows (i.e., 10% of 50000 time points = 10 windows of 5000 time points). For each window, a one-sample t-test was calculated ($\alpha = 0.001$). A count of the number of windows with means significantly different from zero was retained for each residual series (maximum value = 10). Residual series with > 1 section containing means significantly different from zero were excluded (cMEAN in Table 1, 3 and 5).

Constant variance: For each residual series, the 10% nonoverlapping windows were also tested for equal variances using Bartlett's test ($\alpha = 0.001$). Each window was compared to the variance of the full residual series. A count of the number of windows with unequal variances was retained for each residual series (maximum value = 10). Residual series with > 1 section containing significantly different unequal variances were excluded (cVAR in Table 1, 3 and 5).

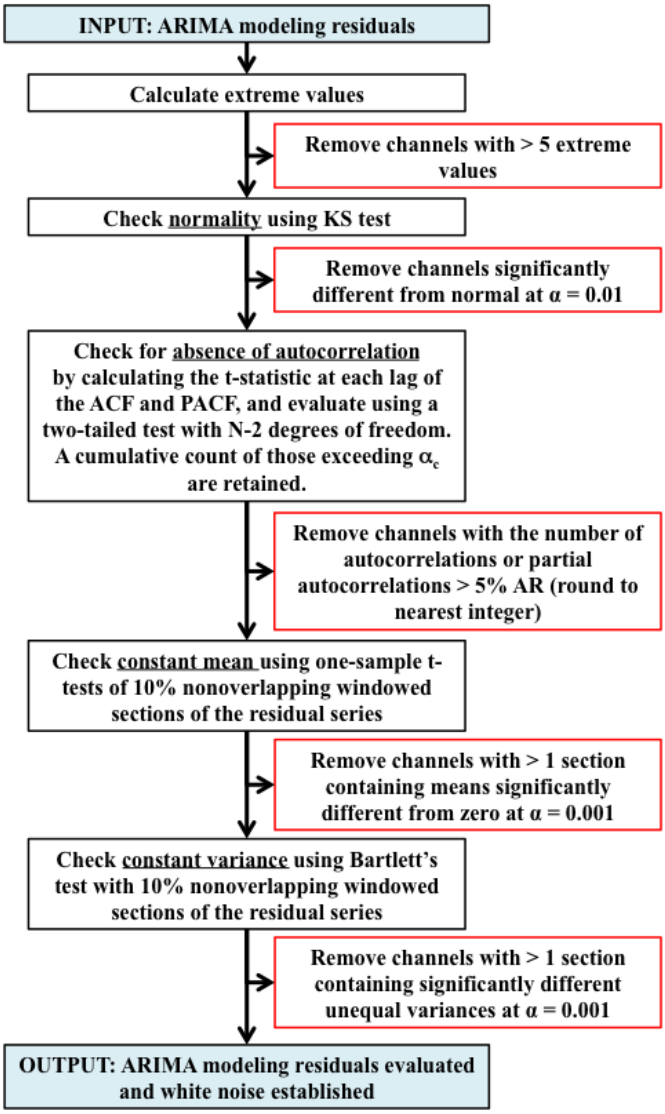


Fig. 1: *White Noise Test Procedure.*

Uncorrelated with other realizations: The ACF and PACF were calculated up to AR lags and the number of lags exceeding statistical significance were counted. To determine this, the

$$t_{statistic} = \frac{|ACF_k|}{StandardError}$$

is calculated at each lag, k, and evaluated against the null hypothesis that ACF_k using a two-tailed test with N-2 degrees of freedom. A cumulative count of those exceeding

$$\alpha_c = \frac{0.01}{AR} \tag{1}$$

are retained (note: α_c incorporates a Bonferroni correction, $\frac{1}{AR}$, and is rounded to the nearest integer). The result is a conservative threshold for detecting a significant autocorrelation or partial autocorrelation. We set a threshold for the cumulative count to be greater than 5% of the AR order (round to nearest integer) for either the ACF or PACF for each channel (tACF and tPACF in Table 1, 3 and 5).

To determine whether our thresholding levels are within what is expected by chance, we apply the White Noise Test procedure

Count	xVAL	tACF	tPACF	cMEAN	cVAR
0	531	593	594	597	596
1	67	0	0	3	4
≥ 2	2	7	6	0	0

TABLE 1: *White Noise Attributes, listed as extreme values (xVAL), thresholded ACF and PACF (tACF, tPACF), constant mean (cMEAN) and constant variance (cVAR), and the count column is the number of randomly generated series failing a given attribute.*

(Figure 1) to 600 randomly generated white noise series. Attributes calculated on these series are shown in Table 1.

Magnetoencephalography (MEG) Dataset

To evaluate the functional brain, MEG is a useful technique because it measures magnetic fluctuations generated by synchronized neural activity in the brain noninvasively and at high temporal resolution. For the applications below, MEG recordings were collected using a 248-channel axial gradiometer system (Magnes 3600WH, 4-D Neuroimaging, San Diego, CA) sampled at ~1 kHz from 50 cognitively healthy women (40 - 93 years, 70.58 ± 14.77, mean ± std dev) in a task-free state (i.e., resting state). The data were time series consisting of 50,000 values per subject and channel. Overall, the full MEG dataset contains 50 samples x 248 channels x 50,000 time points.

Performing ARIMA Modeling

Here, we first determine which method-solver combination from *Statsmodels* provides the most reliable and valid residuals, while also maintaining a respectable processing time for the MEG dataset. Then, using this method-solver, investigations into identifying and selecting model orders are performed, followed by parameter value estimations on a range of model orders. Residuals from these models are processed to detect autocorrelation and nonstationarities using our White Noise Test. Finally, these models are compared and evaluated.

Implementing Method-Solvers

The length and quantity of time series have a direct impact on the ease of modeling. Therefore, we aim to implement an iterative approach to ARIMA modeling while keeping focus on model reliability and validity of residuals, along with incorporating an efficiency cost (i.e., constraints on allowed processing time). The goal for this stage is to determine which method-solver in *Statsmodels* is most appropriate for the application dataset.

To accomplish this, we randomly select 5% (round to nearest integer) of the channels from each sample in the full MEG dataset (i.e., 5% of 248 channels with 50 samples gives N = 600) to construct the test dataset. Next, we select a range of model orders: AR = {10, 20, 30, 40, 50, 60}, I = {1}, MA = {1, 3, 5}. Using each method-solver group (N = 16) and model order combinations (N = 18), we now have 288 testing units. For each of the testing units, ARIMA modeling is performed on each channel in the test dataset.

If 2% of the test dataset channels have a processing time > 5 minutes per channel, the testing unit is withdrawn from further analysis and deemed inefficient. Otherwise, for each channel, four measures are retained. The first measure is the AIC_c (Akaike

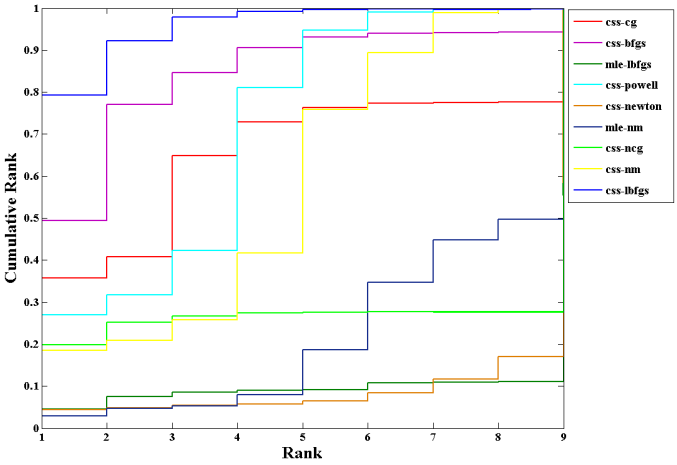


Fig. 2: *MEG CDF Ranks*

Method-Solver	Mean Time (s)	Highest Model	Mean Ranks	Final Rank
`css-lbfgs`	91.47	60-1-3	1.32	1
`css-bfgs`	115.22	60-1-3	2.23	2
`css-powell`	54.47	60-1-5	3.25	3
`css-cg`	132.78	50-1-1	3.77	4
`css-nm`	39.55	60-1-3	4.29	5
`css-ncg`	138.97	20-1-3	6.90	6
`mle-nm`	85.71	30-1-5	7.31	7
`mle-lbfgs`	57.7	10-1-5	8.29	8
`css-newton`	235.11	20-1-1	8.36	9

TABLE 2: *Ranking Method-Solvers for ARIMA modeling of MEG data.*

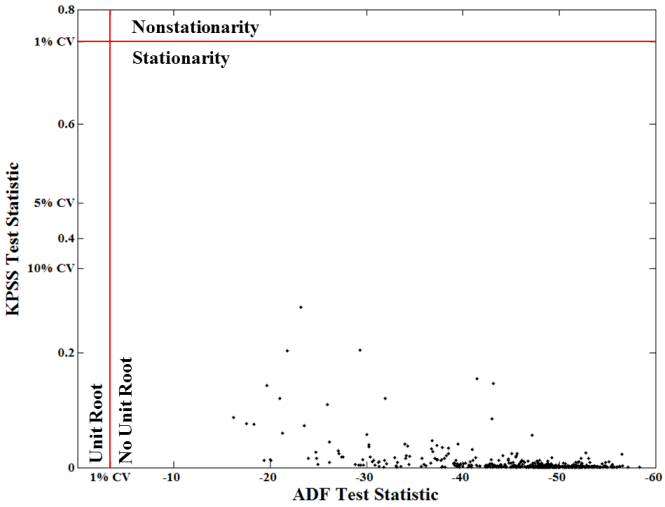


Fig. 3: *Stationarity (KPSS) and Unit Root (ADF) Tests*

Information Criterion with correction), which reflects the quality of the statistical model's performance. The second and third measures are the cumulative counts of tACF and tPACF. The final measure is the processing time, which is measured on each channel and is the time, in seconds, for the ARIMA modeling process to produce residuals. For all four measures, lower values indicate better performance. After calculating the measures, for each channel and model order, ranks for the first three measures are calculated across the method-solver groups, with tied ranks getting the same rank number.

For the 16 method-solver combinations tested, 7 were inefficient at all tested model orders (`css-basinhopping`, `mle-bfgs`, `mle-newton`, `mle-cg`, `mle-ncg`, `mle-powell`, `mle-basinhopping`). The cumulative distribution functions (CDFs) of each method-solver group ranks are calculated and plotted in Figure 2. In this plot, larger area under the curve indicates better performance. Thus, the `css-lbfgs` has the best performance.

In Table 2, the mean time per channel for each method, except withdrawn methods, is given, along with the highest order able to be modeled by the given method-solver group. Mean ranks were calculated for each method-solver, shown in Table 2, and used for the final rank calculation. In the test dataset, the `css-lbfgs` method-solver outperformed all others while maintaining a reasonable time per channel (91.47 seconds). The results also show that the CSS methods generally outperform the MLE methods, for long time series. The `css-lbfgs` method-solver was retained for all further analysis.

Identifying and Selecting Model Orders

Before selecting the differencing model order, d, each series is inspected for extreme values. To determine the model orders, channels with greater than five extreme values are excluded. As discussed previously, if a series is deemed nonstationary, then a first differencing of the series is computed. To determine nonstationarity, examine the ACF plot. A clear indication of nonstationarity will be if the ACF does not tail away to zero quickly or cut-off after a finite number of steps, which is the case with MEG raw time series. Therefore, the MEG time series are first differenced ($d = 1$).

Next we check the series for stationarity; recall, an appropriately differenced process should be stationary. Both the KPSS stationarity test and ADF unit root test are calculated for 60 lags. Their values plotted against each other are shown in Figure 3. The KPSS statistic ranges from 0 to 0.28; since all KPSS test statistics calculated are less than the critical value (CV) of 0.743 at $\alpha = 0.01$, the null hypothesis of stationarity cannot be rejected. The ADF statistic ranges from -16.19 to -58.32; since all ADF test statistics calculated are more negative than the CV of -3.43 at $\alpha = 0.01$, the null hypothesis of a unit root is rejected. Taken together, we have established lack of nonstationaritiy for our test dataset.

Taking the differenced series, the ACF and PACF are calculated for 60 lags. The median correlation value for each lag is plotted in Figure 4. From this figure, a mixed-model ARMA(p, q) process is seen since both the ACF and PACF tail off with oscillating decay. To decide on the p and q orders, we look at Figure 4 and see the highly AR nature of the PACF plot up to about 30 lags; we also see the MA component expressed in the ACF up to about 10 lags. Using this, we decide to implement a range of model orders. For the AR component, we choose to begin with AR = 20 and end with AR = 60 in increments of 5. For the MA component, we choose to begin with MA = 1 and end with MA = 9 in increments of 2. We implement all possible combinations of these ARMA orders (N = 45).

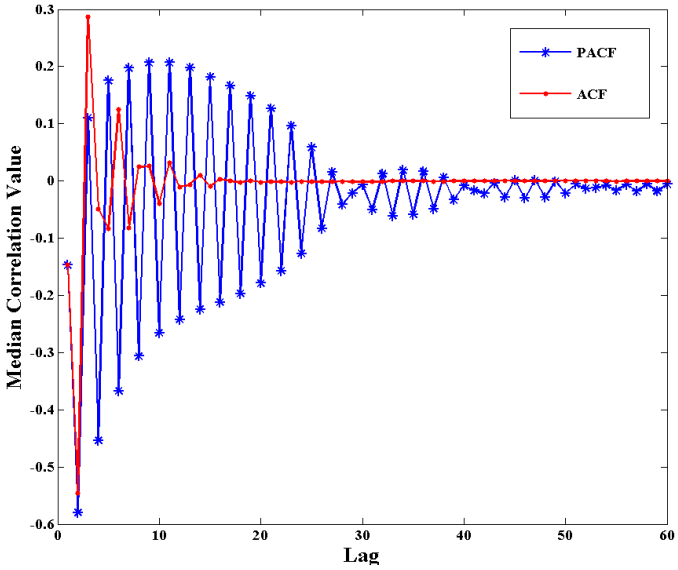


Fig. 4: *ACF and PACF of MEG data after first differencing*

Final Model Order Selection

For each of the 45 model order combinations, the White Noise Test was calculated on the residuals. In the case of the test dataset, there were 4 channels that could not be modeled in each of the model order combinations. Channels with greater than 5 extreme values, and thus excluded, were relatively consistent across model order combinations with a range of 26-29 channels (mean = 27.24, ~5% of the test dataset) per combination. Additionally, residual series were not significantly different from normal ($\alpha = 0.01$). The remaining attributes are shown in Table 3 for up to AR = 50 (AR = 55 and 65 showed similar patterns). In the table, unique channels is the count of unique channels across the tACF, tPACF, cMEAN, and cVAR attributes.

The results in Table 3, show multiple model order combinations provide low counts on several attributes, indicating more than one usable model order combination. However, there are two important patterns that emerge. First, as the AR increases (holding the MA constant), the ACF and PACF counts generally decrease. Second, as the MA increases (holding the AR constant), the ACF and PACF counts generally decrease. Taken together, there exists an ideal candidate model, namely ARIMA(30,1,3). This model order exhibits two qualities to use in evaluating model orders: it is within the lowest on all attribute counts as compared to other model orders, and among those with the lowest attribute values, it has the lowest model orders.

From an analyst perspective, an ideal candidate model is informed by the future analysis to be performed. Basically, when choosing the ideal candidate model, the next stage of analysis needs to be considered and used to identify the ideal candidate model. For instance, if the next stage of analysis is to calculate all possible pairwise partial correlation coefficients between each channel for ±50 lags, then the model order of choice should have an AR $\geq$ 50 or at a minimum, the tACF and tPACF attributes of the residuals need to be examined up to 50 lags. In general, choosing an ideal candidate model will be based on several factors including, but not limited to, the choice of method-solver, future analytic needs, and degree of "whiteness" desired.

We compare our ACF thresholding to two autocorrelation tests, the Ljung-Box and Breusch-Godfrey statistics for up to AR lags, tested at $\alpha = 0.001$, for each residual series. Figure 5 shows a bar graph of the Ljung-Box and ACF counts. The Ljung-Box statistic is calculated at three levels, with degrees of freedom (df) equalling AR, min(20, N-1) as suggested by [Box], and $\ln(N)$ as suggested by [Tsay]. Each bar is for one model order combination with the same labeling as in the first column of Table 3. The bar length is the sum of the elements in the model order combination for the given statistic. Each bar shows different colors for each statistic and the relative contribution each statistic makes to the total sum for that model order combination. The Breusch-Godfrey, in place of the Ljung-Box, showed similar results. It can be seen that the Ljung-Box corresponds well to our ACF thresholding when the df equal the AR order but fails to identify autocorrelation using either of the suggested df. Finally, the Breusch-Godfrey and Ljung-Box statistics are compared in terms of the percent of residual series failing each statistic (Table 4).

#	Model Orders	tACF	tPACF	cMEAN	cVAR	Unique Channels
1	20-1-1	570	570	0	12	570
2	20-1-3	54	54	7	12	70
3	20-1-5	31	31	6	12	49
4	20-1-7	27	27	7	12	46
5	20-1-9	15	15	8	12	34
6	25-1-1	569	569	0	12	569
7	25-1-3	16	16	7	10	33
8	25-1-5	31	31	6	12	49
9	25-1-7	10	10	9	12	31
10	25-1-9	3	3	10	12	24
11	30-1-1	569	569	6	11	569
12	30-1-3	5	5	8	13	26
13	30-1-5	7	7	8	11	26
14	30-1-7	3	3	10	12	25
15	30-1-9	3	3	10	12	23
16	35-1-1	563	563	2	11	563
17	35-1-3	8	8	9	12	28
18	35-1-5	3	3	8	12	23
19	35-1-7	6	6	8	12	26
20	35-1-9	0	0	7	12	19
21	40-1-1	529	529	8	11	530
22	40-1-3	30	30	7	12	47
23	40-1-5	1	1	7	12	20
24	40-1-7	8	8	8	12	27
25	40-1-9	1	1	7	11	19
26	45-1-1	222	222	7	10	234
27	45-1-3	6	6	9	11	26
28	45-1-5	0	0	8	12	20
29	45-1-7	3	3	7	12	22
30	45-1-9	2	2	7	12	21
31	50-1-1	15	15	7	11	33
32	50-1-3	0	0	7	11	18
33	50-1-5	0	0	7	12	19
34	50-1-7	0	0	7	12	19
35	50-1-9	0	0	9	12	21

TABLE 3: *Attributes for the White Noise Test shown for incrementing model order combinations, listed as thresholded ACF and PACF (tACF, tPACF), constant mean (cMEAN) and constant variance (cVAR), and the number of unique channels across the attributes.*

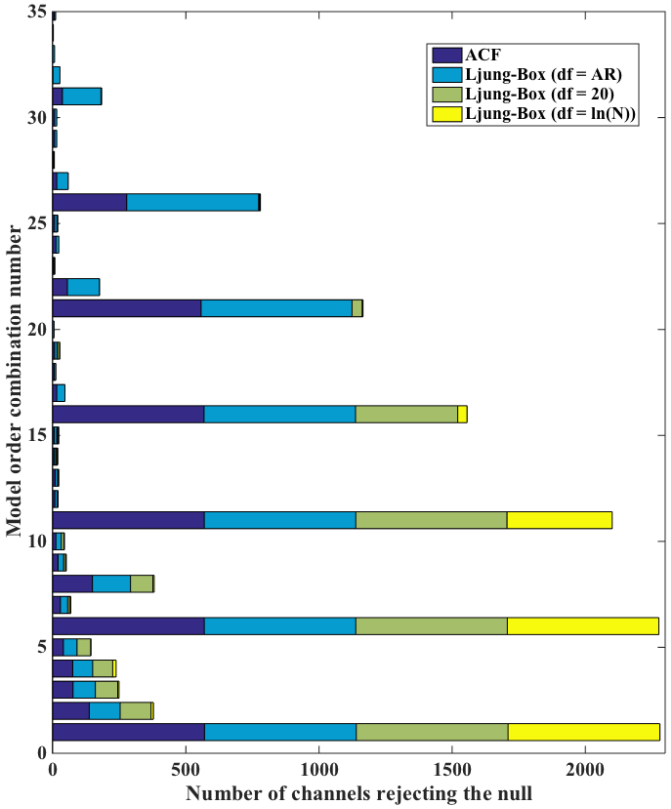


Fig. 5: *ACF and Ljung-Box Attributes Compared*

df	% =	% ≠ by 1	% ≠ by > 1
AR	55.6	20.0	24.4
20	77.8	6.7	15.6
ln(N)	84.4	11.1	4.4

TABLE 4: *Breusch-Godfrey test compared to Ljung-Box test*

MEG Dataset Evaluation

Finally, using ARIMA(30,1,3), we apply the White Noise Test procedure to the full MEG dataset. One channel at each stage of modeling is shown in Figure 6. Descriptive statistics on each of the attributes for the full MEG data are shown in Table 5 and the overall percent of channels removed per subject is shown in Figure 7. One subject had over 200 channels removed, likely due to errors within the recording, and was excluded from Table 5.

Conclusion

In this paper, we presented an expansion on the Box-Jenkins methodology to ARIMA modeling. First, during model identification and selection, we implement two complementary tests (KPSS and ADF) to establish stationarity. Using these stationary series, we use median correlation values at each lag of the ACF and PACF across 600 channels to identify a range of AR(p) and MA(q) order to implement combinatorially. This methodology allows for examining multiple time series simultaneously to determine a valid model order for the majority of time series in a dataset. Second, during parameter value estimation, we utilize the *Statsmodels* package to find the method-solver combination that provides good metrics (model reliability, validity of residuals, and time efficient) for long time series. We found the `css-lbfgs` to outperform all other method-solver combinations on these metrics. Third, during model evaluation, we present a novel approach (White Noise Test: Figure 1) to diagnostic checking of ARIMA modeling for long time series, which evaluates residual series based on stationarity and nonautocorrelation (i.e., "whiteness"). Using this approach, we identify the ideal candidate model for our dataset to be ARIMA(30,1,3). Applying this model to the full MEG dataset, we find an average of 20.55 channels removed from the White Noise Test (i.e., fail to establish "whiteness"), which is about 8.3% of the dataset. Overall, using our approach, an investigator can perform ARIMA modeling and evaluate candidate models with ease for large datasets and datasets containing long time series.

Step	Min	Max	Median	Mean	Std Dev
xVAL	0	60	1	9.67	16.63
Normal	0	0	0	0.00	0.00
tACF	0	51	0	2.53	8.12
tPACF	0	0	0	0.00	0.00
cMEAN	0	8	0	0.20	1.15
cVAR	0	40	4	7.24	9.05
Channels Removed	1	85	10	20.55	21.34

TABLE 5: *Results of White Noise Test on full dataset, with the steps listed as extreme values (xVAL), normality, thresholded ACF and PACF (tACF, tPACF), constant mean (cMEAN) and constant variance (cVAR), and the number of channels removed as a result.*

References

[Bartlett] Bartlett, M.S. 1946. "On the theoretical specification and sampling properties of autocorrelated time-series." *Journal of the Royal Statistical Society*, 8.1, 27-41.

[Box] Box, G. and Jenkins, G. 1976. "Time series analysis: forecasting and control." Holden Day, San Francisco, 2nd edition.

[Breusch] Breusch, T.S. 1978. "Testing for autocorrelation in dynamic linear models", *Australian Economic Papers*, 17, 334–355.

[Durbin] Durbin, J. and Watson, G.S. 1971. "Testing for serial correlation in least squares regression III", *Biometrika*, 58.1, 1–19.

[Godfrey] Godfrey, L.G. 1978. "Testing against general autoregressive and moving average error models when the regressors include lagged dependent variables", *Econometrica*, 49, 1293–1302.

[Hannan] Hannan, E.J. and Rissanen, J. 1985. "Recursive estimation of mixed autoregressive-moving average order". *Biometrika*, 69.1, 81-94.

[Kwiatkowski] Kwiatkowski, D., Phillips, P.C.B., Schmidt, P., Shin, Y. 1992. "Testing the null hypothesis of stationarity against the alternative of a unit root", *Journal of Econometrics*, 54, 159ñ178

[Ljung] Ljung, G.M. and Box, G.P. 1978. "On a Measure of a Lack of Fit in Time Series Models", *Biometrika*, 65.2, 297–303.

[McKinney] McKinney, W., Perktold, J., Seabold, S. 2011. "Time series analysis in python with statsmodels", *Proceedings of the 10th Python in Science Conference*, 96-102.

[Pankratz] Pankratz, A. 1991. "Forecasting with dynamic regression models", John Wiley and Sons, New York.

[Said] Said, S.E. and Dickey, D. 1984. "Testing for unit roots in autoregressive moving-average models with unknown order", *Biometrika*, 71, 599-607.

[Seabold] Seabold, S. and Perktold J. 2010. "Statsmodels: econometric and statistical modeling with python", *Proceedings of the 9th Python in Science Conference*, 57-61.

[Tsay] Tsay, R.S. 2005. "Analysis of Financial Time Series", John Wiley & Sons, Inc., Hoboken, NJ.

[Tukey] Tukey, J.W. 1977. "Exploratory data analysis", Addison-Wesley, Reading, MA.

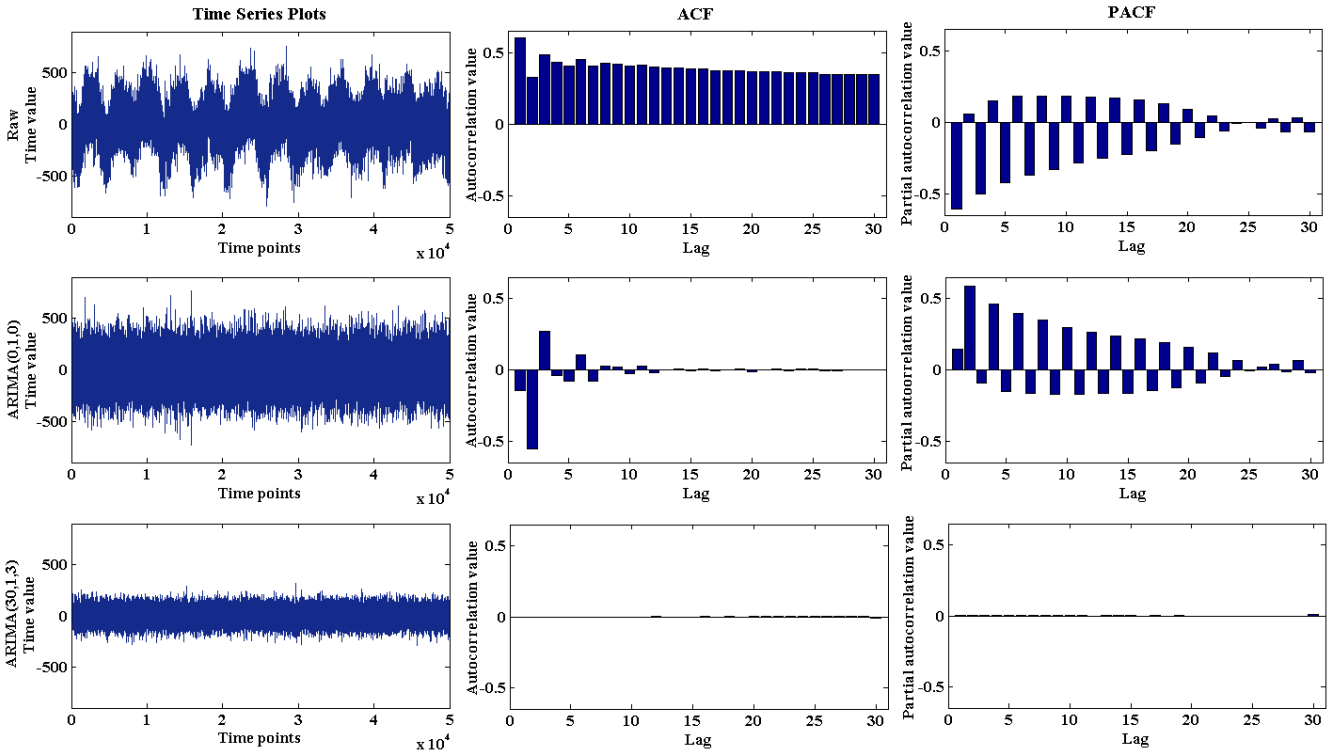


Fig. 6: *Raw, differenced, and ARIMA(30,1,3) series with corresponding ACF and PACF.*

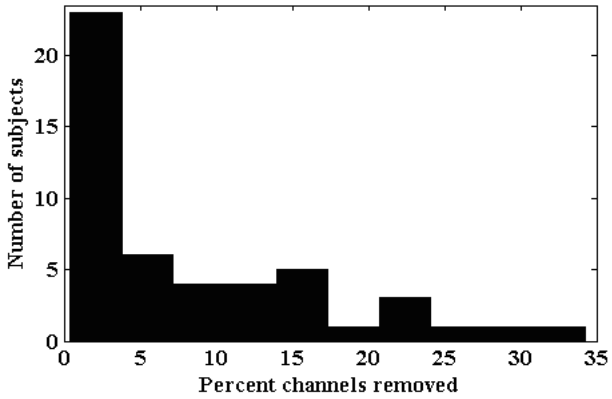


Fig. 7: *Percent of channels removed per subject.*

[Yaffee] Yaffee, R.A. and McGee, M. 2000. "Introduction to time series analysis and forecasting: with applications of SAS and SPSS", Academic Press.

Signal Processing and Communications: Teaching and Research Using IPython Notebook

Mark Wickert‡*

https://www.youtube.com/watch?v=xWREmn7EajM

Abstract—This paper will take the audience through the story of how an electrical and computer engineering faculty member has come to embrace Python, in particular IPython Notebook (IPython kernel for Jupyter), as an analysis and simulation tool for both teaching and research in signal processing and communications. Legacy tools such as MATLAB are well established (entrenched) in this discipline, but engineers need to be aware of alternatives, especially in the case of Python where there is such a vibrant community of developers. In this paper case studies will also be used to describe domain specific code modules that are being developed to support both lecture and lab oriented courses going through the conversion from MATLAB to Python. These modules in particular augment `scipy.signal` in a very positive way and enable rapid prototyping of communications and signal processing algorithms. Both student and industry team members in subcontract work, have responded favorably to the use of Python as an engineering problem solving platform. In teaching, IPython notebooks are used to augment lecture material with live calculations and simulations. These same notebooks are then placed on the course Web Site so students can download and *tinker* on their own. This activity also encourages learning more about the language core and Numpy, relative to MATLAB. The students quickly mature and are able to turn in homework solutions and complete computer simulation projects, all in the notebook. Rendering notebooks to PDF via LaTeX is also quite popular. The next step is to get other signals and systems faculty involved.

Index Terms—numerical computing, signal processing, communications systems, system modeling

Introduction

This journey into Python for electrical engineering problem solving began with the writing of the book *Signals and Systems for Dummies* [Wic2013], published summer 2013. This book features the use of Python (`Pylab`) to bring life to the mathematics behind signals and systems theory. Using Python in the Dummies book is done to make it easy for all readers of the book to develop their signals and system problem solving skills, without additional software tools investment. Additionally, the provided custom code module `ssd.py` [ssd], which is built on top of `numpy`, `matplotlib`, and `scipy.signal`, makes it easy to work and extend the examples found in the book. Engineers love to visualize their work with plots of various types. All of the plots in the book are created using Python, specifically `matplotlib`.

* *Corresponding author: mwickert@uccs.edu*
‡ *University of Colorado Colorado Springs*

The next phase of the journey, focuses on the research and development side of signals and systems work. During a recent sabbatical[1] Python and IPython notebook (IPython kernel for Jupyter) served as the primary digital signal processing modeling tools on three different projects. Initially it was not clear which tool would be best, but following discussions with co-workers[2] Python seemed to be the right choice. Note, for the most part, the analysis team was new to Python, all of us having spent many years using MATLAB/Octave [MATLAB]/[Octave]. A nice motivating factor is that Python is already in the workflow in the real-time DSP platform used by the company.

The third and current phase of the Python transformation began at the start of the 2014-2015 academic year. The move was made to push out Python to the students, via the IPython Notebook, in five courses: digital signal processing, digital communications, analog communications, statistical signal processing, and real-time signal processing. Four of the courses are traditional lecture format, while the fifth is very hands-on lab oriented, involving embedded systems programming and hardware interfacing. IPython Notebook works very well for writing lab reports, and easily allows theoretical and experimental results to be integrated. A notebook interface is not a new concept in scientific computing tool sets[3]. Both of these tools are very powerful for specific problem classes.

The remainder of this paper is organized into the following sections: arriving at Python for communications and signal processing modeling, describing IPython notebook usage, case studies, and conclusions.

Arriving at Python for Communications and Signal Processing Modeling

About three years ago while working on a study contract for a small business, I started investigating the use of open-source alternatives over MATLAB. I initially homed in on using Octave [Octave] for its syntax compatibility with MATLAB. Later I started to explore Python and became fascinated by the ease of use offered by the IPython (QT) console and the high quality of `matplotlib` 2D plots. The full power of Python/IPython

1. Academic year 2013-2014 was spent working for a small engineering firm, Cosmic AES.

2. Also holding the Ph.D. and/or MS in Electrical Engineering, with emphasis in communications and signal processing.

3. See for example *Mathematica* [Mathematica] (commercial) and *wxMaxima* [Maxima] (open source).

for engineering and scientific computing gradually took hold as I learned more about the language and the engineering problem capabilities offered by `pylab`.

When I took on the assignment of writing the *Signals and Systems for Dummies* book [Wic2013] Python seemed like a good choice because of the relative ease with which anyone could obtain the tools and then get hands-on experience with the numerical examples I was writing into the book. The power of `numpy` and the algorithms available in `scipy` are very useful in this discipline, but I immediately recognized that enhancements to `scipy.signal` are needed to make signals and systems tinkering user friendly. As examples were written for the book, I began to write support functions that fill in some of the missing details not found in `scipy`. This is the basis for the module `ssd.py`, a constant work in progress to make open source signals and systems software more accessible to the engineering community.

Modules Developed or Under Development

As already briefly mentioned, the first code module I developed is `ssd.py`[4]. This module contains 61 functions supporting *signal* generation, manipulation, and display, and *system* generation and characterization. Some of the functions implement subsystems such as a ten band audio equalization filter or the model of an automobile cruise control system. A pair of wrapper functions `to_wav()` and `from_wav()` make it easy for students to write and read 1D `ndarrays` from a wave file. Specialized plotting functions are present to make it easy to visualize both signals and systems. The collection of functions provides general support for both continuous and discrete-time signals and systems, as well as specific support for examples found in [Wic2013]. Most all of functions are geared toward undergraduate education. More modules have followed since then.

The second module developed, `digitalcom.py`, focuses on the special needs of digital communications, both *modulation* and *demodulation*. At present this module contains 32 functions. These functions are focused on waveform level simulation of contemporary digital communication systems. When I say simulation I specifically mean *Monte Carlo* techniques which involve the use of *random bit streams*, noise, channel fading, and interference. Knowledge of digital signal processing techniques plays a key role in implementation of these systems. The functions in this module are a combination of communcation waveform generators and specialized signal processing building blocks, such as the *upsampler* and *downsampler*, which allow the sampling rate of a signal to be raised or lowered, respectively. More functions are under development for this module, particularly in the area of orthogonal frequency division multiplexing (OFDM), the key modulation type found in the wireless telephony standard, long term evolution (LTE).

A third module, `fec_conv.py`, implements a rate 1/2 *convolutional encoding* and *decoding* class [Zie2015]. In digital communications digital information in the form of *bits* are sent from the transmitter to the receiver. The transmission channel might be wired or wireless, and the signal carrying the bits may be at *baseband*, as in say Ethernet, or *bandpass* on a *carrier frequency*, as in WiFi. To error protect bits sent over the channel *forward error correction* (FEC) coding, such as *convolutional codes*, may be employed. Encoding is applied before the source bits are modulated onto the carrier to form the transmitted signal. With a rate 1/2 convolutional code each source bit is encoded into two channel bits using a *shift register* of length K (termed *constraint length*) with *excusive or* logic gate connections. The class allows arbitrary constraint length codes and allows *puncturing* and *depuncturing* patterns. With pucturing/depuncturing certain code bits are *erased*, that is not sent, so as to increase the code rate from 1/2 to say 3/4 (4 channel bits for every three source bits).

For decoding the class implements the Viterbi algorithm (VA), which is a *dynamic programming* algorithm. The most likely path the received signal takes through a *trellis structure* is how the VA recovers the sent bits [Zie2015]. Here the *cost* of traversing a particular trellis branch is established using *soft decision metrics*, where soft decision refers to how information in the *demodulated* radio signal is converted metric values.

The class contains seven methods that include two graphical display functions, one of which shows the *survivor traceback paths* through the trellis back in time by the decoder *decision depth*. The traceback paths, one for each of the 2^{K-1} trellis states, give students insight into the operation of the VA. Besides the class, `fec_conv` also contains four functions for computing error probability bounds using the *weight structure* of the code under both *hard* and *soft* branch metric distance calculations [Zie2015].

A fourth module, `synchronization.py`, was developed while teaching a *phase-locked loops* course, Summer 2014. Synchronization is extremely important is all modern communications communications schemes. Digital communication systems fail to get data bits through a wireless link when synchronization fails. This module supplies eight simulation functions ranging from a basic phase-locked loop and both carrier and symbol synchronization functions for digital communications waveforms. This module is also utilized in an analog communications course taught Spring 2015.

Describing IPython Notebook Use Scenarios

In this section I describe how Python, and in particular the IPython notebook, has been integrated into teaching, graduate student research, and industry research and development.

Teaching

To put things into context, the present lecturing style for all courses I teach involves the use of a tablet PC, a data projector, a microphone, and audio/video screen capture software. Live Python demos are run in the notebook, and in many cases all the code is developed in real-time as questions come from the class. The notebook is more than just a visual experience. A case in point is the notebook audio control which adds sound playback capability. A 1D `ndarray` can be saved as a *wave file* for playback. Simply put, signals do make sounds and the action of systems changes what can be heard. Students enjoy hearing as well as seeing results. By interfacing the tablet *lineout* or *headphone* output to the podium interface to the classroom speakers, everyone can hear the impact of algorithm tweaks on what is being heard. This is where the fun starts! The modules `scipy.signal` and `ssd.py`, described earlier, are imported at the top of each notebook.

For each new chapter of lecture material I present on the tablet PC, a new IPython notebook is created to hold corresponding numerical analysis and simulation demos. When appropriate, starter content is added to the notebook before the lecture. For example I can provide relevant theory right in the notebook to

4. `http://www.eas.uccs.edu/wickert/SSD/docs/python/`

transition between the lecture notes mathematics and the notebook demos. Specifically, text and mathematics are placed in *markdown cells*. The notebook theory is however very brief compared to that of the course lecture notes. Preparing this content is easy, since the lecture notes are written in LaTeX I drop the selected equations right into mark down cells will minimal rework. Sample calculations and simulations, with corresponding plots, are often generated in advance, but the intent is to make parameter changes during the lecture, so the students can get a feel for how a particular math model relates to real-word communications and signal processing systems.

Computer projects benefit greatly from the use of the notebook, as sample notebooks with starter code are easily posted to the course Web Site. The sample notebook serves as a template for the project report document that the student will ultimately turn in for grading. The ability to convert the notebook to a LaTeX PDF document works for many students. Others used *screenshots* of selected notebook cells and pasted them into a word processor document. In Spring 2015 semester students turned in printed copies of the notebook and as backup, supplied also the notebook file. Marking on real paper documents is still my preference.

Graduate Student Research

In working with graduate students on their research, it is normal to exchange code developed by fellow graduate students working on related problems. Background discussions, code implementations of algorithms, and worked examples form a perfect use case for IPython notebook. The same approach holds for faculty interaction with their graduate students. In this scenario the faculty member, who is typically short on free time, gains a powerful advantage in that more than one student may need to brought up to speed on the same code base. Once the notebook is developed it is shared with one or more students and often demoed in front the student(s) on a lab or office computer. The ability to include figures means that system block diagrams can also be placed in the notebook.

As the student makes progress on a research task they document their work in a notebook. Faculty member(s) are briefed on the math models and simulation results. Since the notebook is live, hypothetical questions can be quickly tested and answered.

Industry Research and Development

With the notebook engineers working on the same team are able to share analytical models and development approaches using markdown cells. The inclusion of LaTeX markup is a welcome addition and furthers the establishment of notational conventions, during the development of signal processing algorithms.

Later, prototype algorithm development is started using code cells. Initially, computer synthesized signals (waveforms) are used to validated the core functionality of an algorithm. Next, signal captures (date files) from the actual real-time hardware are used as a source of test vectors to verify that performance metrics are being achieved. Notebooks can again be passed around to team members for further algorithm testing. Soon code cell functions can be moved to code modules and the code modules distributed to team members via `git` [git] or some other distributed revision control system. At every step of the way `matplotlib` [matpltlib] graphics are used to visualize performance of a particular algorithm, versus say a performance bound.

Complete subsystem testing at the Python level is the final step for pure Python implementations. When Python is used to construct a behavioral level model, then more testing will be required. In this second case the code is moved to a production environment and recoding to say C/C++. It might also be that the original Python model is simply an abstraction of real electronic hardware, in which case a hardware implementer uses the notebook (maybe just a PDF version) to create a hardware prototype, e.g., a *field programable gate array* (FPGA) or custom integrated circuit.

Live From the Classroom

Here live from the classroom means responding to questions using on-the-fly IPython notebook demos. This is an excellent way to show off the power of Python. Sometimes questions come and you feel like building a quick model right then and there during a lecture. When successful, this hopefully locks in a solid understanding of the concepts involved for the whole class. The fact that the lecture is being recorded means that students can recreate the same demo at their leisure when they watch the lecture video. The notebook is also saved and posted as a supplement/companion to the lecture. As mentioned earlier, there is a corresponding notebook for each chapter of lecture material[5]. I set the goal of re-posting the chapter notebooks each time a new lecture video is posted. This way the students have something to play with as they work on the current homework assignment.

Case Studies

In this section I present case studies that present the details on one or more of the IPython notebook use cases described in the previous section of this paper. Case studies from industry R&D are not included here due to the propriety nature of the work.

In all of the case studies you see that graphical results are produced using the `pylab` interface to `matplotlib`. This is done purposefully for two reasons. The first stems from the fact that currently all students have received exposure to MATLAB in a prior course, and secondly, I wish to augment, and not replace, the students' MATLAB knowledge since industry is still lagging when it comes to using open source tools.

Digital Signal Processing

As a simple starting point this first case study deals with the mathematical representation of signals. A step function sequence $u[n]$ is defined as

$$u[n] = \begin{cases} 1, & n \geq 0 \\ 0, & \text{otherwise} \end{cases} \tag{1}$$

Here I consider the difference between two step sequences starting at $n = 0$ and the other starting at $n = 5$. I thus construct in Python

$$x_3[n] = x_1[n] - x_2[n] = u[n] - u[n-5], \tag{2}$$

which forms a pulse sequence that *turns on* at $n = 0$ and *turns off* at $n = 5$. A screen capture from the IPython notebook is shown in Fig. 1.

Of special note in this case study is how the code syntax for the generation of the sequences follows closely the mathematical form. Note to save space the details of plotting $x_2[n]$ and $x_3[n]$ are omitted, but the code that generates and plots $x_3[n]$ is simply:

```
stem(n,x1 - x2)
```

5. Notebook postings for each course at `http://www.eas.uccs.edu/wickert/`

Notebook Screen Capture

Create Two Step Sequences

The module ssd (file ssd.py) contains the function `ssd.dstep(n)` which produces a step function output using the index vector `n` as the input that turns on at `n = 0`. If you input `n-5` the step will now turn on at `n=5`.

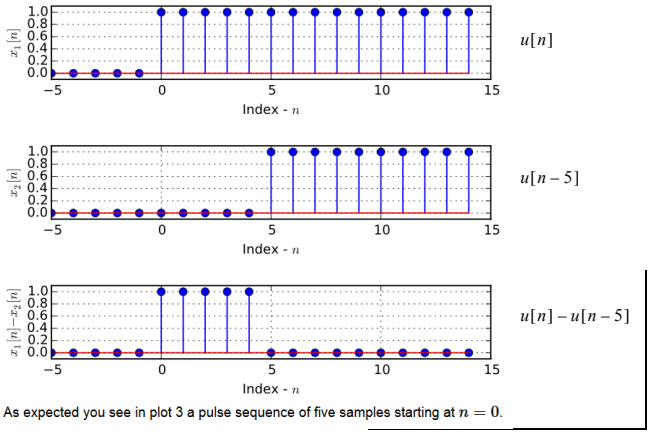
```
n = arange(-5, 15)
x1 = ssd.dstep(n) # step turns on at n = 0
x2 = ssd.dstep(n-5) # Step turns on at n = 5
```

Plot Waveforms using the Stem function

Create a 3x1 array subplots. The first two contain $x_1[n] = u[n]$ and $x_2[n] = u[n-5]$ respectively. The third plot is the difference of the first minus the second, i.e., $x_1[n] - x_2[n] = u[n] - u[n-5]$, which should be a rectangular pulse of duration five samples starting at $n = 0$.

```
figure(figsize=(6, 1.0))
stem(n, x1)
grid()
axis([-5, 15, -.1, 1.1])
xlabel(r'Index - $n$')
ylabel(r'$x_1[n]$')
```

(...Repeat for two more plots)



As expected you see in plot 3 a pulse sequence of five samples starting at $n = 0$.

Fig. 1: *Discrete-time signal generation and manipulation.*

Convolution Integral and LTI Systems

A fundamental signal processing result states that the signal output from a *linear* and *time invariant* (LTI) system is the *convolution* of the input signal with the system *impulse response*. The impulse response of a continuous-time LTI system is defined as the system output $h(t)$ in response to the input $\delta(t)$, where $\delta(t)$ is the *dirac delta function*. A block diagram of the system model is shown in Fig. 2.

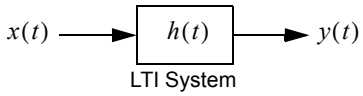


Fig. 2: *Simple one input one output LTI system block diagram.*

In mathematical terms the output $y(t)$ is the integral

$$y(t) = \int_{-\infty}^{\infty} h(\lambda)x(t-\lambda)\,d\lambda \tag{3}$$

Students frequently have problems setting up and evaluating the convolution integral, yet it is an important concept to learn. The waveforms of interest are typically piecewise continuous, so the integral must be evaluated over one or more contiguous intervals. Consider the case of $x(t) = u(t) - u(t-T)$, where $u(t)$ is the unit step function, and $h(t) = ae^{-at}u(t)$, where $a > 0$. To avoid careless errors I start with a sketch of the integrand $h(\lambda)x(t-\lambda)$, as shown in Fig. 3. From there I can discover the support intervals or *cases* for evaluating the integral.

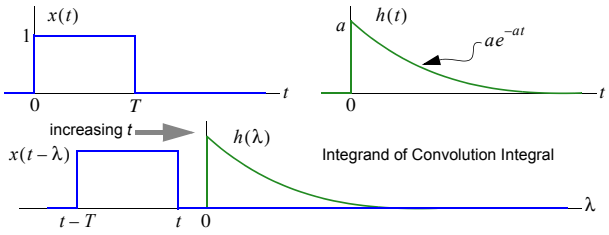


Fig. 3: *Sketches of $x(t)$, $h(t)$, and $h(\lambda)x(t-\lambda)$.*

A screen capture of a notebook that details the steps of solving the convolution integral is given in Fig. 4. In this same figure we see the analytical solution is easily plotted for the case of $T = 1$ and $a = 5$.

Notebook Screen Capture

Convolution Integral Simulation

For a continuous-time linear time invariant (LTI)system having impulse response $h(t)$ and input signal $x(t)$, the output, $y(t)$ can be written in terms of a convolution integral:

$$y(t) = \int_{-\infty}^{\infty} x(\lambda)h(t-\lambda)\,d\lambda = \int_{-\infty}^{\infty} h(\lambda)x(t-\lambda)\,d\lambda$$

Special Case

Consider $x(t) = u(t) - u(t-T)$ a rectangular pulse of duration Ts and $h(t) = ae^{-at}u(t)$ an exponential, where $a > 0$. **Note**: The impulse response is of the form of the well known RC lowpass filter if we let $a = 1/RC$.

Writing out and evaluating the convolution integral for the given $x(t)$ and $h(t)$ results in a piecewise solution involving three contiguous support intervals: (Case 1) $t < 0$, (Case 2) $0 \geq t < T$, and (Case 3) $t \geq T$. The integrand is zero for Case 1. Using the second form of the convolution integral, Case 2 evaluates to:

$$y(t) = \int_0^t ae^{-(t-\lambda)}\,d\lambda = -e^{-a\lambda}\Big|_0^t = 1 - e^{-at},\ 0 \leq t < T$$

For Case 3 we have

$$y(t) = \int_{t-T}^t ae^{-(t-\lambda)}\,d\lambda = -e^{-a\lambda}\Big|_{t-T}^t = e^{-a(t-T)}\left[1 - e^{-aT}\right],\ t \geq T$$

In summary:

$$y(t) = \begin{cases} 0, & t < 0 \\ 1 - e^{-at}, & 0 \leq t < T \\ e^{-a(t-T)}\left[1 - e^{-aT}\right], & t \geq T \end{cases}$$

Plot the piecewise solution:

```
# Let T = 1s and a = 5
figure(figsize=(6, 2))
T = 1; a = 5
tt = arange(-1, 3.001, .01)
yt = (1-exp(-a*tt))*(ssd.step(tt)-ssd.step(tt-T))\
     +exp(-a*(tt-T))*(1-exp(-a*T))*ssd.step(tt-T)
plot(tt, yt,'g')
```

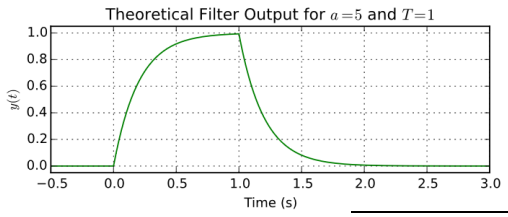


Fig. 4: *Solving the convolution integral in the notebook .*

To bring closure to the tedious analytical solution development, I encourage students check their work using computer simulation. The function `ssd.conv_integral()` performs numerical evaluation of the convolution integral for both finite and semi-infinite extent limits. I simply need to provide an array of signal/impulse response sample values over the complete support interval. The screen capture of Fig. 5 shows how this is done in a notebook. Parameter variation is also explored. Seeing the two approaches provide the same numerical values is rewarding and a powerful testimony to how the IPython notebook improves learning and understanding.

Notebook Screen Capture

Check on the Analytical Solution

```
# Let T = 1s and a = 1
figure(figsize=(6, 2))
a = 1
t = arange(-1, 3.001, .001)
x = ssd.step(t) - ssd.step(t-1)
h = a*exp(-a*t)*ssd.step(t)
y, ty = ssd.conv_integral(x, t, h, t)
plot(ty, y)
```

Generate $x(t)$ and $h(t)$ then numerically convolve with *scipy.signal.convolve* used in the core calculation

(...Repeat for two more plots with a = 5 and 10)

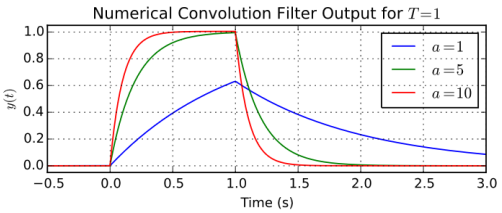


Fig. 5: *Plotting $y(t)$ for $a = 1, 5,$ and 10.*

Convolutional Coding for Digital Communications

In this case study the coding theory class contained in `fec_conv.py` is exercised. Here the specific case is taken from a final exam using a rate 1/2, $K = 5$ code. Fig. 6 shows the construction of a `fec_conv` object and a plot of one code symbol of the trellis.

Notebook Screen Capture

Part a: *K* = 5 Rate 1/2 Code

In this first part you will create a fec_conv object for the $K = 5$ code of Table 1. You will create a BEP plot similar to that found on page 7-41 of the Chapter 7 (text Chapter 12) notes. Note: You will need to increase D to about $5 \times 5 = 25$. Unlike the notes example, your results will be for soft-decision decoding. Functions for computing soft decision decoding upper bounds are contained in the module fec_conv.py. In addition to the BEP plot, also provide the trellis plot and a traceback plot under low and high SNR values.

```
In [5]: # Instantiate a fec_conv coder/decoder object
        cc1 = fec.fec_conv(('10011', '11101'), 25)
        cc1.trellis_plot()
```

Note: $2^{K-1} = 16$ states.
Blue transitions for '0' input bit
Green transitions for '1' input bit

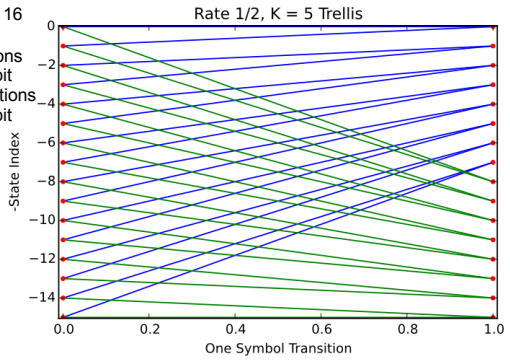


Fig. 6: *Construction of a `fec_conv` object and the corresponding trellis structure for the transmission of one code symbol.*

At the digital communications receiver the received signal is demodulated into *soft decision* channel bits. The soft values are used to calculate *branch metrics*, which then are used to update cumulative metrics held in each of the 16 states of the trellis. There are two possible paths arriving at each state, but the *surviving* path is the one producing the minimum cumulative metric.

Fig. 7 shows the survivor traceback paths in the 16-state trellis while sending random bits through the encoding/decoding process. Additive noise in the communications channel introduces confusion in the formation of the traceback paths. The channel *signal-to-noise ratio* (SNR), defined as the ratio of received signal

Notebook Screen Capture

High SNR Traceback Plot

```
cc1 = fec.fec_conv(('10011', '11101'), 25)
EbN0 = 7
# Create 1000 random 0/1 bits
x = randint(0, 2, 1000)
# Encode with shift register starting state of '0000'
state = '0000'
y, state = cc1.conv_encoder(x, state)
# Add channel noise to bits translated to +1/-1
yn = dc.cpx_AWGN(2*y-1, EbN0-3, 1)  # Channel SNR is dB less
# Translate noisy +1/-1 bits to soft values on [0, 7]
yn = (yn.real+1)/2*7
z = cc1.viterbi_decoder(yn)
# Look at the traceback in the VA trellis
cc1.traceback_plot()
```

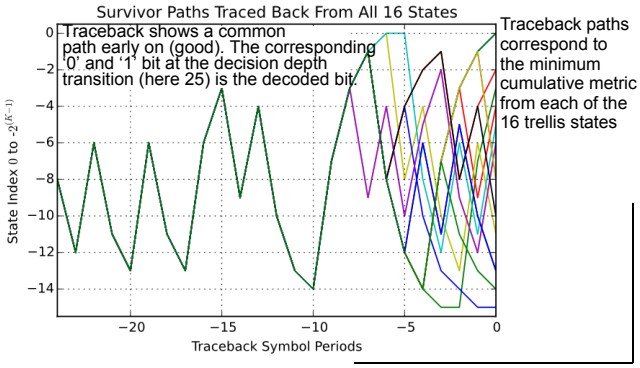


Fig. 7: *Passing random bits through the encoder/decoder and plotting an instance of the survivor paths.*

power to background noise power, sets the operating condition for the system. In Fig. 7 the SNR, equivalently denoted by E_b/N_0), is set at 7 dB. At a *decision depth* of 25 code symbols, all 16 paths merge to a common path, making it very likely that the probability of a bit error, is very very small. At lower a SNR, not shown here, the increased noise level makes it take longer to see a traceback merge and this is indicative of an increase in the probability of making a bit error.

Real-Time Digital Signal Processing

In the real-time digital signal processing (DSP) course C-code is written for an embedded processor. In this case the processor is an ARM Cortex-M4. The objective of this case study is to implement an equal-ripple *finite impulse response* (FIR) lowpass filter of prescribed amplitude response specifications. The filter is also LTI. Python (`scipy.signal`) is used to design the filter and obtain the filter coefficients, $b_1[n]$, $n = 0, \ldots, M$, in `float64` precision. Here the filter order turns out to be $M = 77$. As in the case of continuous-time LTI systems, the relation between the filter input and output again involves a convolution. Since a digital filter is a discrete-time system, the *convolution sum* now appears. Furthermore, for the LTI system of interest here, the convolution sum can be replaced by a *difference equation* representation:

$$y[n] = \sum_{k=0}^{M} x[n]b[n-k], \quad -\infty < n < \infty \tag{4}$$

In real-time DSP (4) becomes an algorithm running in real-time according to the system sampling rate clock. The processor is working with `int16` precision, so once the filter is designed the coefficients are scaled and rounded to 16 bit signed integers as shown in Fig. 8. The fixed-point filter coefficients are written to a C header file using a custom function defined in the notebook (not shown here).

Notebook Screen Capture

Remez Equi-Ripple Design

```
# Load an equi-ripple design code module from GNU Radio
#(in notebook ZIP package)
import optfir
```

```
d_pass = 0.2
d_stop = 60.0
fs = 48000
f_pass = 3500
f_stop = 5000
n, ff, aa, wts=optfir.remezord([f_pass, f_stop], [1, 0],
                               [1-10**(-d_pass/20), 10**(-d_stop/20)],
                               fsamp=48000)
# Bump up the order by 5 to bring down the final d_pass & d_stop
n_bump = n + 5
b1 = signal.remez(n_bump, ff, aa[0::2], wts, Hz=2)
```

Note: The original amplitude response requirements have been changed. The passband ripple is now 0.2 db and the passband critical frequency is reduced from 4000 to 3500 Hz. This reduces the filter order.

Fixed-Point Coefficients (`int16_t`)

```
b1_fix = int16(rint(b1*2**15))
b1_fix
```

```
array([   14,   -13,   -33,   -63,   -97,  -124,  -134,  -118,   -72,
          -2,    80,   151,   191,   179,   109,    -9,  -148,  -269,
        -332,  -304,  -176,    32,   274,   481,   581,   520,   281,
        -103,  -549,  -936, -1128, -1009,  -512,   360,  1519,  2810,
        4036,  4993,  5518,  5518,  4993,  4036,  2810,  1519,   360,
        -512, -1009, -1128,  -936,  -549,  -103,   281,   520,   581,
         481,   274,    32,  -176,  -304,  -332,  -269,  -148,    -9,
         109,   179,   191,   151,    80,    -2,   -72,  -118,  -134,
        -124,   -97,   -63,   -33,   -13,    14], dtype=int16)
```

```
FIR_fix_header('s4_p1_remez.h',b1)
```

Fig. 8: *Designing an equal-ripple lowpass filter using scipy.signal.remez for real-time operation.*

Notebook Screen Capture

Import Measured Results

```
fs_48, remez78_q15 = loadtxt('spec_noise_b_remez78_fs48arm_q15.csv',
                             delimiter=',', skiprows=1, usecols=(0, 1),
                             unpack=True)
```

```
figure(figsize=(6, 3))
f = arange(0, 1.0, .001)
w, B = signal.freqz(b1, 1, 2*pi*f)
w, Bq = signal.freqz(b1_fix, 1, 2*pi*f)
plot(f*48, 20*log10(abs(B)))
plot(f*48, 20*log10(abs(Bq)/sum(b1_fix)))
#plot(fs_48[:600]/1000, remez78_q15[:600]-mean(remez78_q15[:20]))
title(r'Equiripple Lowpass Theory: %d Taps' % n_bump)
ylabel(r'Filter Gain (dB)')
xlabel(r'Frequency in kHz ($f_s =$ %d kHz)' % (fs/1e3,))
legend((r'Theory float64', r'Theory int16'), loc='upper right')
ylim([-70, 2])
xlim([0, fs/1e3/2])
grid();
```

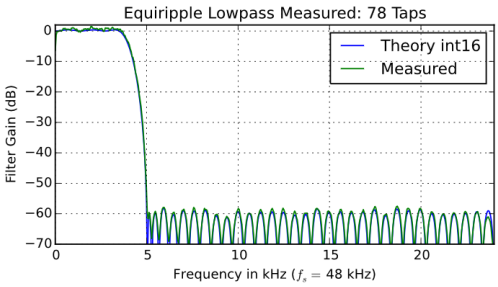


Fig. 9: *Comparing the theoretical fixed-point frequency response with the measured.*

The filter frequency response magnitude is obtained using a noise source to drive the filter input (first passing through an analog-to-digital converter) and then the filter output (following digital-to-analog conversion) is processed by instrumentation to obtain a spectral estimate. Here the output spectrum estimate corresponds to the filter frequency response. The measured frequency response is imported into the notebook using `loadtxt()`. Fig. 9 compares the theoretical frequency response, including quantization errors, with the measured response. The results compare favorably. Comparing theory with experiment is something students are frequently asked to do in lab courses. The fact that the stopband response is not quite equal-ripple is due to coefficient quantization. This is easy to show right in the notebook by overlaying the frequency response using the original `float64` coefficients `b1`, as obtained in Fig. 8, with the response obtained using the `b1_fix` coefficients as also obtained in Fig. 8 (the plot is not shown here).

An important property of the equal-ripple lowpass is that the filter coefficients, $b[n]$, have even symmetry. This means that $b_1[M-n] = b_1[n]$ for $0 \le n \le M$. Taking the z-transform of both sides of (4) using the convolution theorem [Opp2010] results in $Y(z) = H(z)X(z)$, where $Y(z)$ is the z-transform of $y[n]$, $X(z)$ is the z-transform of $x[n]$, and $H(z)$, known as the *system function*, is the z-transform of the system impulse response. The system function $H(z)$ takes the form

$$H(z) = \sum_{n=0}^{M} b_n z^{-n} \overset{\text{also}}{=} \frac{1}{z^M} \prod_{n=1}^{M} (z - z_n), \tag{5}$$

In general $H(z) = N(z)/D(z)$ is a rational function of z or z^{-1}. The roots of $N(z)$ are the system zeros and roots of $D(z)$ are the system poles. Students are taught that a *pole-zero* plot gives much insight into the frequency response of a system, in particular a filter. The module `ssd.py` provides the function `ssd.zplane(b,a)` where `b` contains the coefficients of $N(z)$ and `a` contains the coefficients of $D(z)$; in this case `a = [1]`. The even symmetry condition constrains the system zeros to lie at conjugate reciprocal locations [Opp2010] as seen in Fig. 10.

```
ssd.zplane(b1, [1], auto_scale=False, size=1.2)
```

```
(77, 0)
```

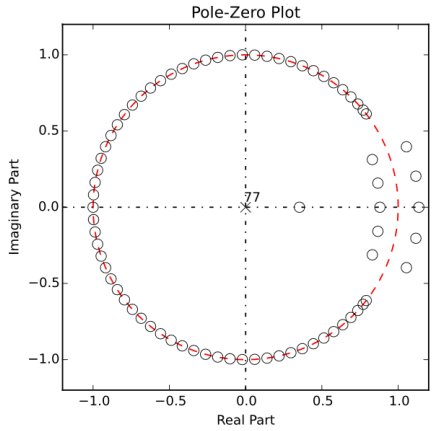


Fig. 10: *Pole-zero plot of the equal-ripple lowpass which confirms that $H(z)$ is linear phase.*

With real filter coefficients the zeros must also occur in conjugate pairs, or on the real axis. When the student sees the pole-zero plot of Fig. 10 whats jumps off the page is all of the zeros on the unit circle for the filter stopband. Zeros on the unit circle block signals from passing through the filter. Secondly, you see conjugate reciprocal zeros at angles over the interval $[-\pi/4, \pi/4]$ to define the filter passband, that is where signals pass through the filter. As a bit of trivia, zeros not on the unit circle or real axis **must** occur as quadruplets, and that is indeed what is seen in

Fig. 10. Note also there are 77 poles at $z = 0$, which is expected since $M = 77$. The pole-zero plot enhances the understanding to this symmetrical FIR filter.

Statistical Signal Processing

This case study is taken from a computer simulation project in a statistical signal processing course taken by graduate students. The problem involves the theoretical calculation of the probability density function of a random variable (RV) $\mathbf{w}$ where

$$\mathbf{w} = \mathbf{xy} + \mathbf{z}$$

is a function of the three RVs $\mathbf{x}$, $\mathbf{y}$, and $\mathbf{z}$. Forming a new RV that is a function of three RV as given here, requires some serious thinking. Having computer simulation tools available to check your work is a great comfort.

The screenshot of Fig. 11 explains the problem details, including the theoretical results written out as the piecewise function `pdf_proj1_w(w)`.

Notebook Screen Capture

Problem 1

The random variable w is defined in terms of the random variables x, y, and z, to be

$$\mathbf{w} = \mathbf{xy} + \mathbf{z}$$
$$= \mathbf{v} + \mathbf{z}$$

The input rv are assumed to be mutually independent, with $x\ U(-1,1)$, $y\ U(-1,1)$, and $z\ U(0,1)$.

Find the theoretical pdf . Start by first finding the pdf on $\mathbf{v} = \mathbf{xy}$ using the fact that for independent $\mathbf{x}$ and $\mathbf{y}$,

$$f_v(v) = \int_{-\infty}^{\infty} \frac{1}{w} f_x(w) f_y\left(\frac{v}{w}\right) dw$$
$$= \int_{-\infty}^{\infty} \frac{1}{w} f_y(w) f_x\left(\frac{v}{w}\right) dw$$

Then find the pdf on the sum $\mathbf{w} = \mathbf{v} + \mathbf{z}$ from a convolution. Note that the rv $\mathbf{v}$ and $\mathbf{z}$ are also independent. Why?

Theoretical Analysis

```
def pdf_proj1_w(w):
    """
    fw = pdf_proj1_w(w)
    Function plot the pdf of w = x*y + z where x~U(-1,1), y~U(-1,1), and
    z~U(0,1).

    Mark Wickert March 2015
    """

    fw = zeros_like(w)

    for k, wk in enumerate(w):
        if wk >= -1 and wk <= 0:
            fw[k] = -1/2*(wk*log(-wk)-wk-1)
        elif wk > 0 and wk <= 1:
            fw[k] = 1/2*(1 + (wk-1)*log(1-wk) - wk*log(wk))
        elif wk > 1 and wk <= 2:
            fw[k] = 1/2*(2 - wk + (wk-1)*log(wk-1))
        else:
            fw[k] = 0
    return fw
```

Fig. 11: *One function of three random variables simulation problem.*

Setting up the integrals is tedious and students are timid about pushing forward with the calculus. To build confidence a simulation is constructed and the results are compared with theory in Fig. 12.

Conclusions and Future Work

Communications and signal processing, as a discipline that sits inside electrical computer engineering, is built on a strong mathematical modeling foundation. Undergraduate engineering students, despite having taken many mathematics courses, are often intimidated by the math they find in communications and signals processing course work. I cannot make the math go away, but good modeling tools make learning and problem solving fun and exciting. I have found, and hopefully this paper shows, that

Notebook Screen Capture

Simulation

Create the Random Variates

```
x = 2*rand(1000000, 1)-1
y = 2*rand(1000000, 1)-1
z = rand(1000000, 1)
w = x*y + z
v = x*y
```

```
figure(figsize=(6, 3))
hist(w, 51, (-1, 2), normed=True, cumulative=False);
xlabel(r'$w$')
ylabel(r'Probability Density $f_w(w)$')
title(r'Probability Density of $w = xy + z$')
wr = arange(-1.2, 2.2, .001)
plot(wr, pdf_proj1_w(wr), 'r')
grid();
```

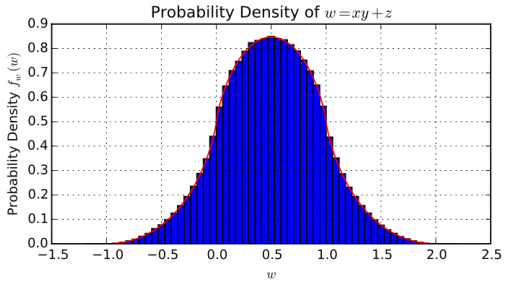


Fig. 12: *The simulation of random variable* $\mathbf{w}$ *and the a comparison plot of theory versus a scaled histogram.*

IPython notebooks are valuable mathematical modeling tools. The case studies show that IPython notebook offers a means for students of all levels to explore and gain understanding of difficult engineering concepts.

The use of open-source software is increasing and cannot be overlooked in higher education. Python is readily accessible by anyone. It is easy to share libraries and notebooks to foster improved communication between students and faculty members; between researchers, engineers, and collaborators. IPython and the IPython notebook stand out in large part due to the enthusiasm of the scientific Python developer community.

What lies ahead is exciting. What comes to mind immediately is getting other faculty on-board. I am optimistic and look forward to this challenge as tutorial sessions are planned over summer 2015. Other future work avenues I see are working on more code modules as well as enhancements to the existing modules. In particular in the convolutional coding class both the encoder and especially the Viterbi decoder, are numerically intensive. Speed enhancements, perhaps using *Cython*, are on the list of things to do. Within the notebook I am anxious to experiment with notebook controls/widgets so as to provide dynamic interactivity to classroom demos.

Acknowledgments

The author wishes to thank the reviewers for their helpful comments on improving the quality of this paper.

References

[Wic2013] M.A. Wickert. *Signals and Systems for Dummies*, Wiley, 2013.
[ssd] http://www.eas.uccs.edu/wickert/SSD/.
[MATLAB] http://www.mathworks.com/.
[Octave] https://en.wikipedia.org/wiki/GNU_Octave.
[Mathematica] https://en.wikipedia.org/wiki/Mathematica.
[Maxima] http://andrejv.github.io/wxmaxima/.

[Zie2015] R.E. Ziemer and W.H. Tranter *Principles of Communications*, seventh edition, Wiley, 2015.

[git] `https://git-scm.com/`

[matpltlib] `http://matplotlib.org/`

[Opp2010] Alan V. Oppenheim and Ronald W. Schafer, *Discrete-Time Signal Processing* (3rd ed.), Prentice Hall, 2010.

pyDEM: Global Digital Elevation Model Analysis

Mattheus P. Ueckermann‡*, Robert D. Chambers‡, Christopher A. Brooks‡, William E. Audette III‡, Jerry Bieszczad‡

https://www.youtube.com/watch?v=bGulPZh_-Mo

Abstract—Hydrological terrain analysis is important for applications such as environmental resource, agriculture, and flood risk management. It is based on processing of high-resolution, tiled digital elevation model (DEM) data for geographic regions of interest. A major challenge in global hydrological terrain analysis is addressing cross-tile dependencies that arise from the tiled nature of the underlying DEM data, which is too large to hold in memory as a single array. We are not aware of existing tools that can accurately and efficiently perform global terrain analysis within current memory and computational constraints. We solved this problem by implementing a new algorithm in Python, which uses a simple but robust file-based locking mechanism to coordinate the work flow between an arbitrary number of independent processes operating on separate DEM tiles.

We used this system to analyze the conterminous US's terrain at 1 arc-second resolution in under 3 days on a single compute node, and global terrain at 3 arc-second resolution in under 4 days. Our solution is implemented and made available as pyDEM, an open source Python/Cython library that enables global geospatial terrain analysis. We will describe our algorithm for calculating various terrain analysis parameters of interest, our file-based locking mechanism to coordinate the work between processors, and optimization using Cython. We will demonstrate pyDEM on a few example test cases, as well as real DEM data.

Index Terms—digital elevation model, hydrology, terrain analysis, topographic wetness index

Introduction

The aspect (or flow direction), magnitude of the slope, upstream contributing area (UCA), and topographic wetness index (TWI), shown in Figure 1, are important quantities in hydrological terrain analysis. These quantities are used to determine, for example, the flow path of water, sediment, and pollutants for applications in environmental resource, agricultural, and flood risk management. These quantities are calculated from gridded digital elevation models (DEM), which describe the topography of a region. DEMs are often stored as raster arrays, where the value of an element in the array gives the elevation of that point (usually in meters). The (i, j) coordinates of the array are also related to (latitude, longitude) coordinates through a geotransform. The aspect is calculated from DEM data and gives the angle (in radians) at each element. The aspect is important for determining the direction that water will flow, and is also important for solar radiation (for example, a north-facing slope is more shaded than a south-facing slope in the northern hemisphere). The slope (meters / meters) can also be calculated, and gives the change in elevation over the change in horizontal distance, quantifying the steepness of the topography. UCA captures the effect of water draining down a slope along particular routes by keeping track of the amount of runoff that is funneled through a point. UCA is defined as the total horizontal area that is up-slope of a point or contour [moore91], and unlike aspect and slope, the UCA at each element depends on more than just the immediately surrounding, or adjacent elements in the array. TWI (κ) is derived from UCA (a) and slope ($\tan\beta$), where $\kappa = \ln\frac{a}{tan\beta}$, and was developed by Beven and Kirkby [beven79] within the runoff model TOPMODEL (see [beven95]). TWI represents the steady-state soil moisture due to topographic effects. Regions with large TWI (flat slopes with large UCA) are generally wetter than regions with small TWI (steep slopes and small UCA).

With the improving availability of large, high quality DEM data, the hydrology of increasingly large regions can be analyzed. For example, in the US, the National Elevation Dataset provides 1×1 arc second (approximately 30×30 m) resolution DEM data for the conterminous US. This data is available as over 3500 files spanning $1^o \times 1^o$ latitude $\times$ longitude with over 3600×3600 pixels per file for 45 gigapixels of data. Analyzing such large data-sets presents unique challenges including:

- Accurately handling grid projections where the data is non-uniformly spaced
- Robustly dealing with no-data and flat regions
- Efficiently calculating the required quantities
- Breaking data up into computable tiles and dealing with the resulting edge effects (see Figure 2).

DEM datasets can be supplied in various coordinate frames where the data is not uniformly spaced. Over large regions and higher latitudes, assuming that the data is uniformly spaced can result in large errors, and accurate algorithms need to take into account this grid anisotropy. Additionally, DEM data often contains no-data values where the elevation could not be determined because of noise in the raw data or other effects. DEM data can also contain regions where the elevation appears to be flat, that is, there is no change in elevation from one pixel to the next. In that case, the aspect is not defined, and the slope is zero, which leads to an undefined TWI. These situations need to be dealt with robustly in order to successfully and automatically process large data-sets. The size of these data-sets can also make analysis intractable because of limited computational resources and slow algorithms. Finally, the discrete nature of the tiles can result in edge effects. Figure 2 shows an example of UCA calculated with

* Corresponding author: mpu@creare.com
‡ Creare LLC, Hanover, NH

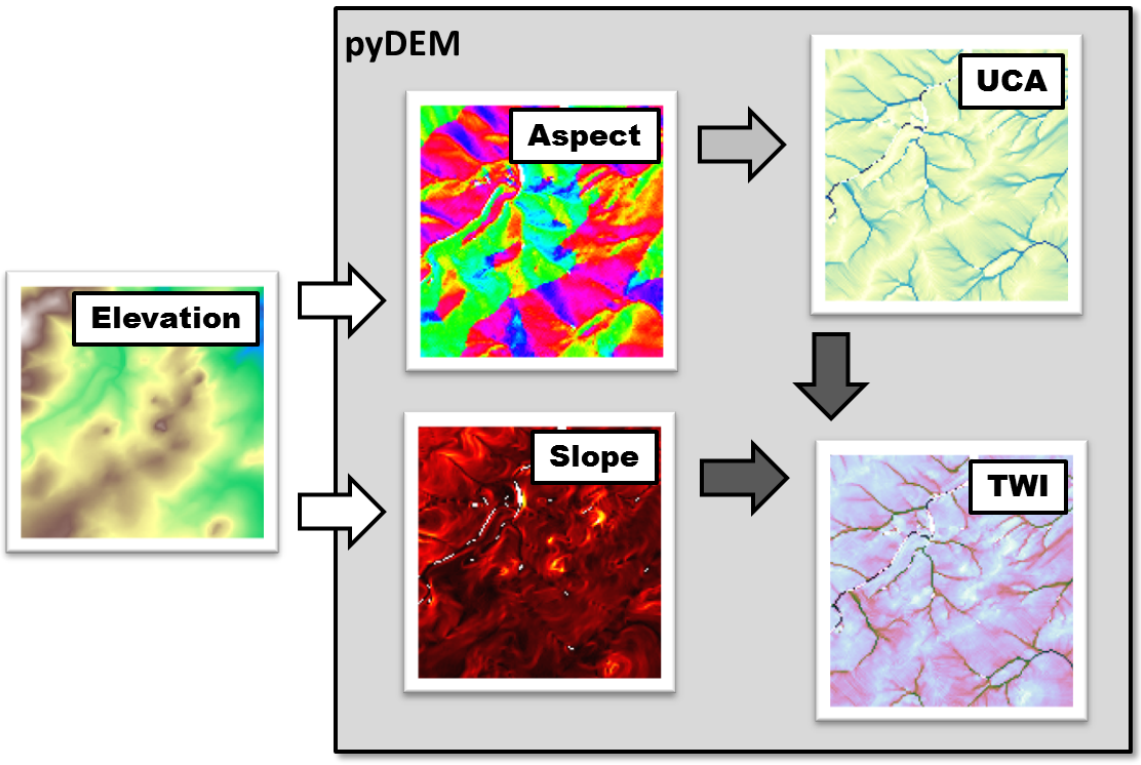


Fig. 1: *pyDEM calculates the aspect, slope, upstream contributing area (UCA), and topographic wetness index (TWI) from digital elevation model data. The aspect and slope are calculated directly from the elevation data, the UCA is calculated from the aspect, and the TWI is calculated from the UCA and the slope.*

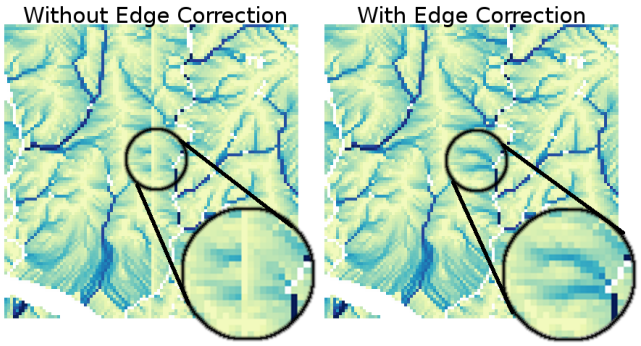


Fig. 2: *pyDEM can correctly follow the UCA calculation across tile boundaries.*

and without edge correction, where the edge artifact is visible as a vertical line.

pyDEM was developed to address these challenges. pyDEM is an open source Python/Cython library that has been used to calculate TWI for the conterminuous US at 30m resolution, and the globe at 90m resolution. In the following sections we will describe our new algorithm for calculating UCA, our file-based locking mechanism to coordinate work between processors, and optimization using Cython. Using pyDEM, we will then show TWI calculated using test elevations, and realistic elevations from the National Elevation Dataset.

Algorithm Design

To calculate the aspect and slope, pyDEM uses the D∞ method [tarboton97]. This method calculates the aspect and slope based on an 8-point stencil around a pixel. The UCA is calculated from the aspect, and it requires more than just an 8-point stencil around a pixel. In Tarboton 1997, a recursive algorithm to calculate the UCA is also presented, but we developed a new algorithm that handles no-data and flat areas differently, while also allowing area updates based on new information at edges of the tile. The recursive algorithm [tarboton97] starts at down-slope pixels and recursively calculates its up-slope area. Our algorithm follows the opposite strategy, and starts at up-slope pixels, then progressively calculates the UCA of down-slope pixels. Next, we will describe the main data-structure used for our approach, then present pseudo-code for the basic algorithm, describe modifications needed to update edges, and explain modifications to deal with flats.

Data Structures: The main data-structure used by the UCA algorithm is an adjacency (or connectivity) matrix, $\mathbf{A}$. For the example in Figure 3 (top), we have a 3×3 elevation array with a total of 9 pixels. Each row in matrix $\mathbf{A}$ (Figure 3, bottom) represents a pixel in the raster array that receives an area contribution from another pixel. The columns represents the pixels that drain into a pixel represented as a row. The value in row i column j represent the fraction of pixel j's area that drains into pixel i. For example, pixel 6 drains completely into pixel 7, so $\mathbf{A}_{7,6} = 1.0$. On the other hand, only 30% of the area in pixel 0 drains into pixel 3, so $\mathbf{A}_{3,0} = 0.3$.

The algorithm also requires a data structure: *ac_pix* to keep track of the "active pixels" which can be computed, *ac_pix_old* to record which pixels were computed last round, *done* to mark which pixels have finished their computations, and *uca* to contain the UCA for each pixel. The *ac_pix* vector is initialized by summing over the columns of $\mathbf{A}$ to select pixels that do not receive an area contribution from another pixel. This would happen for

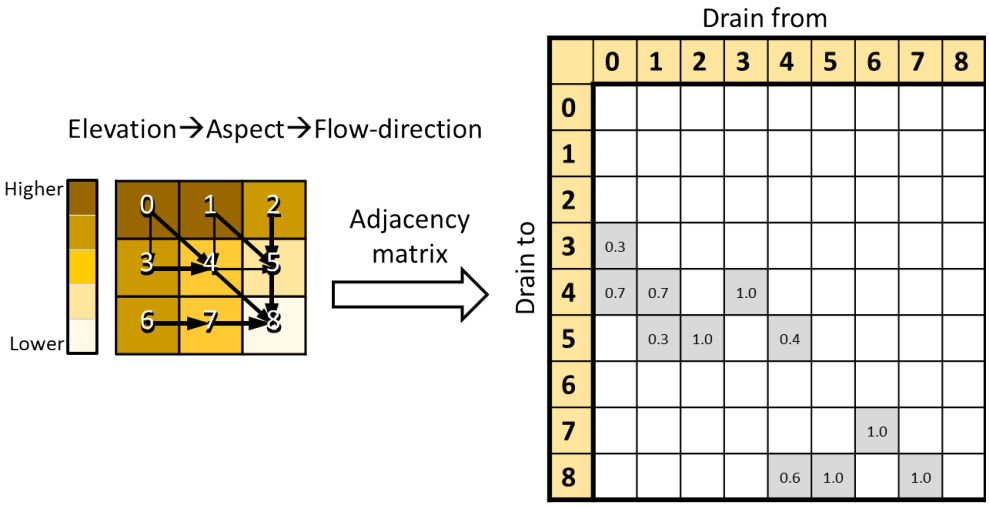


Fig. 3: *The UCA calculation takes a raster of elevation data (top) and constructs an adjacency (or connectivity) matrix.*

pixels at the top of mountains or hills, where the surrounding elevation is lower, and on pixels on the edges of tiles that do not receive contributions from the interior. The indices *ac_pix* of these pixels are stored in a boolean array.

Algorithm: The pseudo-code for our algorithm is given below using Python syntax. Lines 1-5 initialize the working data-structures, and assumes that the adjacency matrix was constructed and *elevation_data* is an array with the shape of the raster DEM data. The UCA should be initialized with the geographic area of a tile, but for simplicity consider $1m \times 1m$ pixels. The calculation is iterative and the exit condition on line 7 ensures that the loop will terminate, even if there are circular dependencies. Circular dependencies should not occur for usual DEM data, but for robustness (in the face of randomly distributed no-data values) this exit condition was chosen.

If a pixel is marked as active, its area will be distributed down-slope, executing lines 15-25. The column of the active pixel is looped over, and the fraction of the area in each row is distributed to the pixel in that row of the adjacency matrix. For example, in Figure 3, pixel 0 will be marked as active in the first loop (sum of elements in the row is zero). Line 17 will then update *uca[3]* and *uca[4]* with *f=0.3* and *0.7* times the area in pixel 0, respectively.

Next, lines 21-25 will check to see if the pixel just drained into is ready to become active. A pixel is allowed to become active once it has received all of its upstream area contributions. This condition for becoming active is crucial for preventing double-accounting. Continuing the example, once *uca[3]* was updated with the contribution from pixel 0, we will loop through the entries of **A** in row 3. If the entry is non-zero and not marked as done, we know that pixel 3 will receive a contribution from an upstream pixel in a later round. In our example, pixel 0 is the only upstream pixel for pixel 3, and it is done. So, in the next round we can drain from pixel 3.

In the worst case, this algorithm appears to be $\mathcal{O}(n^4)$, where n is the number of elements in the DEM array. Each of the loops, the *while* and three *for* loops all could be executed n times. In practice, the sparsity of **A** can be exploited to obtain an algorithm close to $\mathcal{O}(n)$ (see the optimization section).

```
1 # Initialize
2 ac_pix = A.sum(1) == 0
3 ac_pix_old = zeros_like(ac_pix)
4 done = zeros_like(ac_pix)
5 uca = ones(elevation_data.shape)  # Approximately
6
7 while any(ac_pix != ac_pix_old):
8       done[ac_pix] = True
9       ac_pix_old = ac_pix.copy()
10      ac_pix[:] = False
11
12      for i in range(ac_pix.size):
13          if ac_pix[i] is False:
14              continue  # to next i. Otherwise...
15          for j, f in enumerate(A[:, i]):
16              # update area
17              uca[j] += uca[i] * f
18
19              # Determine if pixel is done
20              for k, f2 in enumerate(A[j, :]):
21                  if not done[k] and f2:
22                      break
23              else:
24                  # Drain this pixel next round
25                  ac_pix[j] = 1
```

Modification for Edges Update: A fortunate aspect of the UCA calculation is its linearity, which lends itself well to the principle of superposition. That is, the UCA within a tile can be calculated and later adjusted with new contributions from the edges. In our Figure 3 example, we have a single DEM tile, but this tile might be one of many tiles. Considering only this one tile, we can calculate pixel 0's area contribution to the other pixels within a tile, but we do not know if pixel 0 is on a ridge, or if there is another pixel that would drain into it from another tile in the data-set. Similarly, pixel 8 might need to drain its area downstream to pixels in a downstream tile in the data-set. Ultimately, there will be a tile that has the most up-slope pixel, which has no edge dependencies. Similarly, for realistic data, the UCA of most pixels within a tile does not depend on the edge. Consider Figure 2 which shows that the difference in UCA between the tiles does not extend far past the edge, which indicates that the UCA calculation is relatively local, except for rivers. This means that the edge update can be efficient: we only have to update pixels near the edges, and rivers. Since rivers have a proportionally much smaller area, the edge update requires much fewer computations compared to the initial UCA calculation for a tile.

Our strategy of starting at the up-slope pixels and contributing

area to down-slope pixels is a key algorithmic choice to allow for the edge correction. Edge pixels that receive area contributions from neighboring tiles always need to distribute that area down-slope. It may be possible for every interior pixel to calculate and store its edge dependencies using the recursive strategy that starts at down-slope pixels, but in the worst case, each of these pixels will need to store its dependency on every edge pixel. This results in a large storage structure, or a complex one that compresses the information. Alternatively, every pixel will need to be re-calculated for every edge correction. With our strategy of starting with up-slope pixels, only the interior pixels that are affected by information from the edge needs to be recalculated.

To handle edges, the major modifications to the basic algorithm are: initializing the active pixels (*ac_pix*) based on edge information/dependencies, initializing the *done* pixels, and adding data-structures to keep track of edge dependencies. The main challenge is careful bookkeeping to ensure that edge information is communicated to neighboring tiles. pyDEM does this bookkeeping both within a tile, which can be broken up into multiple chunks, and across tiles, which is described in greater detail under the *Parallel Processing* section.

Modification for Flats: pyDEM considers no-data regions to also be flats. To handle flats, a small adjustment is made to the adjacency matrix. Without modification, the adjacency matrix will allow pixels next to flats to drain their area contributions into the flat, but these contributions never leave. The adjacency matrix is adjusted by adding the black and green arrows depicted in Figure 4. The total area contributions that drain into a flat are collected, for convenience, at a random point within the flat (black arrows). This total area contribution to the flat is then proportionally distributed to pixels at the edge of a flat. The proportions depend on the difference in elevation of the pixels around the flat. The pixel with the lowest elevation around the flat always receives a distribution. If a pixel's elevation satisfies $e_{local} < \min(\vec{e}_{local}) + \sqrt{2}\Delta x$, where e_{local} is the pixel's elevation, $\vec{e}_{local}$ are the elevations of the pixels around the flat and Δx is the approximate grid spacing, then it is also included in the area distribution. This relationship comes from a Taylor series expansion of the grid discretization error, and the $\sqrt{2}$ appears because the maximum error occurs along the diagonal direction. The proportion of the distribution is calculated as $p = \frac{e_{flat} - \vec{e}_{local}}{\sum e_{flat} - \vec{e}_{local}}$, where e_{flat} is the elevation of the flat. This distributes the UCA evenly to pixels with the same elevation surrounding the flat, or slightly more to pixels with a lower elevation (within the calculated error tolerance).

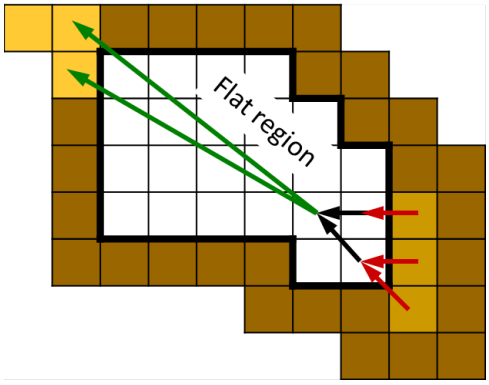


Fig. 4: *To correctly calculate drainage over flat or no-data regions, the total area that drains into the flat (bottom red arrows) are collected at a single point within the flat (middle black arrows) and then redistributed to lower-lying regions (top green arrows).*

Parallel Processing

The majority of the processing on a tile can be done independent of every other tile. This means it is simple to spawn multiple processes on a machine or cluster to churn through a large number of elevation tiles. There are various packages that automate this process. However, in our case, the edge correction step cannot be done efficiently on a tile-by-tile basis, so existing packages did not meet our needs.

The calculation proceeds in three stages. In the first stage, the tile-local quantities, aspect and slope, are calculated in parallel. Then the first pass UCA calculation is performed in parallel, where the initial edge data is written to files. Finally, the UCA is corrected in parallel to eliminate edge effects. This final stage does have an order-dependency, and the parallelism is not as efficient. In each of these stages, separate pyDEM processes can be launched. If a process terminates unexpectedly, it does not affect the remaining processes.

In order to prevent multiple processes from working on the same file, a simple file locking mechanism is used. When a process claims a DEM tile, it creates an empty .lck file with the same name as the elevation file. Subsequent processes will then skip over this file and sequentially process the next available DEM tile. Once a process is finished with a DEM tile, the .lck file is removed. Subsequent processes also check to see if the outputs are already present, in which case it will also skip that DEM tile, moving on to the next available file. This works well for the first two stages of the processing, although future implementations plan to use a cross-platform operating-system-level file locking package such as *lockfile*.

In the second and third stages, numpy's *.npz* format is used to save files which communicate edge information. The following three files are saved for every edge of a tile after calculating the UCA:

1) the current UCA value at each pixel on the edge,
2) whether the UCA calculation on the edge pixel is *done*, and does not still depend on information from neighboring tiles,
3) whether the edge needs to receive information from neighboring tiles and has not yet received it.

The first two quantities are populated by neighboring tiles, while the last quantity is self-calculated by a tile. That is, after calculating the UCA, a tile will set the pixel value and whether an edge is *done* on its neighbors, and update whether an edge needs information on its own edge data file. To explain why this is needed, the second and third quantities are illustrated in Figure 5. The first row represents three DEM tiles with edges in the state after the second calculation stage. The left tile is at the top of a hill, and all of its edges contribute area downstream. This tile does not expect any information to enter from adjacent tiles, so it sets the "to do" value (third quantity) on its edges as False. The left tile also communicates to the middle tile that this edge is "done" and can be calculated next round. Still on the top row, the middle tile determines that area will enter from the left edge, and sets the "to do" value on its left edge as True. Following this dependency along, it calculates that none of its other edges are

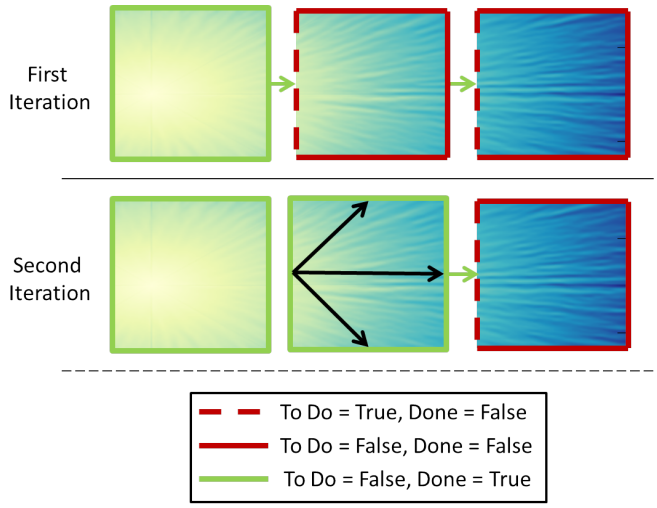


Fig. 5: *To correct edges across DEM tiles, the edge information is communicated to neighboring tiles, which can then update UCA internally to its edges and communicate that information to the next tile.*

done, and communicates this to the tile on the right. The second row in Figure 5 shows what happens during the first round of stage 3. In the first round, the middle tile is selected and the UCA is updated. Since it received finished edge data from the left tile, it now marks the left edge's "to do" status as False, and propagates the updated area through the tile. It communicates this information to the right tile, which will be updated in subsequent rounds in the stage 3 calculation. Note that the calculation on the right tile could not proceed until the left tile was calculated, which means that this computation had to be performed serially and could not be paralellized.

In the example illustrated in Figure 5, the middle tile only needed one correction. However, in general a tile may require multiple corrections. This can happen when a river meanders between two tiles, crossing the tile edge multiple times. In this case, the two adjacent tiles will be updated sequentially and multiple times to fully correct the UCA. This situation is specifically tested in the bottom left (c-1) test-case in Figure 6. There the water flow path spirals across multiple tiles multiple times. At each crossing, the UCA needs to be corrected.

During each round of the second stage, we heuristically select the *best* tile to correct first. This *best* tile is selected by looking at what percentage of edge pixels on that tile will be done after the correction. In the case of ties, the tile with the higher maximum elevation is used. In case another process is already using that tile, the next best tile is selected. As such, the calculation proceeds in a semi-parallel fashion for large data-sets.

Optimization

The first implementation of the UCA algorithm was much more vectorized than the code presented above. This pure-Python vectorized version aimed to take advantage of the underlying libraries used by numpy and scipy. However, this earlier version of the algorithm was not efficient enough to analyze a large data-set using a single compute node. The analysis would have taken over a year using 32 CPU cores.

Initial attempts to re-write the algorithm in Cython were not fruitful, only yielding minor speed improvements. The primary issue causing the poor performance was the adjacency matrix $\mathbf{A}$. This matrix was stored as a sparse array, because it had very few entries. The initial Python and Cython implementations used scipy's underlying sparse matrix implementation, along with linear algebra operations to perform the calculations. These implementations failed to use the underlying sparse matrix storage structure to their full advantage.

Consequently, we re-implemented the algorithm with the adjacency matrix was stored in both the Compressed Sparse Column (CSC) and Compressed Sparse Row (CSR) formats. The CSC format stores three arrays: *data*, *row_ind*, and *col_ptr*. The *data* stores the actual floating point values of the elements in the array, while the *row_ind* stores the row number of the data in each column (same size as data), and *col_ptr* stores the locations in the data vector that start a new column (size is 1 + the number of columns, where the last entry in col_ptr is the total number of data elements). For example, the $\mathbf{A}$ in Figure 3 is stored in CSC as:

$$\begin{aligned} data &= [0.3, 0.7, 1.0, 1.0, 1.0, 0.4, 0.6, 1.0, 1.0, 1.0] \\ row\_ind &= [3, \ 4, \ 4, \ 5, \ 4, \ 5, \ 8, \ 8, \ 7, \ 8] \\ col\_ptr &= [0, 2, 3, 4, 5, 7, 8, 9, 10, 10] \end{aligned}$$

The CSR format, which stores col_ind, row_ptr, and a re-arranged data vector instead, is more computationally efficient for some aspects of the algorithm, which is why both formats are used.

In particular, looping over the rows for a specific column in $\mathbf{A}$ to update the UCA (lines 15-17 of algorithm) can be efficiently done using the CSC format. Determining if a pixel is done, which loops over the columns for a specific row in $\mathbf{A}$ (lines 19-25) can be efficiently done using the CSR format.

Nested *for* loops in Python are generally known to be inefficient and was not implemented. The Cython implementation yielded excellent results, giving approximately a $3\times$ speed-up for smaller problems, and a $1000\times$ speedup for larger problems. These numbers are approximate because the actual values are highly dependent on the DEM data.

The computational complexity for this improved implementation is $\mathcal{O}(n)$. The *for* loop on line 12 will continue past lines 13-14 only n times, regardless of how many times the *while* loop is executed. Since each pixel can only drain to two neighbors, the *for* loop in line 15 only loops over 2 elements when using CSC. The *for* loop in line 20 only loops over a maximum of 8 elements for non-flats (since a pixel can only receive contributions from 8 neighboring pixels) when using CSR. While additional optimization is potentially possible, the present implementation efficiently computes the UCA.

Applications

To verify that pyDEM's core algorithms work as expected, a collection of test cases were created, and a subset is shown in Figure 6. pyDEM was also used to calculate TWI for the conterminous US. Next we will describe the purpose and results of the each of the test cases, and then we will present the results over the conterminous US.

To ensure that the [tarboton97] $D\infty$ method was correctly implemented, we created a number of linearly sloping elevations to test each quadrant of the 8-element stencil used for the slope and magnitude calculation (Figure 6 a-1, b-1, b-2). All of the possible angles are tested in the a-3 case. Notice that the TWI is higher along the diagonals of this case, and this is an artifact of the $D\infty$ method which is expected to be small for real DEM data. The

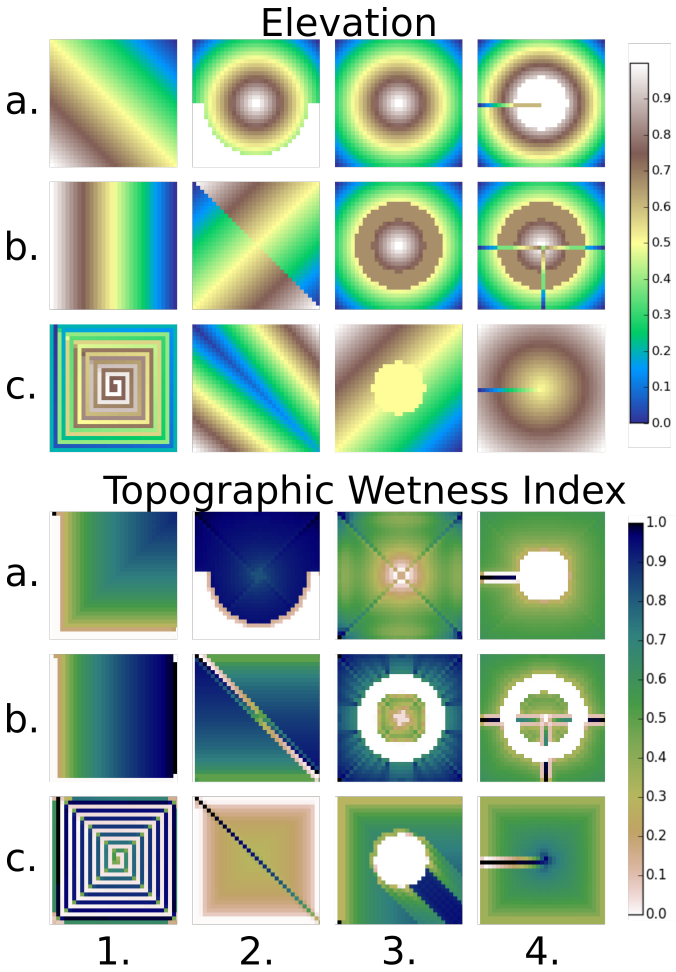


Fig. 6: *To verify that pyDEM's core algorithms work as expected, a collected of test elevations (top) were created to cover anticipated issues in calculating TWI (bottom). This shows that TWI is correctly calculated. In particular, TWI is larger where the elevation is lower (as expected), it is evenly distributed around flats (2nd and 3rd rows, 3rd column), and it is concentrated in rivers or outlets (4th column).*

c-2 case is a trough that tests to make sure that water will drain along the diagonal, which would not happen if a central difference method was used instead of the $D\infty$ method. The a-2 case tests if pyDEM correctly handles no-data values along the edge of a tile. Cases b-3, c-3, and those in column 4 all test pyDEM's handling of flat regions. In case b-3, notice that pyDEM correctly distributes the area that drains into the top of the flat to the pixels at the edge of the flat instead of draining all of the area to a single pixel, or a few pixels. However, when a pixel that has a much lower elevation is present at the edge of a flat (a-4 and b-4), pyDEM drains preferentially along those pixels.

The c-1 case was used to test the third stage of processing, the edge correction stage. This is a challenging case because the drainage pattern is a spiral that crosses a single tile boundary multiple times. Without the edge correction, the UCA builds up in channels along a tile, but never reach the full value required (see Figure 7 right). Figure 7 also shows that pyDEM's edge correction algorithms are working correctly. The left UCA calculation is performed on a single tile using tauDEM, and it does not need edge corrections from adjoining tiles. The middle UCA calculation is performed using pyDEM over chunks of elevation sections forming a 7 by 7 grid. For this middle calculation, 316 rounds of the stage 3 edge correction was performed in serial, which means that every tile required multiple corrections as new information became available on the edges. Except for the edge pixels, the tauDEM and pyDEM results agree to withing 0.02%, which is reasonable considering how different the algorithms are.

pyDEM was also verified against tauDEM using all of the above test cases (not shown). In all cases without flats the results agreed as well as in the spiral case. For the cases with flats, tauDEM and pyDEM do not agree because they treat flat regions differently. Also, for cases with non-uniform grids, tauDEM and pyDEM do not agree. To illustrate the difference, consider the case of a conical topography with some added noise. On a uniform grid, the tauDEM and pyDEM solutions agree very well (Figure 8): the difference between the two UCA calculations is on the order of 10^{-7}, which is excellent given the vast differences between the UCA algorithms. However, Figure 9 shows that on a non-uniform grid only pyDEM correctly captures the shape of the geometry (note that the diagonal artifacts are from the $D\infty$ method). This is because pyDEM does not assume that the DEM data is uniformly gridded, but takes into account the geospatial coordinates when calculating the Aspect using the $D\infty$ method.

Finally, to verify that pyDEM is efficient, robust, and accurate for real data-sets, we calculated TWI over the conterminous US (Figure 10). In the figure, the spurious black areas are due to the interpolation of no data-values of our geoTiff viewer. The full calculation took approximately 3 days on a 32 core AWS compute node. Figure 2 (left) shows the UCA for a small region in Austin, TX from this calculation.

Summary

To solve our problem of analyzing the hydrology of large DEM data-sets spanning national and global scales, we designed, implemented, optimized, parallelized, and tested a new Python package, pyDEM. pyDEM implements the $D\infty$ method [tarboton97] to calculate the aspect and slope, and it uses a novel algorithm to calculate the upstream contributing area.

pyDEM enables the efficient, accurate, and robust analysis of large data-sets, while correcting for edge effects. pyDEM has been tested and agrees well with tauDEM.

Availability

The pyDEM package is available from the Python package index or through *pip install pydem*. Note this package is still in alpha and has not been tested on a wide range of operating systems. The source code is also hosted on GitHub (https://github.com/creare-com/pydem), and is free to modify, change, and improve under the Apache 2.0 license.

Acknowledgments

The authors are grateful to the Cold Regions Research and Engineering Laboratory for support under the SBIR grant W913E5-14-C-0002.

References

[beven79] Beven, K.J.; Kirkby, M. J.; Seibert, J. (1979). "A physically based, variable contributing area model of basin hydrology". Hydrolological Science Bulletin 24: 43–69

[beven95] Beven, K., Lamb, R., Quinn, P., Romanowicz, R., Freer, J., & Singh, V. P. (1995). Topmodel. Computer models of watershed hydrology., 627-668.

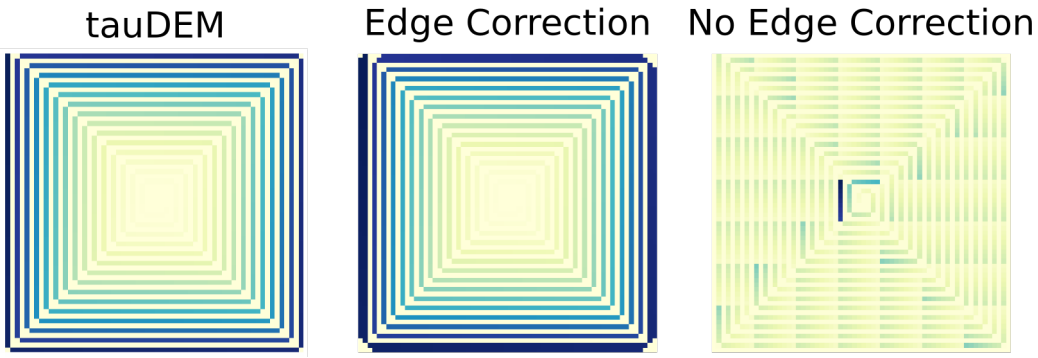


Fig. 7: *UCA for the spiral test case calculated over a single tile (left), multiple tiles with edge correction (middle) and multiple tiles without edge correction (right).*

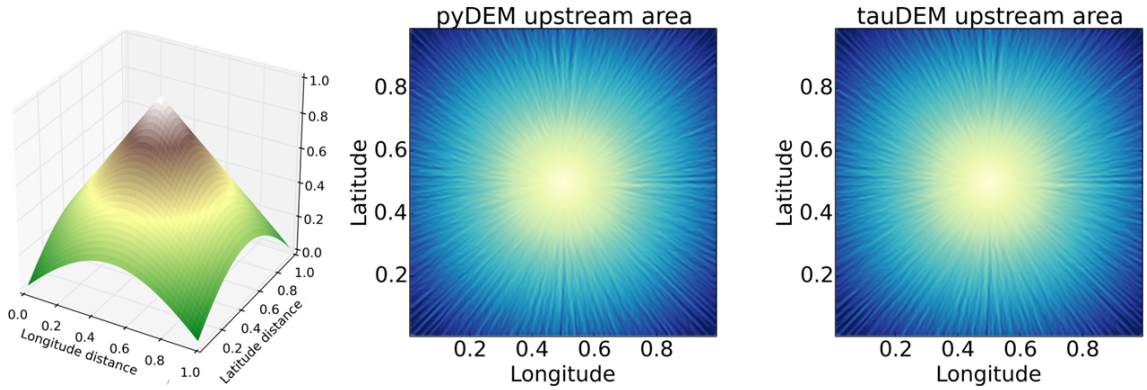


Fig. 8: *For a noisy cone (left), the UCA calculated using pyDEM (middle) and tauDEM (right) agree well when the DEM data is on a uniform grid.*

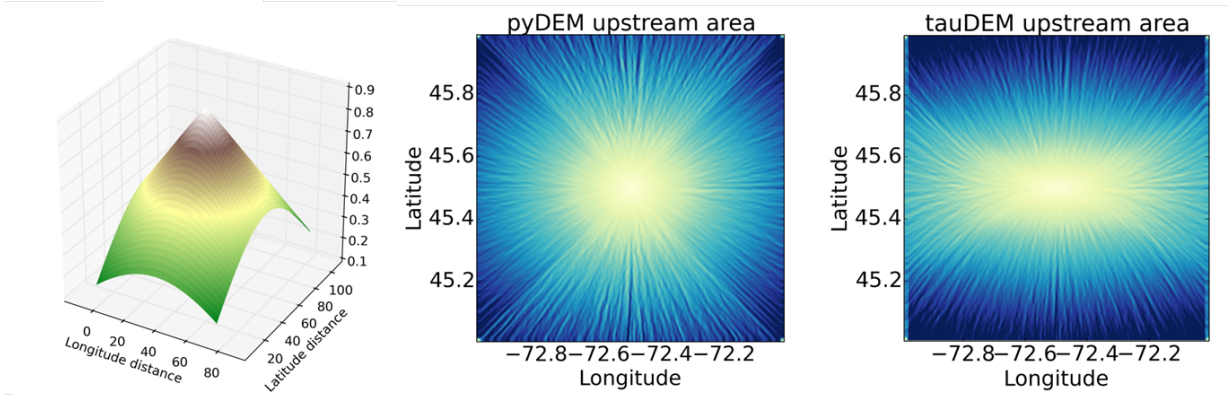


Fig. 9: *For a noisy cone (left), the UCA calculated using pyDEM (middle) and tauDEM (right) do not agree well when the DEM data is on a non-uniform grid. pyDEM correctly captures the shape of the geometry.*

[moore91] Moore, I. D., Grayson, R. B., & Ladson, A. R. (1991). Digital terrain modelling: a review of hydrological, geomorphological, and biological applications. Hydrological processes, 5(1), 3-30.

[tarboton97] Tarboton, D. G. (1997). A new method for the determination of flow directions and upslope areas in grid digital elevation models. Water Resources Research, 33(2), 309-319.

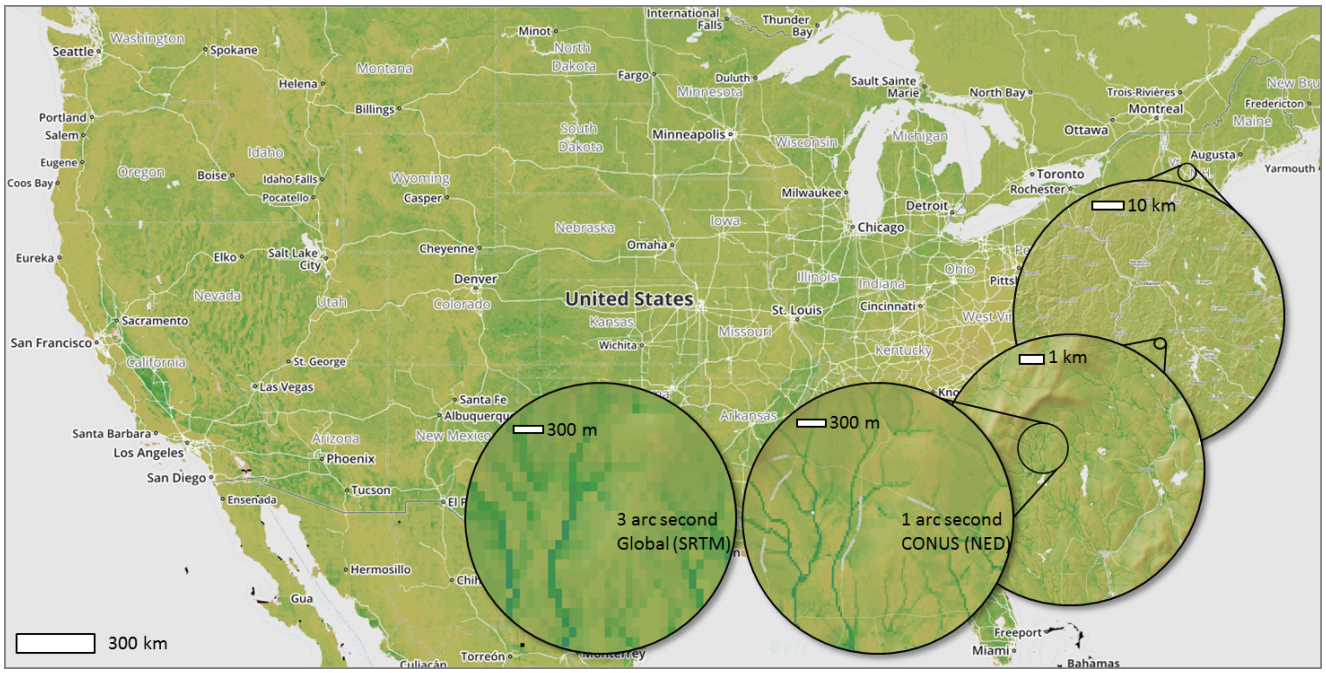


Fig. 10: *To verify pyDEM's performance over a large data set, TWI was calculated for the 1 arc-second resolution US National Elevation Database (shown with hill-shading overlay) and 3 arc-second SRTM globally (shown in inset).*

Widgets and Astropy: Accomplishing Productive Research with Undergraduates

Matthew Craig‡*

https://www.youtube.com/watch?v=hyxCDdBH1Mg

Abstract—This paper describes a tool for astronomical research implemented as an IPython notebook with a widget interface. The notebook uses Astropy, a community-developed package of fundamental tools for astronomy, and Astropy affiliated packages, as the back end. The widget interface makes Astropy a much more useful tool to undergraduates or other non-experts doing research in astronomy, filling a niche for software that connects beginners to research-grade code.

Index Terms—astronomy

Introduction

Incoming students interested in majoring in Physics at Minnesota State University Moorhead are often interested in doing astronomical research. The department encourages students to become involved in research as early as possible to foster their interest in science and because research experiences are correlated with successful completion of a degree [Lopatto2004].

The students typically have no programming experience, but even the smallest project requires calibrating and taking measurements from a couple of hundred images. To the extent possible, analysis needs to be automated. Roughly half of the students use Windows, the rest Mac OSX.

The problem, described in more detail below, is that the GUI-based software most accessible to these students is expensive, often available only on Windows, not clearly documented and does not leave a record of the choices made in calibrating the images so that future researchers can use the images with confidence. The free options largely require programming.

The proposed solution is a widget-based IPython notebook [Pérez2007] for calibrating astronomical images, called `reducer`.[1] A widget-based interface was chosen because students at this level are more comfortable with a GUI than with programming. An IPython notebook was chosen because of its rich display format, the ability to save both code and text, and the persistence of output in the notebook, which provides a record of the work done.

The back end of `reducer` is built on the Astropy project [Astropy2013], a community-driven effort to develop high-quality, open source tools for Python in astronomy, and on Astropy affiliated projects.[2] Astropy was chosen because it has a large developer community of professional astronomers.

Section *Background: Image analysis in optical stellar astronomy* provides background on the science of image calibration. In the following section the problem is discussed more completely, including a review of some of the available options for astronomical image processing. The section *"reducer package and notebook* discusses the use of `reducer`, while *"reducer" widget structure* presents its implementation. The widget classes in `reducer` are potentially useful in other applications.

Background: Image analysis in optical stellar astronomy

While a detailed description of astronomical data analysis is beyond the scope of this paper, some appreciation of the steps involved is useful for understanding its motivation.

An image from a CCD camera on a telescope is simply an array of pixel values. Several sources contribute to the brightness of an individual pixel in a raw image:

- Light from stars and other astronomical objects.
- Light from the nighttime sky; even a "dark" sky is not perfectly black.
- Noise that is related to the temperature of the camera and to the electronics that transfer the image from the detector chip in the camera to a computer.
- A DC offset to prevent negative pixel values.

The first stage of calibration is to remove the noise and offset from each image. The second stage is to correct for imperfections in the optical system that affect how much light gets to each pixel in the camera. An example of this sort of imperfection is dust on the camera itself.

A series of images is taken and then combined to perform each type of calibration. *Bias* images correct for the DC offset, *dark* images correct for thermal noise and *flats* correct for non-uniform illumination. One combines several frames of each type to reduce the electronic read noise present in the calibration images.

After calibration, the brightness of a pixel in the image is directly proportional to the amount of light that arrived at that pixel through the telescope. Note that light includes both starlight and light from the atmosphere.

* *Corresponding author: mcraig@mnstate.edu*
‡ *Department of Physics and Astronomy, Minnesota State University Moorhead*

1. Source code is at: https://github.com/mwcraig/reducer
2. http://www.astropy.org/affiliated/

Extraction of the brightness of individual stars is called photometry. There are several techniques for performing photometry, all of which estimate and eliminate the sky background.

The problem

Several software packages can calibrate astronomical images and perform photometry, so why write another one?

Ideally, such software would:

1) Be easily usable by an undergraduate with limited or no programming experience.
2) Work on Windows and Mac.
3) Have its operation well tested in published articles and/or be open source so that the details of its implementation can be examined.
4) Leave behind a record of the settings used by the software in calibrating the images and measuring star brightness.
5) Be maintained by a large, thriving community of developers.

Commercial software, like *MaxIm DL*[3], typically meets the first criteria. Past MSUM students were able to learn the software quickly. However, it leaves behind almost no record of how calibration was done: a fully calibrated image has one keyword added to its metadata: `CALSTAT='BDF'`. While this does indicate which corrections have been made[4], it omits important information like whether cosmic rays were removed from the calibration images and how the individual calibration images were combined.

The most extensively-tested and widely-used professional-grade package for calibration and photometry is IRAF [IRAF1993]. IRAF is both a scripting language and a set of predefined scripts for carrying out common operations. It is certainly widely used, with approximately 450 citations of the paper, and, because IRAF scripts store settings in text files, there is a record of what was done.

However, there are several challenges to using IRAF. It is easiest to install in Linux, though distributions exist for Mac and it is possible to use on Windows with Cygwin[5]. The IRAF command language (CL) is difficult to learn; undergraduates who have worked with it in summer REU programs report spending 3-4 weeks learning IRAF. That makes it infeasible to use as part of a one-semester research project. It is also no longer maintained[6].

One option that comes close to meeting all of the criteria is AstroImageJ[7], a set of astronomy plug-ins for the Java-based ImageJ [ImageJ2012]. It has a nice graphical interface that students in both an introductory astronomy course for non-majors and an upper-level course for majors found easy to use, is open source, free, and available on all platforms. It has a rich set of features, including both image calibrating and aperture photometry, and very flexible configuration. Its two weaknesses are that it leaves an incomplete record of the settings used in calibrating data and measuring brightness and it does not have an extensive support community.

3. http://www.cyanogen.com/

4. The bias offset and dark current were subtracted and the result divided by a flat frame to correct for non-uniform illumination.

5. http://www.cygwin.com/

6. The last update was in 2012 according to the IRAF web site, http://iraf.noao.edu

7. http://www.astro.louisville.edu/software/astroimagej/

The solution, broadly

Two relatively recent developments suggest the broad outlines of a solution that is sustainable in the long run:

- Initiation of the Astropy project in 2011, which unified what had previously been several independent effort to develop python software for astronomy. In addition to developing the core Astropy package, the Astropy organization gives affiliate status to packages that request it and meet its documentation, testing and coding standards[8]
- Addition of widgets to IPython notebooks in IPython, version 2. From the developer perspective, widgets are helpful because the Python API for widgets is rich enough to allow construction of complicated interfaces. There is no need to learn JavaScript to use the widgets effectively.

It is the combination of high-quality python packages for both the back-end and front-end that made development of `reducer` relatively straightforward.

A notebook-based solution offers a couple of other advantages over even the strongest of the GUI tools discussed in the previous section. The first is that exposure to programming broadly is useful to both the few students who become professional astronomers and the ones who do not. Though no programming is required to use `reducer`, there is code in several of the notebook cells. It represents something intermediate between a fully GUI application and script-only interface. Another is that exposure to Python programming is useful to both students who work immediately after graduation and those who go on to become scientists.

The `reducer` package and notebook

`reducer` is a pure Python package available on PyPI and as a conda package[9]. The user-facing part of the package is a single script, also called `reducer`. When invoked, it creates an IPython notebook, called `reduction.ipynb`, in the directory in which it is invoked.

The notebook will not overwrite images. The intent is that the raw, uncalibrated images are stored in a directory separate than the one containing the notebook. The calibrated images are saved, by default, in the same directory as the notebook, leaving a *human-readable* record with the images describing the choices made in calibration.

The notebook also does not provide an easy way to re- run the calibration short of deleting any calibrated files in the directory with the notebook and starting fresh. In discussions with students while developing `reducer` it became clear that it would be difficult or impossible to ensure that the state of the notebook reflected the state of the calibrated files, since it is possible for some notebook cells to be re-executed without all cells being re-executed.

That design decision simplified the package, allowed the notebook to refuse to overwrite files in the directory in which it is stored, and led to a focus on making sure a human could read the record of what was done. The package itself makes it easy to re-run the calibration with different settings should a later researcher choose to do so.

8. See http://www.astropy.org/affiliated for a list of affiliated packages and criteria.

9. Use channel `mwcraig` to get the conda package.

Image calibration

All of the calibration steps in reducer are performed by `ccdproc`, an Astropy affiliated package for astronomical image reduction [ccdproc]. Some of the `reducer` widgets contain some logic for automatically grouping and selecting images based on metadata in the image headers, described in more detail below.

This section begins with examples of the individual widgets that appear and the notebook, followed by an outline of the structure of the notebook as a whole.

Most of the widgets in `reduction.ipynb` are geared towards image calibration. There are two broad types, one for applying calibrations to a set of images, the other for combining calibration images.

Each widget has four states:

- Unselected; the widget is a simple button.
- Activated, but with incorrect or incomplete settings, shown in Fig. 1 for a `CombinerWidget`.
- Activated and ready for action, with settings that enable the action to be completed, shown in Fig. 2.
- Locked, after execution of calibration step in the widget, shown in Fig. 3. *Note that the IPython notebook does not store the widget state in the notebook.*[10] When a `reducer` notebook is re-opened the only record guaranteed to be preserved is the printed text below the widget.

Fig. 1: *Example widget for combining images before settings have been set in a self-consistent way. Compare to Fig. 2*

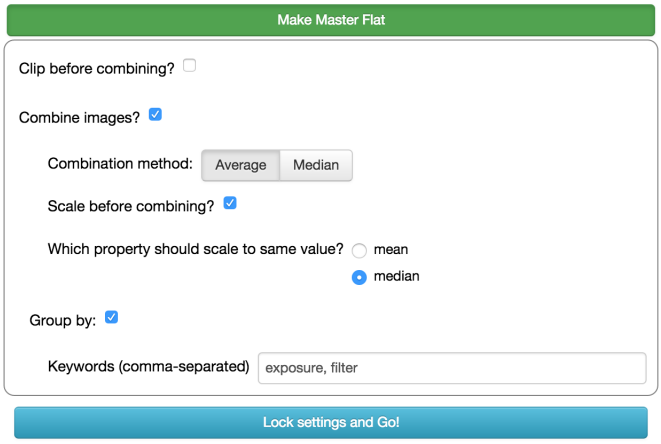
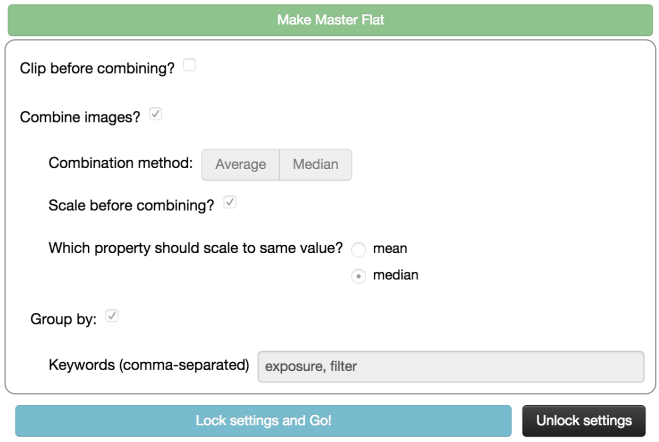


Fig. 2: *Same widget as Fig. 1 after consistent settings have been chosen. Note that the style of the top button changes and a "Go" button appears when settings are sensible; in this case the user needs to at least select a combination method. The additional options under "Combine images" are presented when the checkbox is selected.*

A few features of the `CombinerWidget` illustrate the logic used in `reducer` to semi-automatically select the images on




```
Make Master Flat Yes
        Clip before combining? No
        Combine images? Yes
                Combination method:: Average
                Scale before combining? Yes
                        Which property should scale to same value?: median
        Group by: Yes
                Keywords (comma-separated): exposure, filter
```

Fig. 3: *Same widget as Fig. 2, after executing the calibration step. Note that a record of the settings is printed into the notebook cell below the widget to ensure a record remains in the notebook after reopening it.*

which it should act. An `apply_to` argument to the initializer controls which calibrated images the widget will act on; in this case its value is `{'imagetyp': 'flat'}`, which selects the calibration images used to correct non-uniform illumination. A `group_by` argument to the widget initializer controls controls how the images selected by `apply_to` are combined. In the example shown, all images with the same filter and exposure time will be combined by averaging, after each image has been scaled to the same median value.

Each image, including the images used in the calibration itself, is processed by a `ReductionWidget`, like that shown in Fig. 4. That examples is for a "light" image, an image that contains the objects of interest. Each of the calibration images has some of these steps applied also, though some of the calibration steps are not displayed for some of the calibration images.

As with the `CombinerWidget`, an `apply_to` argument to the widget constructor determines which images are processed by the widget.

The calibration part of the notebook is composed of four pairs of widgets, one pair for calibrating and combining bias images, and additional pairs for darks, flats, and science images. One of the strengths of widget-based notebooks is that they are user- editable applications. If there is a particular calibration step that is not needed, the cells that create those widgets can simply be deleted.

Image browser

Reducer also contains a basic image browser, which organizes the images based on a table of metadata, and displays, when an image is selected, the image and all of the metadata in that image in separate tabs in the widget. An example is shown in Fig. 5.

10. In IPython 2.x it is impossible to easily save the widget state, and the widget is not part of the DOM, so it is not stored when the notebook is saved. In 3.x the widget is preserved, but saving the state takes additional developer work.

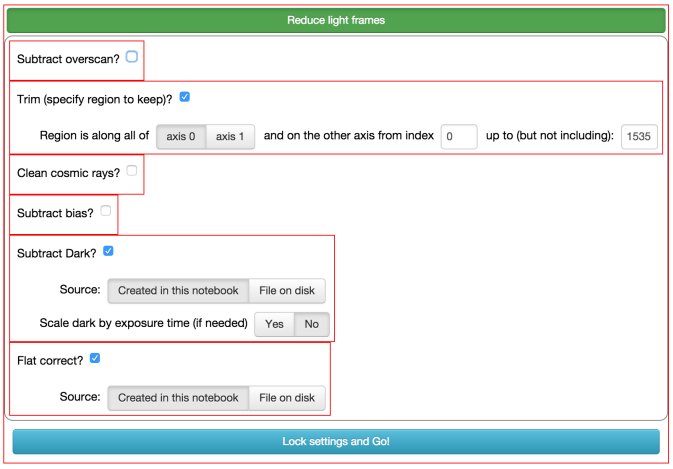


Fig. 4: *Widget that applies calibrations to a set of images. Display of some of the individual steps (e.g. subtracting bias) can be suppressed with optional arguments when the widget object is created. Red borders are drawn around each instance of the base widget class described in the section "`reducer` widget structure".*

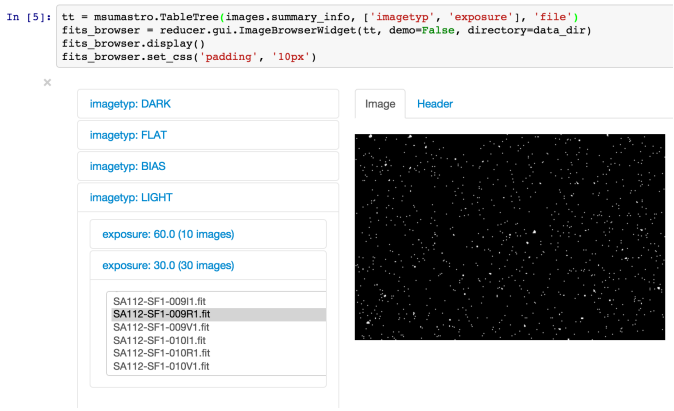


Fig. 5: *The image display widget arranges images nested by image metadata values. In this case the two keywords used for grouping the images were `imagetyp` and `exposure`. When an file name is selected, either the image or its metadata can be displayed.*

`reducer` widget structure

At the base of the reducer widget structure is an extension of a container widget from IPython. This class, `ToggleContainerWidget`, adds a toggle to control display of the contents of the container, and a list of child widgets displayed in the container.[11] Since a `ToggleContainerWidget` can have another `ToggleContainerWidget` as a child, this immediately provides an interface for presenting a user with a nested list of options. Fig. **??** has a thin red border drawn around each element that is a subclass of"ToggleContainerWidget"

In IPython 2 it is not possible to preserve the state of widgets between sessions, and in IPython 3 it remains difficult, so the `ToggleContainerWidget` class defines a `__str__` method to facilitate printing the contents of the widget. The purpose of this is not to provide a way to progammatically rebuild the widget; it is to provide a human reader of the notebook a history of what was done in the notebook.

The code below implements a basic `ToggleContainerWidget` called `MyControl`. The widget it produces is shown in Fig. 6.

```
from reducer.gui import ToggleContainerWidget
from reducer.astro_gui import override_str_factory
from IPython.html.widgets import CheckboxWidget

class MyControl(ToggleContainerWidget):
    """
    Straightforward reducer-widget subclass.
    """
    def __init__(self, *arg, **kwd):
        super(MyControl, self).__init__(*arg, **kwd)

        # b_box is a plain IPython checkbox with a more
        # meaningful string representation.
        b_box = override_str_factory(\
            CheckboxWidget(description='Check me'))

        # Another plain check box, but with the default
        # string representation.
        c_box = CheckboxWidget(description="Don't check me")

        # These children are contained in the
        # MyControl widget
        self.add_child(b_box)
        self.add_child(c_box)
```

The `is_sane` property of a `ToggleContainerWidget` can be overridden by subclasses to indicate that the settings in the widget are sensible. This provides some minimal validation of user input. The code below implements `is_sane` for `MyControl`.

```
@property
def is_sane(self):
    """
    Settings are correct when the "Check me" box is
    checked and the "Don't check me" box is unchecked.
    """
    return (self.container.children[0].value and
            not self.container.children[1].value)
```

The widget also has an `action` method. This method must be overridden by subclasses to do anything useful. It is used in some cases to set up an environment for acting on data files and to invoke the action of each child widget on each data file, in the order the children are listed in the widget. In other cases, the action simply invokes a function that acts on the data file.

The action method for this example is below.

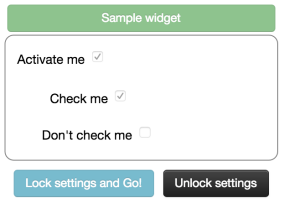
```
def action(self):
    """
    A simple action, one for each child.
    """
    import time

    for child in self.container.children:
        time.sleep(0.5)
```

One subclass of `ToggleContainerWidget`, a `ToggleGoWidget`, styles the toggle as a button instead of a checkbox, and adds a "Start" button that is displayed only when the settings of the widget and all of its children is "sane" as defined by the `is_sane` method. What the "Start" button is pushed it invokes the `action` method of the `ToggleGoWidget` and displays a progress bar while working. In Fig. 4, the outermost container is a `ToggleGoWidget`.

The code below creates a `ToggleGoWidget`, adds an instance of `MyControl` to it, and displays it, creating the widget in Fig. 6.

```
from reducer.gui import ToggleGoWidget
go_widget = ToggleGoWidget(description='Sample widget',
                           toggle_type='button')
control = MyControl(description='Activate me')
go_widget.add_child(control)
go_widget.display()
```




```
Sample widget Yes
        Activate me Yes
                Check me: True
                <IPython.html.widgets.widget_bool.CheckboxWidget object at 0x109fc26d0>
```

Fig. 6: *The widget produced by the sample code in the section* "reducer" widget structure. *Note the string output of the checkbox "Don't check me", whose* `__str__` *method has not been overridden.*

Use with students

This package has been used with 8 undergraduate physics majors ranging from first-semester freshman to seniors; it was also used in an astronomical imaging course that included two non-physics majors. It typically took one 1-hour session to train the students to use the notebook. The other graphical tool used in the course took considerably longer for the students to set up and left no record the steps and settings the students followed in calibrating the data.

Conclusion

IPython widgets provide a convenient glue for connecting novice users with expert-developed software. The notebook interface preserves a bare-bones record of the actions taken by the user, sufficient for another user to reproduce the calibration steps taken.

Appendix: Bootstrapping a computing environment for students

While the goal of this work is to minimize the amount of programming new users need to do, there are a few things that cannot be avoided: installing Python and the SciPy [scipy2001] stack, and learning a little about how to use a terminal.

Students find the Anaconda Python distribution[12] easy to install and it is available for all platforms. From a developer point of view, it also provides a platform for distributing binary packages, particularly useful to the students on Windows.

Students also need minimal familiarity with the terminal to install the reducer package, generate a notebook for analyzing their data and launching the notebook. The *Command Line Crash Course* from *Learn Code the Hard Way*[13] is an excellent introduction, has tracks for each major platform, and is very modular.

REFERENCES

[Astropy2013] Astropy Collaboration, Robitaille, T.~P., Tollerud, E.~J., et al., *Astropy: A community Python package for astronomy*, Astronomy & Astrophysics, 558: A33, October 2013.

[scipy2001] Jones, E., Oliphant, T., Peterson, P. *et al*, *SciPy: Open source scientific tools for Python*, http://scipy.org/ 2001

[Pérez2007] Pérez, F. and Granger, B.E. *IPython: A System for Interactive Scientific Computing*, Computing in Science and Engineering, 9(3):21-29, May/June 2007

[ccdproc] Crawford, S and Craig, M., https://github.com/ccdproc

[Lopatto2004] Lopatto, D. *Survey of undergraduate research experiences (SURE): First findings.* Cell biology education 3.4 (2004).

[IRAF1993] Tody, D., *IRAF in the Nineties*, Astronomical Data Analysis Software and Systems II, A.S.P. Conference Series, Vol. 52, 1993

[ImageJ2012] Schneider, C.A., Rasband, W.S., Eliceiri, K.W. *NIH Image to ImageJ: 25 years of image analysis*, Nature Methods 9, 671-675, 2012.

11. Classes in the current version of `reducer` use IPython 2-style class names ending in "Widget". Part of upgrading the package to IPython 3 widgets will be removing that ending.

12. https://store.continuum.io/cshop/anaconda/

13. http://cli.learncodethehardway.org/book/

Dask: Parallel Computation with Blocked algorithms and Task Scheduling

Matthew Rocklin‡*

https://www.youtube.com/watch?v=1kkFZ4P-XHg

Abstract—Dask enables parallel and out-of-core computation. We couple blocked algorithms with dynamic and memory aware task scheduling to achieve a parallel and out-of-core NumPy clone. We show how this extends the effective scale of modern hardware to larger datasets and discuss how these ideas can be more broadly applied to other parallel collections.

Index Terms—parallelism, NumPy, scheduling

Introduction

The Scientific Python stack [Oli07] rarely leverages parallel computation. Code built off of NumPy [vdW11] or Pandas [McK10] generally runs in a single thread on data that fits comfortably in memory. Advances in hardware in the last decade in multi-core processors and solid state drives provide significant and yet largely untapped performance advantages.

However, the Scientific Python stack consists of hundreds of software packages, papers, PhD theses, and developer-years. This stack is a significant intellectual and financial investment that, for the most part, does not align well with modern hardware. We seek software solutions to parallelize this software stack without triggering a full rewrite.

This paper introduces `dask`, a specification to encode parallel algorithms, using primitive Python dictionaries, tuples, and callables. We use `dask` to create `dask.array` a parallel N-dimensional array library that copies the NumPy interface, uses all of the cores in a modern processor, and manages data well from disk. Dask.array serves both as a general library for parallel out-of-core `ndarrays` and also as a demonstration that we can parallelize complex codebases like NumPy in a straightforward manner using blocked algorithms and task scheduling.

We first define `dask` graphs and give a trivial example of their use. We then share the design of `dask.array` a parallel `ndarray`. Then we discuss dynamic task scheduling and policies to minimize memory footprint. We then give two examples using `dask.array` on computational problems. We then briefly discuss `dask.bag` and `dask.dataframe`, two other collections in the `dask` library. We finish with thoughts about extension of this approach into the broader Scientific Python ecosystem.

* *Corresponding author: mrocklin@gmail.com*
‡ *Continuum Analytics*

Modern Hardware

Hardware has changed significantly in recent years. The average personal notebook computer (the bulwark of most scientific development) has roughly four physical cores and a solid state drive (SSD). The four physical cores present opportunities for linear speedup of computationally bound code. We refer to algorithms that use multiple cores simultaneously as *parallel*. The solid state drives have high read bandwidths and low seek times which enables them to serve as large and cheap extensions of physical memory. We refer to systems that efficiently use disk as extensions of memory as *out-of-core*.

Modern workstations extend these trends to include sixteen to sixty-four cores, hundreds of gigabytes of RAM, and RAID arrays of SSDs offering 2GB/s read bandwidths. These systems rival small clusters in scale but continue to offer the convenience of single-machine administration and shared-memory computing. This system rivals the performance of massively parallel distributed systems up to a surprisingly large scale while maintaining a low maintenance and programming cost.

Dask Graphs

Normally humans write programs and then compilers/interpreters interpret them (e.g. `python`, `javac`, `clang`). Sometimes humans disagree with how these compilers/interpreters choose to interpret and execute their programs. In these cases humans often bring the analysis, optimization, and execution of code into the code itself.

Commonly a desire for parallel execution causes this shift of responsibility from compiler to human developer. In these cases we often represent the structure of our program explicitly as data within the program itself.

Dask is a specification that encodes task schedules with minimal incidental complexity using terms common to all Python projects, namely dicts, tuples, and callables. Ideally this minimum solution is easy to adopt and understand by a broad community.

We define a dask graph as a Python dictionary mapping keys to tasks or values. A key is any Python hashable, a value is any Python object that is not a task, and a task is a Python tuple with a callable first element.

Example

Consider the following simple program

```
def inc(i):
    return i + 1
```

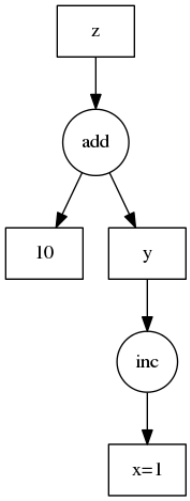


Fig. 1: *A simple dask dictionary*

```
def add(a, b):
    return a + b

x = 1
y = inc(x)
z = add(y, 10)
```

We encode this as a dictionary below:

```
d = {'x': 1,
     'y': (inc, 'x'),
     'z': (add, 'y', 10)}
```

While less pleasant than our original code this representation can be analyzed and executed by other Python code, not just the CPython interpreter. We don't recommend that users write code in this way, but rather that it is an appropriate target for automated systems. Also, in non-toy examples the execution times are likely much larger than for `inc` and `add`, warranting the extra complexity.

Specification

We represent a computation as a directed acyclic graph of tasks with data dependencies. Dask is a specification to encode such a graph using ordinary Python data structures, namely dicts, tuples, functions, and arbitrary Python values.

A **dask graph** is a dictionary mapping identifying keys to values or tasks. We explain these terms after showing a complete example:

```
{'x': 1,
 'y': 2,
 'z': (add, 'x', 'y'),
 'w': (sum, ['x', 'y', 'z'])}
```

A **key** can be any hashable value that is not a task.

```
'x'
('x', 2, 3)
```

A **task** is a tuple with a callable first element. Tasks represent atomic units of work meant to be run by a single worker.

```
(add, 'x', 'y')
```

We represent a task as a tuple such that the *first element is a callable function* (like `add`), and the succeeding elements are *arguments* for that function.

An **argument** may be one of the following:

1) Any key present in the dask like `'x'`
2) Any other value like `1`, to be interpreted literally
3) Other tasks like `(inc, 'x')`
4) List of arguments, like `[1, 'x', (inc, 'x')]`

So all of the following are valid tasks

```
(add, 1, 2)
(add, 'x', 2)
(add, (inc, 'x'), 2)
(sum, [1, 2])
(sum, ['x', (inc, 'x')])
(np.dot, np.array([...]), np.array([...]))
```

The dask spec provides no explicit support for keyword arguments. In practice we combine these into the callable function with `functools.partial` or `toolz.curry`.

Dask Arrays

The `dask.array` submodule uses dask graphs to create a NumPy-like library that uses all of your cores and operates on datasets that do not fit in memory. It does this by building up a dask graph of blocked array algorithms.

The `dask.array` submodule is not the first library to implement a "Big NumPy Clone". Other partial implementations exist including Biggus an out-of-core `ndarray` specialized for climate science, Spartan [Pow14] a distributed memory `ndarray`, and Distarray a distributed memory `ndarray` that interacts well with other distributed array libraries like Trillinos. There have also been numerous projects in traditional high performance computing space including Elemental [Pou13], High Performance Fortran, etc.. Finally Theano [Ber10], an array compiler in Python with powerful optimizations and GPU support, statically schedules and reasons about array computations and has proven particularly valuable in machine learning applications.

Each of these implementations focuses on a particular application or problem domain. Dask.array distinguishes itself in that it focuses on a very general class of NumPy operations and streaming execution through dynamic task scheduling.

Blocked Array Algorithms

Blocked algorithms compute a large result like "take the sum of these trillion numbers" with many small computations like "break up the trillion numbers into one million chunks of size one million, sum each chunk, then sum all of the intermediate sums." Through tricks like this we can evaluate one large problem by solving very many small problems.

Blocked algorithms have proven useful in modern numerical linear algebra libraries like Flame [Gei08] and Plasma [Agu09] and more recently in data parallel systems like Dryad [Isa07] and Spark [Zah10]. These compute macroscopic operations with a collection of related in-memory operations.

Dask.array takes a similar approach to linear algebra libraries but focuses instead on the more pedestrian `ndarray` operations, like arithmetic, reductions, and slicing common in interactive use.

Example: `arange`

Dask array functions produce `Array` objects that hold on to dask graphs. These dask graphs use several `numpy` functions to achieve the full result. In the following example one call to `da.arange` creates a graph with three calls to `np.arange`

```
>>> import dask.array as da
>>> x = da.arange(15, chunks=(5,))
```

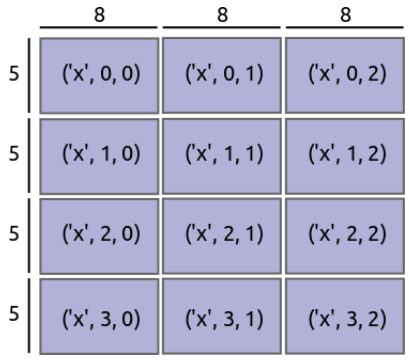


Fig. 2: *A dask array*

```
>>> x       # Array object metadata
dask.array<x-1, shape=(15,), chunks=((5, 5, 5)), dtype=int64>
>>> x.dask  # Every dask array holds a dask graph
{('x', 0): (np.arange, 0, 5),
 ('x', 1): (np.arange, 5, 10),
 ('x', 2): (np.arange, 10, 15)}
```

Further operations on `x` create more complex graphs

```
>>> z = (x + 100).sum()
>>> z.dask
{('x', 0): (np.arange, 0, 5),
 ('x', 1): (np.arange, 5, 10),
 ('x', 2): (np.arange, 10, 15),
 ('y', 0): (add, ('x', 0), 100),
 ('y', 1): (add, ('x', 1), 100),
 ('y', 2): (add, ('x', 2), 100),
 ('z', 0): (np.sum, ('y', 0)),
 ('z', 1): (np.sum, ('y', 1)),
 ('z', 2): (np.sum, ('y', 2)),
 ('z',): (sum, [('z', 0), ('z', 1), ('z', 2)])}
```

Dask.array also holds convenience functions to execute this graph, completing the illusion of a NumPy clone

```
>>> z.compute()
1605
```

Array metadata

In the example above `x` and `z` are both `dask.array.Array` objects. These objects contain the following data

1) A dask graph, `.dask`
2) Information about shape and chunk shape, called `.chunks`
3) A name identifying which keys in the graph correspond to the result, `.name`
4) A dtype

The second item here, `chunks`, deserves further explanation. A normal NumPy array knows its `shape`, a dask array must know its shape and the shape of all of the internal NumPy blocks that make up the larger array. These shapes can be concisely described by a tuple of tuples of integers, where each internal tuple corresponds to the lengths along a single dimension.

In the example above we have a 20 by 24 array cut into uniform blocks of size 5 by 8. The `chunks` attribute describing this array is the following:

```
chunks = ((5, 5, 5, 5), (8, 8, 8))
```

Where the four fives correspond to the heights of the blocks along the first dimension and the three eights correspond to the widths of the blocks along the second dimension. This particular example has uniform sizes along each dimension but this need not be the case. Consider the chunks of the following example operations

```
>>> x[::2].chunks
((3, 2, 3, 2), (8, 8, 8))

>>> x[::2].T.chunks
((8, 8, 8), (3, 2, 3, 2))
```

Every `dask.array` operation, like `add`, slicing, or `transpose` must take the graph and all metadata, add new tasks into the graph and determine new values for each piece of metadata.

Capabilities and Limitations

Adding subgraphs and managing metadata for most of NumPy is difficult but straightforward. At present `dask.array` is around 5000 lines of code (including about half comments and doc-strings). It encompasses most commonly used operations including the following:

- Arithmetic and scalar mathematics, `+, *, exp, log, ...`
- Reductions along axes, `sum(), mean(), std(), sum(axis=0), ...`
- Tensor contractions / dot products / matrix multiply, `tensordot`
- Axis reordering / transpose, `transpose`
- Slicing, `x[:100, 500:100:-2]`
- Fancy indexing along single axes with lists or NumPy arrays, `x[:, [10, 1, 5]]`
- A variety of utility functions, `bincount, where, ...`

However `dask.array` is unable to handle any operation whose shape can not be determined ahead of time. Consider for example the following common NumPy operation

```
x[x > 0]  # can not determine shape of output
```

The shape of this array depends on the number of positive elements in `x`. This shape is not known given only metadata; it requires knowledge of the values underlying `x`, which are not available at graph creation time. Note however that this case is fairly rare; for example it is possible to determine the shape of the output in all other cases of slicing and indexing, e.g.

```
x[10::3, [1, 2, 5]]  # can determine shape of output
```

Dynamic Task Scheduling

We now discuss how `dask` executes task graphs. How we execute these graphs strongly impacts performance. Fortunately we can tackle this problem with a variety of approaches without touching the graph creation problem discussed above. Graph creation and graph execution are separable problems. The dask library contains schedulers for single-threaded, multi-threaded, multi-process, and distributed execution.

Current dask schedulers all operate *dynamically*, meaning that execution order is determined during execution rather than ahead of time through static analysis. This is good when runtimes are not known ahead of time or when the execution environment contains uncertainty. However dynamic scheduling does preclude certain clever optimizations.

Dynamic task scheduling has a rich literature and numerous projects, both within the Python ecosystem with projects like

Spotify's Luigi for bulk data processing and projects without the ecosystem like DAGuE [Bos12] for more high performance task scheduling. Additionally, data parallel systems like Dryad or Spark contain their own custom dynamic task schedulers.

None of these solutions, nor much of the literature in dynamic task scheduling, suited the needs of blocked algorithms for shared memory computation. We needed a lightweight, easily installable Python solution that had latencies in the millisecond range and was mindful of memory use. Traditional task scheduling literature usually focuses on policies to expose parallelism or chip away at the critical path. We find that for bulk data analytics these are not very relevant as parallelism is abundant and critical paths are comparatively short relative to the depth of the graph.

The logic behind dask's schedulers reduces to the following situation: A worker reports that it has completed a task and that it is ready for another. We update runtime state to record the finished task, mark which new tasks can be run, which data can be released, etc.. We then choose a task to give to this worker from among the set of ready-to-run tasks. This small choice governs the macro-scale performance of the scheduler.

Instead of these metrics found in the literature we find that for out-of-core computation we need to choose tasks that allow us to release intermediate results and keep a small memory footprint. This lets us avoid spilling intermediate values to disk which hampers performance significantly. After several other policies we find that the policy of *last in, first out* is surprisingly effective. That is we select tasks whose data dependencies were most recently made available. This causes a behavior where long chains of related tasks trigger each other, forcing the scheduler to finish related tasks before starting new ones. We implement this with a simple stack, which can operate in constant time.

We endeavor to keep scheduling overhead low at around 1ms per task. Updating executing state and deciding which task to run must be made very quickly. To do this we maintain a great deal of state about the currently executing computation. The set of ready-to-run tasks is commonly quite large, in the tens or hundreds of thousands in common workloads and so in practice we must maintain enough state so that we can choose the right task in constant time (or at least far sub-linear time).

Finally, power users can disregard the dask schedulers and create their own. Dask graphs are completely separate from the choice of scheduler and users may select the right scheduler for their class of problem or, if no ideal scheduler exists, build one anew. The default single-machine scheduler is about three hundred significant lines of code and has been adapted to single-threaded, multi-threaded, multi-processing, and distributed computing variants.

Example: Matrix Multiply

We benchmark dask's blocked matrix multiply on an out-of-core dataset. This demonstrates the following:

1) How to interact with on-disk data
2) The blocked algorithms in dask.array achieve similar performance to modern BLAS implementations on compute-bound tasks

We set up a trivial input dataset

```
import h5py
f = h5py.File('myfile.hdf5')
A = f.create_dataset(name='/A',
         shape=(200000, 4000), dtype='f8',
         chunks=(250, 250), fillvalue=1.0)
B = f.create_dataset(name='/B',
         shape=(4000, 4000), dtype='f8',
         chunks=(250, 250), fillvalue=1.0)
out = f.create_dataset(name='/out',
         shape=(4000, 4000), dtype='f8',
         chunks=(250, 250))
```

Performance (GFLOPS)	NumPy	Dask.array
ATLAS BLAS	6	18
OpenBLAS (one)	11	23
OpenBLAS (four)	22	11

TABLE 1: *Matrix Multiply GigaFLOPS for NumPy/Dask.array and for ATLAS and OpenBLAS with one and four threads*

The Dask convenience method, `da.from_array`, creates a graph that can pull data from any object that implements NumPy slicing syntax. The `da.store` function can then store a large result in any object that implements NumPy setitem syntax.

```
import dask.array as da
a = da.from_array(A, chunks=(1000, 1000))
b = da.from_array(B, chunks=(1000, 1000))

c = a.dot(b)  # another dask Array, not yet computed
c.store(out)  # Store result into output space
```

Results: We do this same operation in different settings.

We use either use NumPy or `dask.array`:

1) Use NumPy on a big-memory machine
2) Use dask.array in a small amount of memory, pulling data from disk, using four threads

We compare different BLAS implementations:

1) ATLAS BLAS, single threaded, unblocked
2) OpenBLAS, single threaded
3) OpenBLAS, multi-threaded

For each configuration we compute the number of floating point operations per second.

We note the following

1) Compute-bound tasks are computationally bound by memory; we don't experience a slowdown
2) Dask.array can effectively parallelize and block ATLAS BLAS for matrix multiplies
3) Dask.array doesn't significantly improve when using an optimized BLAS, presumably this is because we've already reaped most of the benefits of blocking and multi-core
4) One should not mix multiple forms of multi-threading. Four dask.array threads each spawning multi-threaded OpenBLAS DGEMM calls results in worse performance.

Example: Meteorology

Performance is secondary to capability. In this example we use `dask.array` to manipulate climate datasets that are larger than memory. This example shows the following:

1) Use `concatenate` and `stack` to manage large piles of HDF5 files (a common case)
2) Use reductions and slicing to manipulate stacks of arrays

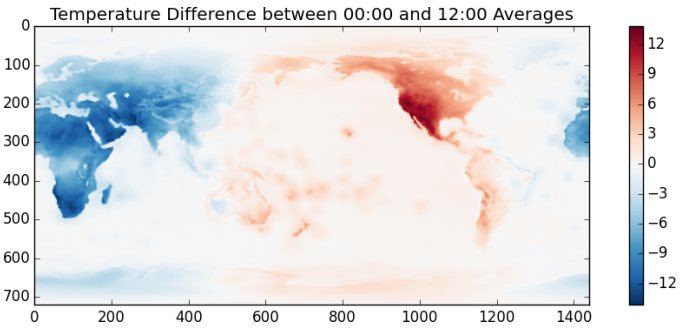


Fig. 3: *We use typical NumPy slicing and reductions on a large volume of data to show the average temperature difference between noon and midnight for year 2014*

3) Interact with other libraries in the ecosystem using the `__array__` protocol.

We start with a typical setup, a large pile of NetCDF files.:

```
$ ls
2014-01-01.nc3  2014-03-18.nc3  2014-06-02.nc3
2014-01-02.nc3  2014-03-19.nc3  2014-06-03.nc3
2014-01-03.nc3  2014-03-20.nc3  2014-06-04.nc3
2014-01-04.nc3  2014-03-21.nc3  2014-06-05.nc3
...             ...             ...
```

Each of these files contains the temperature at two meters above ground over the earth at quarter degree resolution, every six hours.

```
>>> from netCDF4 import netCDF4
>>> t = Dataset('2014-01-01.nc3').variables['t2m']
>>> t.shape
(4, 721, 1440)
```

We can collect many of these files together using `da.concatenate`, resulting in a single large array.

```
>>> from glob import glob
>>> filenames = sorted(glob('2014-*.nc3'))
>>> temps = [Dataset(fn).variables['t2m']
...          for fn in filenames]

>>> import dask.array as da
>>> arrays = [da.from_array(t, blockshape=(4,200,200))
...           for t in temps]
>>> x = da.concatenate(arrays, axis=0)

>>> x.shape
(1464, 721, 1440)
```

We can now play with this array as though it were a NumPy array. Because dask.arrays implement the `__array__` protocol we can dump them directly into functions of other libraries. These libraries will trigger computation when they call `np.array(...)` on their input.

```
>>> from matplotlib import imshow
>>> imshow(x[::4].mean(axis=0) - x[2::4].mean(axis=0)
...        , cmap='RdBu_r')
```

This computation took about a minute on an old notebook computer. It was bound by disk access. Meteorological cases tend to be I/O bound rather than compute bound, taking more advantage of `dask`'s memory-aware schedulers rather than parallel computation. In other cases, such as parallel image processing, this trend is reversed.

Other Collections

The dask library contains parallel collections other than `dask.array`. We briefly describe `dask.bag` and `dask.dataframe`

- `dask.array` = `numpy` + `threading`
- `dask.bag` = `toolz` + `multiprocessing`
- `dask.dataframe` = `pandas` + `threading`

Bag

A *bag* is an unordered collection with repeats. It is like a Python list but does not guarantee the order of elements. Because we typically compute on Python objects in `dask.bag` we are bound by the Global Interpreter Lock and so switch from using a multi-threaded scheduler to a multi-processing one.

The `dask.bag` API contains functions like `map` and `filter` and generally follows the PyToolz API. We find that it is particularly useful on the front lines of data analysis, particularly in parsing and cleaning up initial data dumps like JSON or log files because it combines the streaming properties and solid performance of projects like `cytoolz` with the parallelism of multiple processes.

```
>>> import dask.bag as db
>>> import json
>>> b = db.from_filenames('2014-*.json.gz')
...       .map(json.loads)

>>> alices = b.filter(lambda d: d['name'] == 'Alice')
>>> alices.take(3)
({'name': 'Alice', 'city': 'LA',  'balance': 100},
 {'name': 'Alice', 'city': 'LA',  'balance': 200},
 {'name': 'Alice', 'city': 'NYC', 'balance': 300},

>>> dict(alices.pluck('city').frequencies())
{'LA': 10000, 'NYC': 20000, ...}
```

DataFrame

The `dask.dataframe` module implements a large dataframe out of many Pandas DataFrames. The interface should be familiar to users of Pandas.

```
>>> import dask.dataframe as dd
>>> df = dd.read_csv('nyc-taxi-*.csv.gz')

>>> g = df.groupby('medallion')
>>> g.trip_time_in_secs.mean().head(5)
medallion
0531373C01FD1416769E34F5525B54C8     795.875026
867D18559D9D2941173AD7A0F3B33E77     924.187954
BD34A40EDD5DC5368B0501F704E952E7     717.966875
5A47679B2C90EA16E47F772B9823CE51     763.005149
89CE71B8514E7674F1C662296809DDF6     869.274052
Name: trip_time_in_secs, dtype: float64
```

Currently `dask.dataframe` uses the threaded scheduler but does not achieve the same parallel performance as `dask.array` due to the GIL. We are enthusiastic about ongoing work in Pandas itself to release the GIL.

The dask dataframe can compute efficiently on *partitioned* datasets where the different blocks are well separated along an index. For example in time series data we may know that all of January is in one block while all of February is in another. Join, groupby, and range queries along this index are significantly faster when working on partitioned datasets.

Dask.dataframe benefits users by providing trivial access to larger-than-memory datasets and, where Pandas does release the GIL, parallel computation.

Dask for General Computing

The higher level collections `dask.array/bag/dataframe` demonstrate the flexibility of the dask graph specification to encode sophisticated parallel algorithms and the capability of the dask schedulers to execute those graphs intelligently on a multi-core machine. Opportunities for parallel execution extend beyond beyond `ndarrays` and dataframes.

In the beginning of this document we gave the following toy example to help define dask graphs.

```
d = {'x': 1,
     'y': (inc, 'x'),
     'z': (add, 'y', 10)}
```

While this example of dask graphs is trivial it represents a broader class of free-form computations that don't fit neatly into a single high-level abstraction like arrays or dataframes but are instead just a bunch of related Python functions with data dependencies. In this context Dask offers a lightweight spec and range of schedulers as well as excellent error reporting and diagnostic facilities. In private projects we have seen great utility and performance from using the dask threaded scheduler to refactor and execute existing processing pipelines on large multi-core computers.

Low Barrier to Entry

The simplicity of dask graphs (no classes or frameworks) presents a very low barrier to entry. Users only need to understand basic concepts common to Python (or indeed most modern languages) like dictionaries, tuples, and functions as variables. As an example consider the work in [Tep15] in which the authors implement out-of-core parallel non-negative matrix factorizations on top of dask.array without significant input from dask core developers. This demonstrates that algorithmic domain experts can implement complex algorithms with dask and achieve good results with a minimum of framework investment.

To demonstrate complexity we present the graph of an out-of-core singular value decomposition contributed by those authors to the `dask.array.linalg` library.

```
>>> import dask.array as da
>>> x = da.ones((5000, 1000), chunks=(1000, 1000))
>>> u, s, v = da.svd(x)
```

This algorithm is complex enough without having to worry about software frameworks. Mathematical experts were able to implement this without having to simultaneously develop expertise in a complex parallel programming framework.

Final Thoughts

Extend the Scale of Convenient Data: The dask collections (`array`, `bag`, `dataframe`) provide reasonable access to parallelism and out-of-core execution. These significantly extend the scale of data that is convenient to manipulate.

Low Barrier to Entry: More importantly these collections demonstrate the feasibility of dask graphs to describe parallel algorithms and of the dask schedulers to execute those algorithms efficiently in a small space. The lack of a more baroque framework drastically reduces the barrier to entry and the ability of developers to use dask within their own libraries.

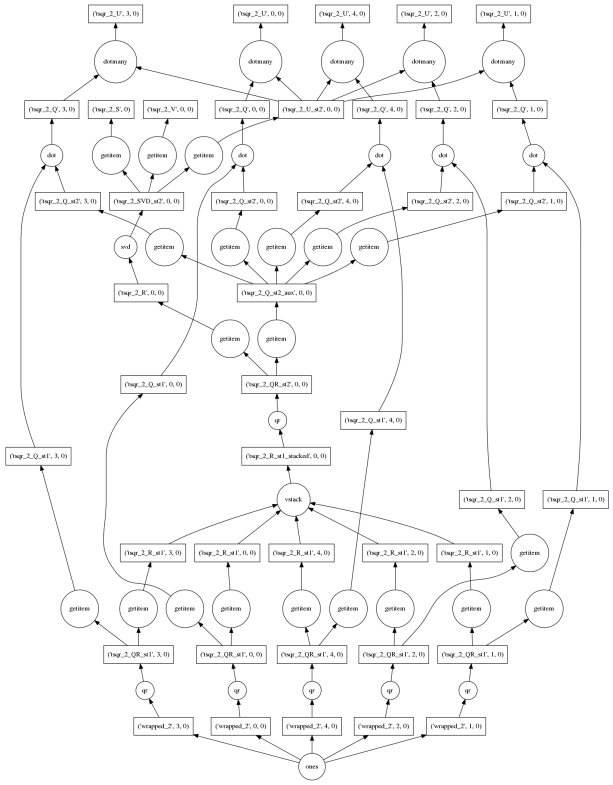


Fig. 4: *Out-of-core parallel SVD*

Administratriva and Links

Dask is available on github, PyPI, and is now included in the Anaconda distribution. It is BSD licensed, runs on Python 2.6 to 3.4 and is tested against Linux, OSX, and Windows.

This document was compiled from numerous blogposts that chronicle dask's development and go more deeply into the computational concerns encountered during dask's construction.

Dask is used on a daily basis, both as a dependency in other projects in the SciPy ecosystem (xray, scikit-image, ...) and also in production in private business.

- http://dask.pydata.org/en/latest
- http://github.com/ContinuumIO/dask
- http://matthewrocklin.com/blog
- http://pypi.python.org/pypi/dask/

Acknowledgements

Dask has had several contributors, both in terms of code and in terms of active use and reporting. Some notable contributions follow (roughly ordered by chronological involvement):

- Stephan Hoyer - Patiently used and bug-fixed `dask.array`
- Erik Welch - Implemented many of the graph optimizations
- Mariano Tepper - Implemented the `dask.array.linalg` module
- Wesley Emeneker - Worked on some of slicing
- Peter Steinberg - Worked on some of rechunking
- Jim Crist - Implemented rewrite rule optimizations
- Blake Griffith - Integrated `dask.array` with `scikit-image` and has done a variety of bug-fixing, particularly around `dask.distributed`

- Min Regan-Kelley - Provided guidance around `ZeroMQ` during the construction of `dask.distributed`
- Phillip Cloud - Improved `dask.dataframe`

REFERENCES

[Oli07] Travis E. Oliphant. Python for Scientific Computing, Computing in Science & Engineering, 9, 10-20 (2007), DOI:10.1109/MCSE.2007.58

[vdW11] 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)

[McK10] Wes McKinney. Data Structures for Statistical Computing in Python, Proceedings of the 9th Python in Science Conference, 51-56 (2010)

[Isa07] Isard, Michael, et al. "Dryad: distributed data-parallel programs from sequential building blocks." ACM SIGOPS Operating Systems Review. Vol. 41. No. 3. ACM, 2007.

[Zah10] Zaharia, Matei, et al. "Spark: cluster computing with working sets." Proceedings of the 2nd USENIX conference on Hot topics in cloud computing. Vol. 10. 2010. APA

[But09] Buttari, Alfredo, et al. "A class of parallel tiled linear algebra algorithms for multicore architectures." Parallel Computing 35.1 (2009): 38-53. APA

[Bos12] Bosilca, George, et al. "DAGuE: A generic distributed DAG engine for high performance computing." Parallel Computing 38.1 (2012): 37-51. APA

[Van08] Van De Geijn, Robert A., and Enrique S. Quintana-Ortí. "The science of programming matrix computations." (2008). APA

[Pou13] Poulson, Jack, et al. "Elemental: A new framework for distributed memory dense matrix computations." ACM Transactions on Mathematical Software (TOMS) 39.2 (2013): 13. APA

[Tep15] Mariano Tepper and Guillermo Sapiro, "Compressed Nonnegative Matrix Factorization is Fast and Accurate", 2015.

[Agu09] Agullo, Emmanuel, et al. "Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects." Journal of Physics: Conference Series. Vol. 180. No. 1. IOP Publishing, 2009. APA

[Gei08] Van De Geijn, Robert A., and Enrique S. Quintana-Ortí. "The science of programming matrix computations." (2008). APA

[Ber10] Bergstra, James, et al. "Theano: A CPU and GPU math compiler in Python." Proc. 9th Python in Science Conf. 2010. APA

[Pow14] Power, Russell. Abstractions for In-memory Distributed Computation. Diss. New York University, 2014. APA

PySPLIT: a Package for the Generation, Analysis, and Visualization of HYSPLIT Air Parcel Trajectories

Mellissa Cross‡*

https://www.youtube.com/watch?v=2mzhTC4Kp-Y

Abstract—The National Oceanic and Atmospheric Administration (NOAA) Air Resources Laboratory's HYSPLIT (HYbrid Single Particle Lagrangian Transport) model [Drax98], [Drax97] uses a hybrid Langrangian and Eulerian calculation method to compute air parcel trajectories and particle dispersion and deposition simulations. Air parcels are hypothetical small volumes of air with uniform characteristics. The HYSPLIT model outputs air parcel paths projected forwards or backwards in time (trajectories) and is used in a variety of scientific contexts. Here we present the first package in the mainstream scientific Python ecosystem designed to facilitate HYSPLIT trajectory analysis workflow by providing an intuitive API for generating, inspecting, and plotting trajectory paths and data.

Index Terms—HYSPLIT, trajectory analysis, matplotlib Basemap

Introduction

Government agencies and researchers use the HYSPLIT system, particularly the particle dispersion simulations, for academic and emergency response purposes such as monitoring nuclear fallout, the dispersion of volcanic aerosols, and dust storms. Trajectory simulations are also applied to a variety of tasks, including visualizing regional atmospheric circulation patterns, investigating meteorological controls on the isotopic composition of precipitation, and calculating moisture uptake and transport. HYSPLIT can also be applied to non-academic uses, such as ballooning. The HYSPLIT model is available online via the Real-time Environmental Applications and Display sYstem (READY) interface [Rolph03] - and has been since the late 1990s - or as a downloadable version compatible with PC or Mac [Drax03].

A key component of air parcel trajectory research problems is the along-trajectory data that HYSPLIT outputs. Although the PC and Mac versions allow for greater batch processing than is available via the online READY interface, neither interface provides users with a means to inspect, sort, or analyze trajectories on the basis of along-trajectory data. Users are left with limited options: write their own scripts for performing the desired data analysis, or manage trajectory data by hand via spreadsheet and GIS programs. Both options are inefficient and error-prone. Additionally, HYSPLIT ships with limited inbuilt options for trajectory visualization, though it does provide a shapefile/KML output tool. Using a non-Python based workflow, a figure similar to the third panel in Figure 2 took approximately three weeks to generate. This process involved manually generating a couple hundred trajectories, sorting out the rainy trajectories and importing their data into Excel, calculating moisture flux, and arranging the data for a third party program to convert to a KML file to view on Google Earth. In contrast, after two nights of letting PySPLIT generate trajectories (totalling 60,000 trajectories), the complete Figure 2 was made in a single afternoon.

PySPLIT is a Python-based tool for HYSPLIT trajectory analysis available on Github under a modified BSD license. This package's key aim is to provide an open source, reusable, reproducible, flexible system for a Python-based trajectory analysis workflow. Though a Python-based HYSPLIT frontend (physplit) is available on Google Code, this code is poorly documented and organized, and is incomplete, unmaintained, an not reusable, as it contains hard-coded variables specific to particular workstations.

PySPLIT depends on NumPy [NumPy], matplotlib [matplotlib], and the matplotlib Basemap toolkit; and comprises five classes and a trajectory generation toolkit. The scope of this package is currently bulk trajectory generation, trajectory data analysis and management, and path and data visualizations.

The API

The current PySPLIT API comprises five classes, four of which deal with trajectory data. The fundamental class of PySPLIT is the `Trajectory`; each `Trajectory` instance represents one HYSPLIT air parcel trajectory. Three of the other classes, `TrajectoryGroup`, `Cluster`, and `ClusterGroup`, are essentially variations on a `Trajectory` container. The fifth data type is `MapDesign`, which is not a `Trajectory`-related class, but holds map construction information and draw a map, using the matplotlib Basemap toolkit command. This class was included to enable the user to quickly create attractive maps without detracting focus from the trajectory analysis workflow.

Trajectory Generation

Typically the first step in a HYSPLIT workflow is trajectory generation. This can be accomplished via the online READY interface, or the HYSPLIT GUI, or command line, but bulk generation is inefficient. Additionally, READY users are limited to 500 trajectories per day. PySPLIT includes a method for generating large numbers of trajectories of a particular length in hours at various times of day and at several different altitudes in a single call, allowing the user to set up a comprehesive batch to run overnight without constant user monitoring or action:

* *Corresponding author: cros0324@umn.edu, mellissa.cross@gmail.com*
‡ *Department of Earth Sciences, University of Minnesota*

```
generate_trajectories(
    'example', r'C:/hysplit4/working',
    r'C:/traj_dir', r'E:/meteorology',
    [2007, 2008, 2009], [6, 7], [5, 11, 17, 23],
    [500, 1500], (32.29, 119.05), -120,
    meteo_type='gdas1')
```

In this example, 120-hour trajectories are launched at 500 and 1500 meters above ground level at 32.39 N and 119.05 E four times daily (`[5, 11, 17, 23]`) throughout June and July of 2007-2009. All HYSPLIT trajectory files created with this method have the same basename of `'example'`, followed by the altitude, season, and year, month, day, and hour in the format YYMMDDHH, for example: `example1500winter09063105`. The trajectory files are extensionless and live in the specified output directory (`'C:/traj_dir'`).

`pysplit.generate_trajectories()` currently only supports `gdas1` data, which refers to the 1 x 1 degree Global Data Assimilation System 3-hour meteorology product from the National Weather Service's National Centers for Environmental Prediction (NCEP) archived in a packed format appropriate for HYSPLIT (referred to as ARL-packed). Archived `gdas1` data is available from 2005 onwards; registered HYSPLIT users may also access forecast data (see HYSPLIT use agreement for more information concerning publishing and the redistribution of HYSPLIT model results using forecast data). Future versions of PySPLIT will support other datasets, for example ARL-packed ERA-interim data, for which decades of data are available; and other user-defined ARL-packed data sources.

PySPLIT comes with two additional features not available in the READY interface or directly through HYSPLIT. One feature enables an estimation of integration error. This error is estimated by comparing the distance between where an original trajectory begins and where a trajectory run in the opposite direction starting at the endpoint of the original trajectory ends. We expect the paths of the trajectories to be identical, but HYSPLIT uses finite-precision arithmetic, so there is some deviation. Low integration error is indicated by a short distance between the original trajectory start and the reverse trajectory end points relative to the total distance covered by the trajectory pair. During trajectory generation (unless disabled), PySPLIT automatically opens a new trajectory file, reads in the altitude, longitude, and latitude of the last time point, and initializes the reverse trajectory. Then in the `Trajectory` class, discussed below, a method is available to estimate integration error.

The second feature facilitates HYSPLIT clustering. HYSPLIT trajectory data files are plaintext with a limited number of characters per line. Typically, each timepoint is recorded on a single line. However, there are nine possible along-trajectory meteorological output variables, and if more than seven are selected, each timepoint overflows onto a second line. Timepoints will span multiple lines, however, if more than seven of nine possible available output variables are selected. HYSPLIT's clustering method fails given files with multi-line timepoints, but PySPLIT can account for this when it occurs. `pysplit.clip_traj()` opens a trajectory file, copies the trajectory header and path (latitude, longtitude, altitude) data, and outputs the header and path to a new file that HYSPLIT will readily use to perform clustering, as HYSPLIT clusters solely on the basis of path. The clipped and reverse trajectories live in subdirectories inside the output directory.

Trajectory

The `Trajectory` class is the fundamental unit in PySPLIT, designed to manage and promote the analysis of air parcel trajectory data in an intuitive manner. Each object represents one air parcel trajectory calculated by HYSPLIT, containing latitude, longitude, altitude (meters above ground level or meters above sea level), along-path data, file location, path start information, and summary data. `Trajectory` instances are intialized as follows:

```
traj = Trajectory(data, header, fullpath)
```

where `data` is the 2D array of along-trajectory data read by PySPLIT from the HYSPLIT output file (using `pysplit.load_hysplitfile()`), `header` is a list of strings indicating the information present in each column, and `fullpath` is the location of the output file. However, the user will typically not initialize individual `Trajectories`, but will instead initialize a `TrajectoryGroup` that contains them.

The 2D `data` array of a `Trajectory` is parsed into separate attributes as 1D NumPy arrays of floats, readily exposing the data. The `data` and `header` are also kept as attributes, and can be reloaded into the corresponding 1D attributes at any time, wiping out changes.

Most `Trajectory` analysis methods live in or are accessed directly by the `Trajectory` class. These include calculations of along-trajectory and overall great-circle distance, mean trajectory vector, humidity data conversions, and along-trajectory moisture flux. The results of most of these calculations are stored as new attributes in 1D NumPy arrays of floats of identical size. Additionally, the `Trajectory` class contains the methods for loading forward trajectories and estimating trajectory integration error in both horizontal and vertical dimensions.

The `Trajectory` class also includes a flexible implementation of the moisture uptake calculation from back trajectories from Sodeman et al. [Sod08].

```
moistureuptake(self, rainout_threshold,
               evap_threshold, uptake_window=6,
               window_overlap=0,
               vertical_criterion='pbl',
               pressure_threshold=900.0,
               mixdepth_factor=1,
               q_type='specific_humidity')
```

Using this method, humidity is compared at the beginning and end of a period of time with length `uptake_window`, repeated over the whole back trajectory, from the earliest timepoint to the most recent time point. A good uptake window is 6 hours, since it is a short enough period of time that evaporation or precipitation will dominate, and long enough that performing this calculation over 120-hour trajectories is not particularly onerous.

The purpose of this algorithm is to find moisture sources on the Earth's surface that contribute to the moisture received at the starting location of the backwards trajectory. So, a criterion to distinguish surficial from atmospheric moisture sources is required. In Sodeman's original paper, which did not use HYSPLIT, uptakes that occurred below the planetary boundary level were regarded as uptakes from the Earth's surface. In other works that have used this algorithm but employed HYSPLIT, a particular pressure level, often 900.0 hPa, is as the boundary between uptake from the surface and uptake from the atmosphere. In PySPLIT, the user can choose for their boundary the mixing depth, a pressure level, or both.

For each window, PySPLIT records the coordinates of the midpoint, the mean pressure, mixing depth (if available), and

altitude; the change in humidity; and the fraction of current humidity levels taken up below or above the vertical criteria or due to unknown sources. Change in humidity and humidity fractions in previous windows are also adjusted to reflect rainout and the fact that early sources of moisture become less important as moisture is acquired further along in the trajectory. The result is a 2D array of moisture uptake data where each row represents a time window and each column a variable. The array and header are stored as `Trajectory` attributes.

TrajectoryGroup

The `TrajectoryGroup` is the basic container for PySPLIT `Trajectory` objects, and is initialized simply by providing a list of `Trajectory` objects. Typically the first `TrajectoryGroup` in a PySPLIT workflow is initialized upon loading `Trajectory` objects from file as discussed above:

```
trajgroup, _ = make_trajectorygroup(signature)
```

In this method, HYSPLIT output files sharing a Bash-style signature (with wildcards supported) are read, initialized as `Trajectory`s and appended to a list, which is then used to initialize a `TrajectoryGroup`. These containers are fully iterable, returning a `Trajectory` when indexed and a new `TrajectoryGroup` when sliced.

Once the initial `TrajectoryGroup` is created, a typical PySPLIT workflow involves cycling through the `TrajectoryGroup` (`umn` in the example workflow below), initializing and inspecting attributes of the member `Trajectory` instances. `Trajectory`s that meet certain criteria are then sorted into new `TrajectoryGroups`.

```
# Trajectories with integration error better than 10%
good_traj = []

for traj in umn:
  traj.load_reversetraj(r'C:/traj/reversetraj')
  traj.integration_error()

  if self.integ_error_xy < 10:
     good_traj.append(traj)

# Sort out rain-bearing traj starting at 1700 UTC
# (local noon) and 1500 m
umn_trajls=[]

for traj in good_traj:
  traj.set_rainstatus()
  if (traj.rainstatus and traj.hour[0] == 17 and
      traj.altitude[0] == 1500):
    umn_trajls.append(traj)

# Create new TrajectoryGroup:
umn_noon = pysplit.TrajectoryGroup(umn_trajls)
```

And perform more calculations:

```
for traj in umn_noon:
  traj.set_vector()
  traj.set_specifichumidity()
  traj.calculate_moistureflux()
```

Repeating sorting and analysis as necessary.

Using the visualization defaults as described in the Data Plotting and MapDesign section below, we can quickly look at the `Trajectory` paths, as seen in Figure 1.

```
mapd = pysplit.MapDesign([40.0, -15.0, 170.0, 60.0],
                         [100.0, 20.0, 30.0, 10.0])

umap = mapd.make_basemap()
```

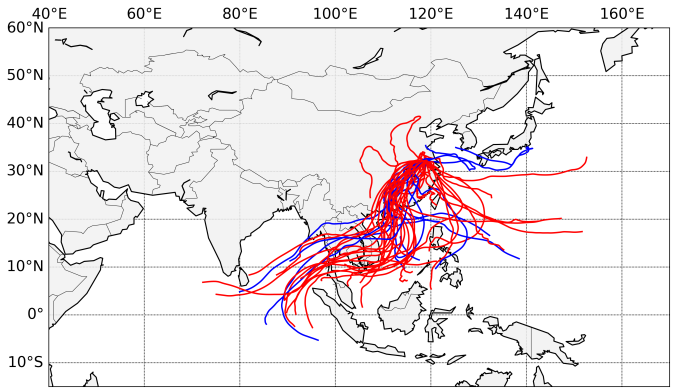


Fig. 1: *Simple visualization of trajectory paths using MapDesign defaults (see Data Plotting and MapDesign section) . Red indicates June trajectories, blue indicates July trajectories.*

```
for traj in umn_noon:
 if traj.month[0] == 6:
   traj.trajcolor == 'blue'
 else:
   traj.trajcolor == 'red'

umn_noon.map_data_line(umap)
```

The `TrajectoryGroup` class also has additional capabilities for organizing `Trajectory` instances and `Trajectory` data. `TrajectoryGroup` instances are additive: two instances are checked for duplicte trajectories (determined by examining the filename and path) and can be combined into a new group of unique trajectories. The `TrajectoryGroup` also comes with methods for assembling particular member `Trajectory` attributes and moisture uptake arrays into a single array to facilitate scatter plotting and for interpolating along-path and moisture uptake data to a grid. The procedure is given below and the results are shown in These are discussed below in the Data Plotting and MapDesign section.

Cluster and ClusterGroup

To investigate the dominant flow patterns in a set of trajectories, HYSPLIT includes a clustering procedure. PySPLIT includes several methods to expedite this process.

The first step is to generate a list of trajectories to be clustered. Once the user has created a `TrajectoryGroup` with trajectories that meet their specifications, then they can use the `TrajectoryGroup` method `make_infile()` to write member `Trajectory` full paths to an extensionless file called 'INFILE' that HYSPLIT requires to perform clustering. PySPLIT will attempt to write the full paths of the clipped versions of the trajectories to INFILE, if available, otherwise the full paths of the regular trajectories will be used. Clipped trajectories are usually generated during trajectory generation, as discussed above. However, as clipping does not actually require calculating a new trajectory this can be performed later:

```
for traj in trajgroup:
  clip_traj(traj.folder, traj.filename)
```

However, the `TrajectoryGroup` (`trajgroup`) and its member `Trajectories` must be reloaded for the clipped trajectory files to become available for clustering.

Once the INFILE is created, the user must open HYSPLIT to run the cluster analysis and assign trajectories to clusters. Advice

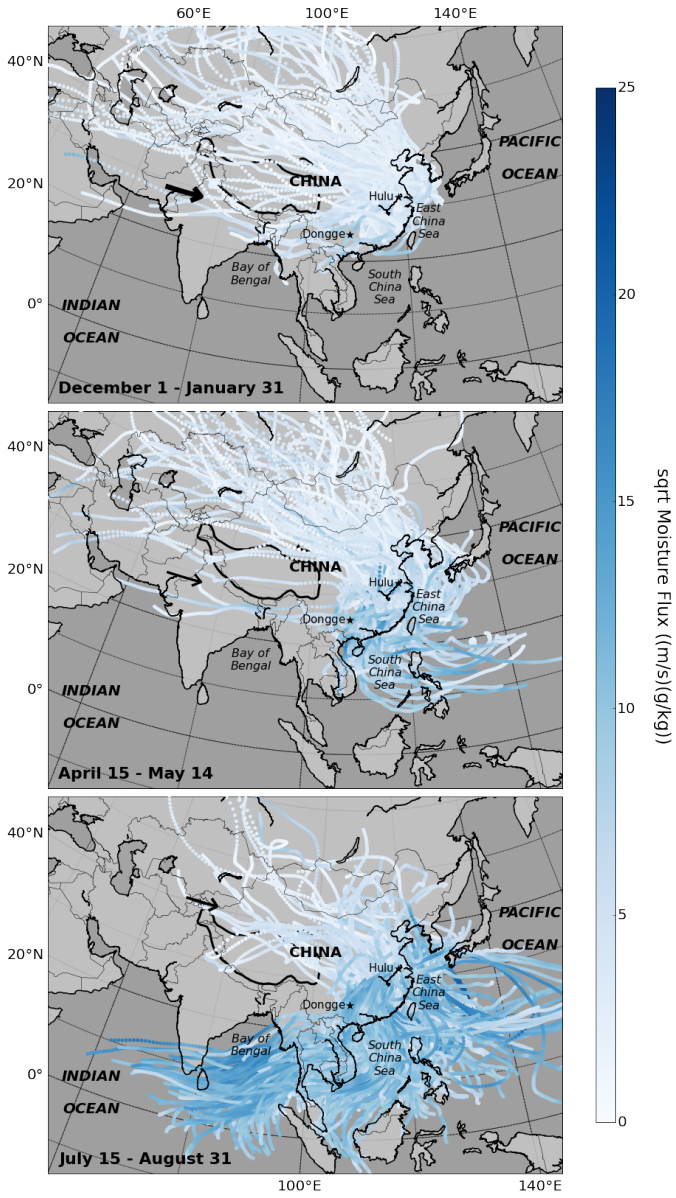


Fig. 2: *Visualization of seasonal moisture flux. Place labels are generated with the labeller in* `MapDesign` *, discussed in Data Plotting and MapDesign section.*

concerning the determination of the number of clusters (along with all other HYSPLIT aspects) is available in the HYSPLIT manual [Drax97]. Assigning trajectories to clusters will create a file called 'CLUSLIST_3' or some other number corresponding to the number of clusters specified by the user. This file indicates the distribution of `Trajectory` in the `TrajectoryGroup` among clusters, and is used to create `Cluster` instances contained in a `ClusterGroup`:

```
clusgroup = spawn_clusters(trajgroup, traj_distrib,
                           clusterpath_dir)
```

The `Cluster` class is a specialized subclass of `TrajectoryGroup`. In addition to a list of member `Trajectorys` (indicated by the distribution file), initialization requires the cluster mean path data and cluster index. Like `TrajectoryGroups`, `Clusters` are additive, but adding Clusters creates a regular `TrajectoryGroup`, not a new `Cluster`. As a `Cluster` has an associated path, some `Trajectory`-like methods (distance, vector calculations) are available.

A `ClusterGroup` is a container of `Clusters` produced in a *single* clustering procedure. Iterating over a `ClusterGroup` returns member `Clusters`.

Data Plotting and MapDesign

As visualization and figure creation is a key part of the scientific process, a major focus of PySPLIT is exposing data and enabling the user to create attractive maps and plots.

One part of this equation is the `MapDesign` class. A `MapDesign` instance holds the information necessary to create an attractive matplotlib Basemap. The user provides the coordinates of the lower left and upper right corners of the map, as well as a few standard parallels and meridians. From there, the defaults are sufficient to produce a professional-looking map as shown in Figure 1. Users can also choose between two additional neutral color-schemes, as shown in Figures 2, and 3.

`MapDesign` also encompasses more complex formatting like labelling, as shown in Figure 2. During the initialization of `MapDesign`, or later using `MapDesign.edit_labels()`, the user can generate a text file with example labels in defined label categories at a given file location. The user can then edit the example labels for their needs, and select which groups are placed on the map, once `MapDesign.make_basemap()` is called and a Basemap is generated.

Although `MapDesign` was created to expedite the process of creating an attractive Basemap and let users focus on the trajectory analysis rather than figure-tweaking, PySPLIT plotting functions accept any Basemap instance, allowing users to incorporate PySPLIT into their existing workflow. Additionally, as all `Trajectory`, `Cluster`, `TrajectoryGroup`, and `ClusterGroup` attributes are exposed, users are free to create their own visualization routines beyond what is provided in PySPLIT.

Among the `Trajectory` attributes are linewidth and path color. A user can incorporate these into their plotting workflow, setting linewidth and path color to correspond to `Trajectory` instances with particular characteristics, as shown in Figure 1. Plotting the paths of a `TrajectoryGroup`'s member `Trajectorys` is performed one-by-one on the given map. To facilitate scatter plotting, the `TrajectoryGroup` assembles `Trajectory` latitude, longtitude, the variable plotted as a color change, and, if selected, the variable plotted as a size change each into single arrays. `Trajectory` data, as well as moisture uptake data, can also be interpolated onto a grid and plotted.

Prior to being passed to `Basemap.plot()` and `Basemap.scatter()`, scatter plot data passes through `traj_scatter()`. This exposes Normalize instances and other methods of normalization (square root, natural log), allowing users to normalize both color and size data. Square root and natural log normalizations require the user to edit tick labels on colorbars (or incorporate into the colorbar label itself, as in Figure 2). After plotting, wrappers around matplotlib's colorbar creation methods with attractive default options are available to initialize colorbars.

As a `Cluster` is a specialized `TrajectoryGroup`, member `Trajectorys` can be plotted similarly. Additionally, `Cluster` mean paths can also be plotted, either individually or all

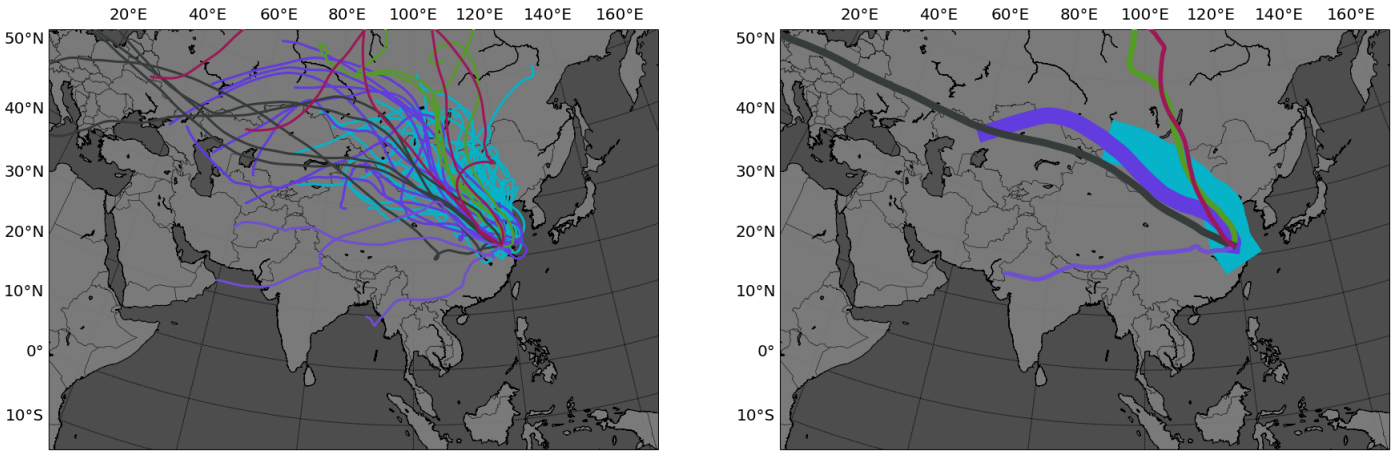


Fig. 3: *Left: Winter back trajectories arriving at Nanjing, colored to match the cluster they belong to. Right: Plot of* `ClusterGroup` *in which member* `Clusters` *have randomly-chosen colors and linewidths corresponding to their* `Trajectory` *counts.*

together in the `ClusterGroup`. `Cluster` linewdiths can either be determined by an absolute `Trajectory` count or the fraction of total `Trajectorys` in the `ClusterGroup` belonging to the `Cluster`. Both `Cluster` and `Trajectory` paths shown in Figure 3.

The Future of PySPLIT

PySPLIT provides an intuitive API for extremely efficient HYSPLIT trajectory data processing and for creating visualizations using matplotlib and the matplotlib Basemap toolkit. The goal of PySPLIT is to provide users with a powerful, flexible Python-oriented HYSPLIT trajectory analysis workflow, and in the long-term to become the toolkit of choice for research using HYSPLIT. Features in the pipeline include HYSPLIT clustering process entirely accessible via the PySPLIT interface, and a greater variety of statistical, moisture uptake, and other methods available for trajectory analysis. Additionally, there are several areas for improvement within the trajectory generation portion of PySPLIT, notably support for meteorologies besides `gdas1`, more granular trajectory generation, and generation on pressure and condensation levels.

Acknowledgments

I gratefully thank the reviewers for their patience, comments, and suggestions; and the NOAA ARL for the provision of the HYSPLIT transport and dispersion model.

REFERENCES

[Sod08] H. Sodeman, C. Schwierz, and H. Wernli. *Interannual Variability of Greenland winter precipitation sources: Lagrangian moisture diagnostic and North Atlantic Oscillation influence*, Journal of Geophysical Research, 113:D03107, February 2008.

[Drax98] R.R. Draxler and G.D. Hess. *An overview of the HYSPLIT_4 modeling system of trajectories, dispersion, and deposition*, Aust. Meteor. Mag., 47:295-308, 1998.

[Drax97] R.R. Draxler and G.D. Hess. *Description of the HYSPLIT_4 modeling system*, NOAA Technical Memorandum ERL ARL-230, NOAA Air Resources Laboratory, Silver Spring, MD, 1997.

[Drax03] R.R. Draxler and G.D. Rolph. HYSPLIT (HYbrid Single-Particle Lagrangian Integrated Trajectory) Model access via NOAA ARL READY Website (http://www.arl.noaa.gov/ready/hysplit4.html). NOAA Air Resources Laboratory, Silver Spring, MD, 2003.

[Rolph03] G.D. Rolph. Real-time Environmental Applications and Display sYstem (READY) Website (http://www.arl.noaa.gov/ready/hysplit4.html). NOAA Air Resources Laboratory, Silver Spring, MD, 2003.

[NumPy] S. van der Walt et al. *The NumPy Array: A Structure for Efficient Numerical Computation*, Computing in Science & Engineering, 13:22-30, 2011.

[matplotlib] J. D. Hunter. Matplotlib: A 2D Graphics Environment*, Computing in Science & Engineering, 9:90-95, 2007.

TrendVis: an Elegant Interface for dense, sparkline-like, quantitative visualizations of multiple series using matplotlib

Mellissa Cross‡*

https://www.youtube.com/watch?v=tklAFsce7eg

Abstract—TrendVis is a plotting package that uses matplotlib to create information-dense, sparkline-like, quantitative visualizations of multiple disparate data sets in a common plot area against a common variable. This plot type is particularly well-suited for time-series data. We discuss the rationale behind and the challenges associated with adapting matplotlib to this particular plot style, the TrendVis API and architecture, and various features available for users to customize and enhance the readability of their figures while walking through a sample workflow.

Index Terms—time series visualization, matplotlib, plotting

Introduction

Data visualization and presentation is a key part of scientific communication, and many disciplines depend on the visualization of multiple time-series or other series datasets. The field of paleoclimatology (the study of past climate and climate change), for example, relies heavily on plots of multiple time-series or "depth series", where data are plotted against depth in an ice core or stalagmite, for example. These plots are critical to place new data in regional and global contexts and they facilitate interpretations of the nature, timing, and drivers of climate change. Figure 1, created using TrendVis, compares stalagmite records of climate and hydrological changes that occurred during the last two deglaciations, or "terminations". Ice core records of carbon dioxide (black) and methane (pink) [Petit] concentrations and Northern Hemisphere summer insolation (the amount of solar energy received on an area, gray) are also included.

Creating such plots can be difficult, however. Many scientists depend on expensive software such as SigmaPlot and Adobe Illustrator. With pure matplotlib [matplotlib], users have two options: display data in a grid of separate subplots or overlaid using twinned axes. This works for two or three traces, but does not scale well. The ideal style in cases with larger datsets is the style shown in Figure 1: a densely-plotted figure that facilitates direct comparison of curve features. The key aim of TrendVis, available on GitHub, is to enable the creation and readability of

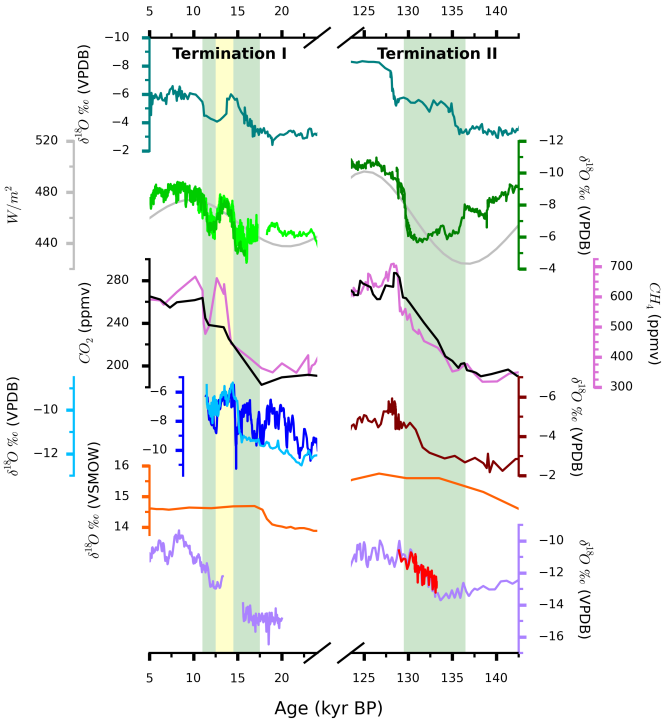


Fig. 1: *A TrendVis figure illustrating the similarities and differences among climate records from Israel [BarMatthews], China [Wang], [Dykoski], [Sanbao]; Italy [Drysdale], the American Southwest [Wagner], [Asmerom], and Great Basin region [Winograd0], [Winograd1], [Lachniet], [Shakun] between the last deglaciation and the penultimate deglaciation (respectively known as Termination I and Termination II). Most of these records are stalagmite oxygen isotope records - oxygen isotopes, depending on the location, may record temperature changes, changes in precipitation seasonality, or other factors. All data are available online as supplementary materials or through the National Climatic Data Center.*

* *Corresponding author: cros0324@umn.edu, mellissa.cross@gmail.com*
‡ *Department of Earth Sciences, University of Minnesota*

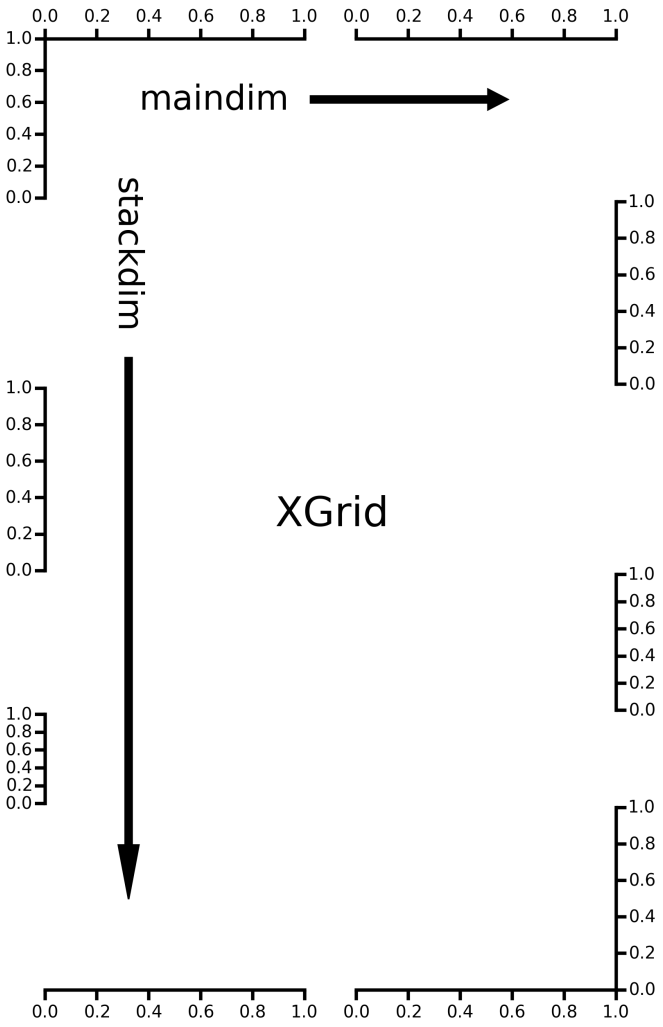


Fig. 2: *In* `XGrid`*,* `stackdim` *refers to number of rows of y axes and* `maindim` *indicates the number of columns. This is reversed in* `YGrid`*. Both dimension labels begin in* `XGrid.axes[0][0]`*.*

these plots in the scientific Python ecosystem using a matplotlib-based workflow. Here we discuss how TrendVis interfaces with matplotlib to construct and format this complex plot type as well as several challenges faced while we walk through the creation of Figure 1.

The TrendVis Figure Framework

The backbone of TrendVis is the `Grid` class, in which the figure, basic attributes, and orientation-agnostic methods are initialized. `Grid` should only be initialized through one of its two subclasses, `XGrid` and `YGrid`. As a common application of these types of plots is time-series data, we will examine TrendVis from the perspective of `XGrid`. In `XGrid`, the x axis is shared among all the datasets, and y axes are individual - in the terminology of TrendVis, x axes are the main axes, and y axes are the stacked axes. This is reversed for `YGrid`. A graphical representation of `XGrid` is shown in Figure 2.

TrendVis figures appear to consist of a common plot space. This, however, is an illusion carefully crafted via a framework of axes and a mechanism to systematically hide extra axes spines, ticks, and labels. This framework is created when the figure is initialized:

```
paleofig = XGrid([7, 8, 8, 6, 4, 8], xratios=[1, 1],
                 figsize=(6,10))
```

First, let's examine the construction of this framework. The overall area of the figure is determined by `figsize`, which is passed to matplotlib. The relative sizes of the rows (`ystack_ratios`, the first argument), however, is determined by the contents of `ystack_ratios` and the sum of `ystack_ratios` (`self.gridrows`), which in this case is 41. Similarly, the contents and sum of `xratios` (`self.gridcols`) determine the relative sizes of the columns. So, all axes in `paleofig` are initialized on a 41 row, 2 column grid within the 6 x 10 inch space set by `figsize`. The axis in position 0,0, (2) spans 7/41 unit rows (0 through 6) and the first unit column; the next axis created spans the same unit rows and the second unit column, finishing the first row of `paleofig`. The next row spans 8 unit rows, numbers 7 through 15, and so on. All axes in the same row share a y axis, and all axes in the same column share an x axis. This axes creation process, shown in the code below, is repeated for all the values in `ystack_ratios` and `xratios`, yielding a figure with 6 rows and 2 columns of axes. The code below and all other unnumbered snippets indicate an internal process rather than part of the `paleofig` workflow.

```
xpos = 0
ypos = 0

# Create axes row by row
for rowspan in self.yratios:
    row = []

    for c, colspan in enumerate(self.xratios):
        sharex = None
        sharey = None

        # All ax in row share y with first ax in row
        if xpos > 0:
            sharey = row[0]

        # All ax in col share x with first ax in col
        if ypos > 0:
            sharex = self.axes[0][c]

        ax = plt.subplot2grid((self.gridrows,
                               self.gridcols),
                              (ypos, xpos),
                              rowspan=rowspan,
                              colspan=colspan,
                              sharey=sharey,
                              sharex=sharex)

        ax.patch.set_visible(False)

        row.append(ax)
        xpos += colspan

    self.axes.append(row)

    # Reset x position to left, move to next y pos
    xpos = 0
    ypos += rowspan
```

Axes are stored in `paleofig.axes` as a nested list, where the sublists contain axes in the same rows. Next, two parameters that dictate spine visibility are initialized:

`paleofig.dataside_list`
: This list indicates where each row's y axis spine, ticks, and label are visible. This by default alternates sides from left to right (top to bottom in `YGrid`), starting at left, unless indicated otherwise during the

initialization of `paleofig`, or changed later on by the user.

`paleofig.stackpos_list`
: This list controls the x (main) axis visibility. Each row's entry is based on the physical location of the axis in the plot; by default only the x axes at the top and bottom of the figure are shown and the x axes of middle rows are invisible. Each list is exposed and can be user-modified, if desired, to meet the demands of the particular figure.

These two lists serve as keys to TrendVis formatting dictionaries and as arguments to axes (and axes child) methods. At any point, the user may call:

```
paleofig.cleanup_grid()
```

and this method will systematically adjust labelling and limit axis spine and tick visibility to the positions indicated by `paleofig.dataside_list` and `paleofig.stackpos_list`, transforming the mess in Figure 3 to a far clearer and more readable format in Figure 2.

Creating Twinned Axes

Although for large datasets, using twinned axes as the sole plotting tool is unadvisable, select usage of twinned axes can improve data visualization. In the case of `XGrid`, a twinned axis is a new axis that shares the x axis of the original axis *but* has a different y axis on the opposite side of the original y axis. Using twins allows the user to directly overlay datasets. TrendVis provides the means to easily and systematically create and manage entire rows (`XGrid`) or columns (`YGrid`) of twinned axes.

In our `paleofig`, we need four new rows:

```
paleofig.make_twins([1, 2, 3, 3])
paleofig.cleanup_grid()
```

This creates twinned x axes, one per column, across the four rows indicated and hides extraneous spines and ticks, as shown in Figure 4. As with the original axes, all twinned axes in a column share an x axis, and all twinned axes in the twin row share a y axis. The twin row information is appended to `paleofig.dataside_list` and `paleofig.stackpos_list` and twinned axes are stored at the end of the list of axes, which previously contained only original rows. If the user decides to get rid of twin rows (`paleofig.remove_twins()`), `paleofig.axes`, `paleofig.dataside_list`, and `paleofig.stackpos_list` are returned to their state prior to adding twins.

Accessing Axes

Retrieving axes, especially when dealing with twin axes in a figure with many hapazardly created twins, can sometimes be non-straightforward. The following means are available to return individual axes from a TrendVis figure:

`paleofig.fig.axes[axes index]`
: Matplotlib stores axes in a 1D list in `Figure` in the order of creation. This method is easiest to use when dealing with an `XGrid` of only one column.

`paleofig.axes[row][column]`
: An `XGrid` stores axes in a nested list in the order of creation, no matter its dimensions. Each sublist contains all axes that share the same y axis- a row. The row index corresponds to the storage position in the list, not the actual physical position on the grid, but in original axes (those created when `paleofig` was initialized) these are the same.

`paleofig.get_axis()`
: Any axis can be retrieved from `paleofig` by providing its physical row number (and if necessary, column position) to `paleofig.get_axis()`. Twins can be parsed with the keyword argument `is_twin`, which directs `paleofig.twin_rownum()` to find the index of the sublist containing the twin row.

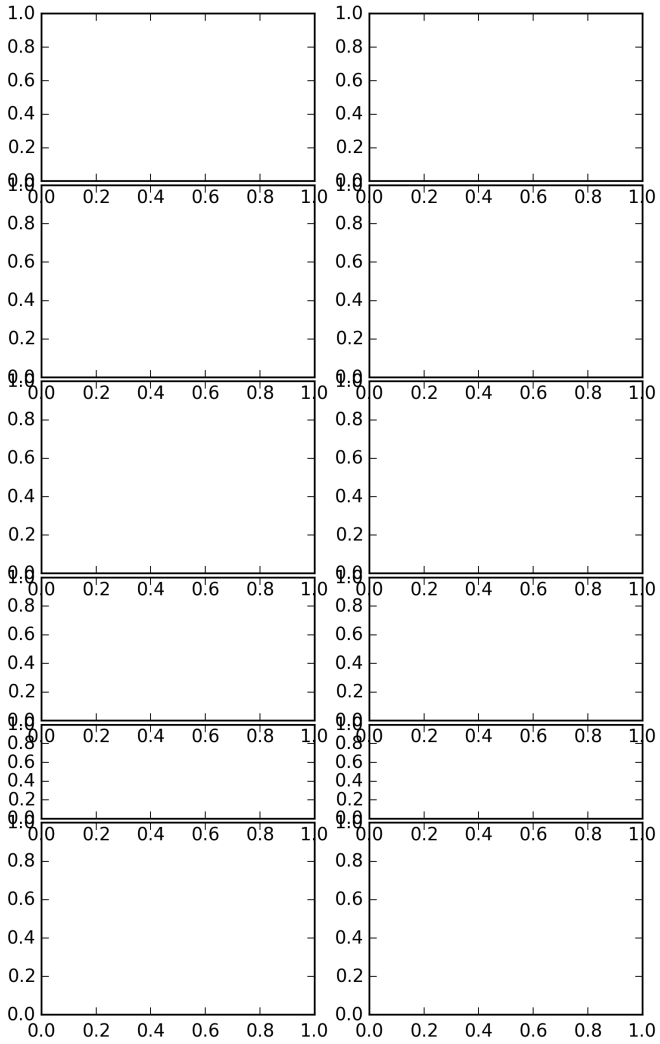


Fig. 3: *Freshly initialized `XGrid`. After running `XGrid.cleanup_Grid()` (and two formatting calls adjusting the spinewidth and tick appearance), the structure of Figure 2 is left, in which stack spines are staggered, alternating sides according to `XGrid.dataside_list`, starting at left.*

In the case of `YGrid`, the row, column indices are flipped: `YGrid.axes[column][row]`. Sublists correspond to columns rather than rows.

Plotting and Formatting

The original TrendVis procedurally generated a simple, 1-column version of `XGrid`. Since the figure was made in a single function

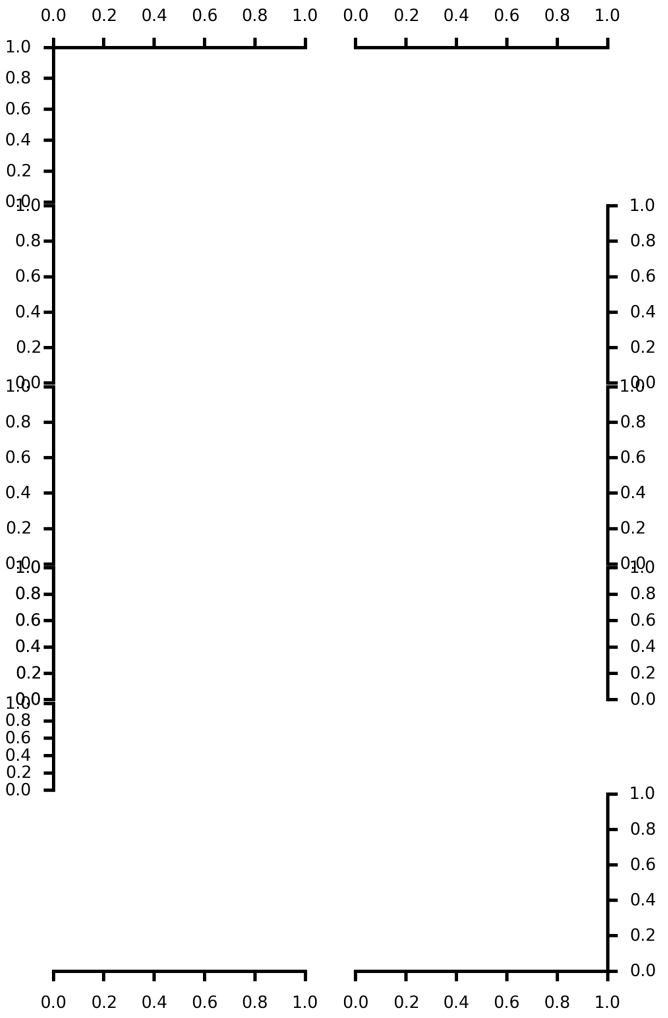


Fig. 4: *The results of* `paleofig.make_twins()`*, performing another grid cleanup and some minor tick/axis formatting.*

call, all data had to be provided at once in order, and it all had to be line/point data, as only `Axes.plot()` was called. TrendVis still provides convenience fuctions `make_grid()` and `plot_data()` to enable easy figure initialization and quick line plotting on all axes with fewer customization options. The regular object-oriented API is designed to be a highly flexible wrapper around matplotlib. Axes are readily exposed via the matplotlib and TrendVis methods described above, and so the user can determine the most appropriate plotting functions for their figure. The author has personally used `Axes.errorbar()`, `Axes.fill_betweenx()`, and `Axes.plot()` on two published TrendVis figures (see figures 3 and 4 in [Cross]), which required the new object-oriented API. Rather than make individual calls to plot on each axis, we will use the convenience function `plot_data`. The datasets have been loaded from a spreadsheet into individual 1D NumPy [NumPy] arrays containing age information or climate information:

```
plot_data(paleofig,[[(sorq_age, sorq, '#008080')],
                    [(hu_age, hu, '#00FF00',[0]),
                     (do_age, do, '#00CD00', [0]),
                     (san_age, san, 'green', [1])],
                    [(co2age, co2, 'black')],
                    [(cor_age, cor, 'maroon', [1])],
                    [(dh_age, dh, '#FF6103')],
                    [(gb_age, gb, '#AB82FF'),
                     (leh_age, leh, 'red', [1])],
                    [(insol_age, insol, '0.75')],
                    [(ch4_age, ch4, 'orchid')],
                    [(fs_age, fs, 'blue')],
                    [(cob_age, cob, '#00BFFF')]],
          marker=None, lw=2, auto_spinecolor=False)
```

Using `plot_data`, simple line plotting only requires a tuple of the x and y values and the color in a sublist in the appropriate row order. Some tuples have a fourth element that indicates which column the dataset should be plotted on. Without this element, the dataset will be plotted on all, or in this case both columns. Setting different x axis limits for each column will mask this fact.

Although plots individualized on a per axis basis may be important to a user, most aspects of axis formatting should generally be uniform. In deference to that need and to potentially the sheer number of axes in play, TrendVis contains wrappers designed to expedite these repetitive axis formatting tasks, including setting major and minor tick locators and dimensions, axis labels, and axis limits.

```
paleofig.set_ylim([(3, -7, -2), (4, 13.75, 16),
                   (5, -17, -9),
                   (6, 420, 520, (7, 300, 725),
                   (8, -11.75, -5))])

paleofig.set_xlim([(0, 5, 24), (1, 123.5, 142.5)])

paleofig.reverse_yaxis([0, 1, 3])

paleofig.set_all_ticknums([(5, 2.5), (5, 2.5)],
                          [(2,1),(2,1),(40,20),(2,1),
                           (1,0.5), (2,1),(40,20),
                           (100,25),(2,1),(2,1)])

paleofig.set_ticks(major_dim=(7, 3), labelsize=11,
                   pad=4, minor_dim=(4, 2))

paleofig.set_spinewidth(2)

# Special characters for axis labels
d18o = r'$\delta^{18}\!O$'
d13c = r'$\delta^{13}\!C$'
d234u = r'$\delta^{234}\!U_{initial}$'
co2label = r'$CO_{2}$'
ch4label = r'$CH_{4}$'
mu = ur'$\u03BC$'
vpdb = ' ' + ur'$\u2030$'+ ' (VPDB)'
vsmow =' ' + ur'$\u2030$'+' (VSMOW)'

paleofig.fig.suptitle('Age (kyr BP)', y=0.065,
                      fontsize=16)
paleofig.set_ylabels([d18o + vpdb, d18o + vpdb,
                      co2label +' (ppmv)',
                      d18o + vpdb,
                      d18o + vsmow, d18o + vpdb,
                      r'$W/m^{2}$',
                      ch4label + ' (ppmv)', '',
                      d18o + vpdb, d13c + vpdb],
                      fontsize=13)
```

In this plot style, there are two other formatting features that are particularly useful: moving data axis spines, and automatically coloring spines and ticks. The first involves the lateral movement of data axis (y axis in `XGrid`, x axis in `YGrid`) spines into or out of the plot space. Although the default TrendVis behavior is alternating the data axis spines from left to right, resulting in space between data axis spines, adding twin rows disrupts this pattern and spacing, as shown in Figure 5. This problem is exacerbated when compacting the figure, which is a typical procedure in this plot type, to improve both the look of the figure and its readability. The solution in `XGrid` plots is to move spines laterally- along the

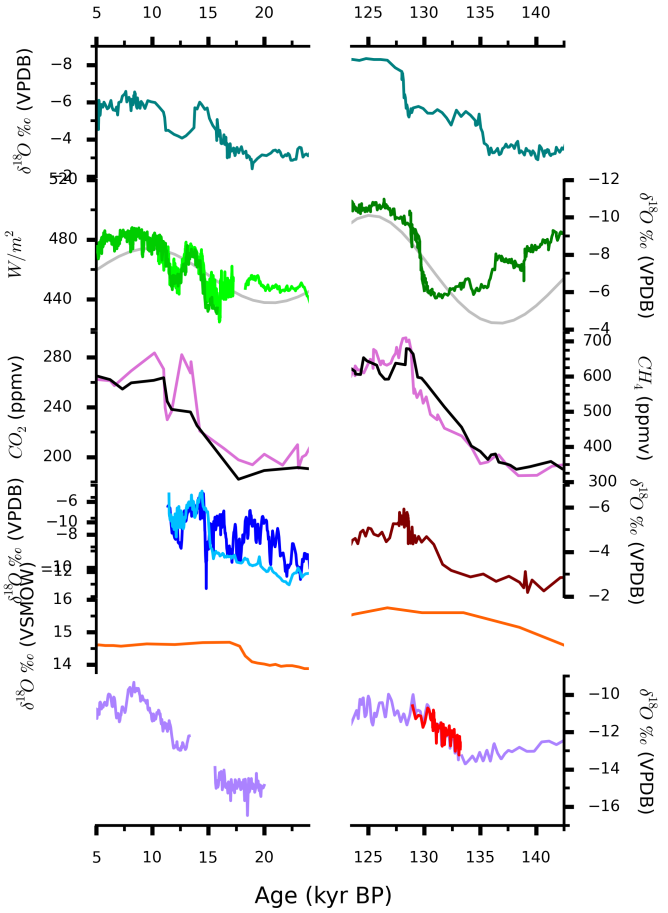


Fig. 5: *Figure after plotting paleoclimate time series records, editing the axes limits, and setting the tick numbering and axis labels. At this point it is difficult to see which dataset belongs to which axis and to clearly make out the twin axis numbers and labels.*

x dimension- out of the way of each other, into or out of the plot space. TrendVis provides means to expedite the process of moving spines:

```
# Make figure more compact:
paleofig.fig.subplots_adjust(hspace=-0.4)

# Move spines
# Shifts are in fractions of figure
# Absolute position calc as 0 - shift (ax at left)
# or 1 + shift (for ax at right)
paleofig.move_spines(twin_shift=[0.45, 0.45,
                                 -0.2, 0.45])
```

In the above code, all four of the twinned visible y axis spines are moved by an individual amount; the user may set a universal `twin_shift` or move the y axis spines of the original axes in the same way. Alternatively, all TrendVis methods and attributes involved in `paleofig.move_spines()` are exposed, and the user can edit the axis shifts manually and then see the results via `paleofig.execute_spineshift()`. As the user-provided shifts are stored, if the user changes the arrangement of visible y axis spines (via `paleofig.set_dataside()` or by directly altering `paleofig.dataside_list`), then all the user needs to do to get the old relative shifts applied to the new arrangement is get TrendVis to calculate new spine positions (`paleofig.absolute_spineshift()`) and perform the shift (`paleofig.execute_spineshift()`).

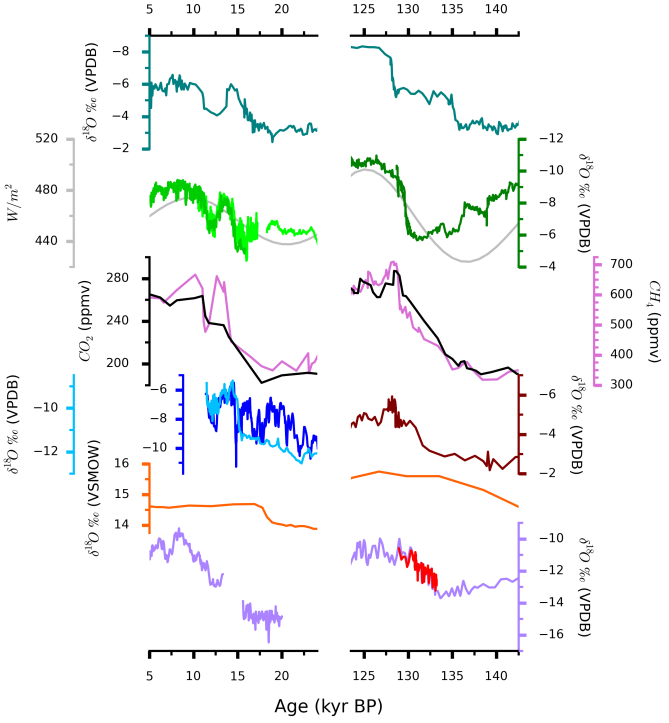


Fig. 6: *Although the plot is very dense, the lateral movement of spines and coloring them to match the curves has greatly improved the readability of this figure relative to Figure 5. The spacing between subplots has also been decreased.*

Although the movement of y axis spines allows the user to read each axis, there is still a lack of clarity in which curve belongs with which axis, which is a common problem for this plot type. TrendVis' second useful feature is automatically coloring the data axis spines and ticks to match the color of the first curve plotted on that axis. As we can see in Figure 6, this draws a visual link between axis and data, permitting most viewers to easily see which curve belongs against which axis.

```
paleofig.autocolor_spines()
```

Visualizing Trends

Large stacks of curves are overwhelming to viewers. In complicated figures, it is critical to not only keep the plot area tidy and link axes with data, as we saw above, but also to draw the viewer's eye to essential features. This can be accomplished with shapes that span the entire figure, highlighting areas of importance or demarcating particular spaces. In `paleofig`, we are interested in the glacial terminations. Termination II coincided with a North Atlantic cold period, while during Termination I there were two cold periods interrupted by a warm interval:

```
# Termination I needs three bars, get axes that will
# hold the lower left, upper right corners of bar
ll = paleofig.get_axis(5)
ur = paleofig.get_axis(0)
alpha = 0.2

paleofig.draw_bar(
 ll, ur, (11, 12.5), alpha=alpha,
 edgecolor='none', facecolor='green')
paleofig.draw_bar(
 ll, ur, (12.5, 14.5), alpha=alpha,
 edgecolor='none', facecolor='yellow')
```

```
paleofig.draw_bar(
 ll, ur, (129.5, 136.5), alpha=alpha,
 edgecolor='none', facecolor='green')

# Draw bar for Termination II, in column 1
paleofig.draw_bar(paleofig.get_axis(5, xpos=1),
                  paleofig.get_axis(0, xpos=1),
                  (129.5, 136.5), alpha=alpha,
                  facecolor='green',
                  edgecolor='none')

# Label terminations
ax2 = paleofig.get_axis(0, xpos=1)
paleofig.ax2.text(133.23, -8.5, 'Termination II',
                  fontsize=14, weight='bold',
                  horizontalalignment='center')

ax1 = paleofig.get_axis(0)
paleofig.ax1.text(14, -8.5, 'Termination I',
                  fontsize=14, weight='bold',
                  horizontalalignment='center')
```

The user provides the axes containing the lower left corner of the bar and the upper right corner of the bar. In the vertical bars of `paleofig` the vertical limits consist of the upper limit of the upper right axis and the lower limit of the lower left axis. The horizontal upper and lower limits are provided in data units, for example (11, 12.5). The default zorder is -1 in order to place the bar behind the curves, preventing data from being obscured.

As these bars typically span multiple axes, they must be drawn in Figure space rather than on the axes. This presents two challenges. The first is converting data coordinates to figure coordinates. In the private function `_convert_coords()`, we transform data coordinates (`dc`) into axes coordinates, and then into figure coordinates:

```
ac = ax.transData.transform(dc)

fc = self.fig.transFigure.inverted().transform(ac)
```

The figure coordinates are then used to determine the width, height, and positioning of the Rectangle in figure space.

TrendVis strives to be as order-agnostic as possible. However, a patch drawn in Figure space is completely divorced from the data the patch is supposed to highlight. If axes limits are changed, or the vertical or horizontal spacing of the plot is adjusted, then the bar will no longer be in the correct position relative to the data.

As a solution, for each bar drawn with TrendVis, the upper and lower horizontal and vertical limits, the upper right and lower left axes, and the index of the patch in XGrid.fig.patches are all stored as XGrid attributes. Storing the patch index allows the user to make other types of patches that are exempt from TrendVis' patch repositioning. When any of TrendVis' wrappers around matplotlib's subplot spacing adjustment, x or y limit settings, etc are used, the user can stipulate that the bars automatically be adjusted to new figure coordinates. The stored data coordinates and axes are converted to figure space, and the x, y, width, and height of the existing bars are adjusted. Alternatively, the user can make changes to axes space relative to figure space without adjusting the bar positioning and dimensions each time or without using TrendVis wrappers, and simply adjust the bars at the end.

TrendVis also enables a special kind of bar, a frame. The frame is designed to visually anchor data axis spines, and appears around an entire column (row in `YGrid`) of data axes under the spines. However, for `paleofig` we will use a softer division of our the columns by using cut marks on the main axes to signify a broken axis:

```
paleofig.draw_cutout(di=0.075)
```

Similar to bars, frames are drawn in figure space and can sometimes be moved out of place when axes positions are changed relative to figure space, thus they are handled in the same way. Cutouts, however, are actual line plots on the axes that live in axes space and will not be affected by adjustments in axes limits or subplot positioning. With the cut marks drawn on `paleofig`, we have completed the dense but highly readable plot shown in Figure 1.

Conclusions and Moving Forward

TrendVis is a package that expedites the process of creating complex figures with multiple x or y axes against a common y or x axis. It is largely order-agnostic and exposes most of its attributes and methods in order to promote highly-customizable and reproducible plot creation in this particular style. In the long-term, with the help of the scientific Python community, TrendVis aims to become a widely-used higher level tool for the matplotlib plotting library and alternative to expensive software such as SigmaPlot and MATLAB, and to time-consuming, error-prone practices like assembling multiple Excel plots in vector graphics editing software.

REFERENCES

[Petit] J. R. Petit et al. *Climate and Atmospheric History of the Past 420,000 years from the Vostok Ice Core, Antarctica* Nature, 399:429-436, 1999.

[BarMatthews] M. Bar-Matthews et al. *Sea--land oxygen isotopic relationships from planktonic foraminifera and speleothems in the Eastern Mediterranean region and their implication for paleorainfall during interglacial intervals*, Geochimica et Cosmochimica Acta, 67(17):3181-3199, 2003.

[Drysdale] R. N. Drysdale et al. *Stalagmite evidence for the onset of the Last Interglacial in southern Europe at 129 pm1 ka*, Geophysical Research Letters, 32(24), 2005.

[Wang] Y. J. Wang et al. *A high-resolution absolute-dated late Pleistocene monsoon record from Hulu Cave, China*, Science, 294(5550):2345-2348, 2001.

[Dykoski] C. A. Dykoski et al., *A high-resolution, absolute-dated Holocene and deglacial Asian monsoon record from Dongge Cave, China*, Earth and Planetary Science Letters, 233(1):71-86, 2005.

[Sanbao] Y. J. Wang et al. *Millennial-and orbital-scale changes in the East Asian monsoon over the past 224,000 years*, Nature, 451(7182):1090-1093, 2008.

[Wagner] J. D. M. Wagner et al. *Moisture variability in the southwestern United States linked to abrupt glacial climate change*, Nature Geoscience, 3:110-113, 2010.

[Asmerom] Y. Asmerom et al. *Variable winter moisture in the southwestern United States linked to rapid glacial climate shifts*, Nature Geoscience, 3:114-117, 2010.

[Winograd0] I. J. Winograd et al. *Continuous 500,000-year climate record from vein calcite in Devils Hole, Nevada*, Science, 258(5080):255-260, 1992.

[Winograd1] I. J. Winograd et al. *Devils Hole, Nevada, $delta$ 18 O record extended to the mid-Holocene*, Quaternary Research, 66(2):202-212, 2006.

[Lachniet] M. S. Lachniet et al. *Orbital control of western North America atmospheric circulation and climate over two glacial cycles*, Nature Communications, 5, 2014.

[Shakun] J. D. Shakun et al. *Milankovitch-paced Termination II in a Nevada speleothem?* Geophysical Research Letters, 38(18), 2011.

[matplotlib] J. D. Hunter. *Matplotlib: A 2D Graphics Environment*, Computing in Science & Engineering, 9:90-95, 2007.

[Cross] M. Cross et al. *Great Basin hydrology, paleoclimate, and connections with the North Atlantic: A speleothem stable isotope and trace element record from Lehman Caves, NV*, Quaternary Science Reviews, in press.

[NumPy] S. van der Walt et al. *The NumPy Array: A Structure for Efficient Numerical Computation*, Computing in Science & Engineering, 13:22-30, 2011.

Causal Bayesian NetworkX

Michael D. Pacer[‡*]

https://www.youtube.com/watch?v=qWAQgWOD_nA

Abstract—Probabilistic graphical models are useful tools for modeling systems governed by probabilistic structure. *Bayesian networks* are one class of probabilistic graphical model that have proven useful for characterizing both formal systems and for reasoning with those systems. Probabilistic dependencies in Bayesian networks are graphically expressed in terms of directed links from parents to their children. *Casual Bayesian networks* are a generalization of Bayesian networks that allow one to "intervene" and perform "graph surgery" — cutting nodes off from their parents. *Causal theories* are a formal framework for generating causal Bayesian networks.

This report provides a brief introduction to the formal tools needed to comprehend Bayesian networks, including probability theory and graph theory. Then, it describes Bayesian networks and causal Bayesian networks. It introduces some of the most basic functionality of the extensive NetworkX python package for working with complex graphs and networks [HSS08]. I introduce some utilities I have build on top of NetworkX including conditional graph enumeration and sampling from discrete valued Bayesian networks encoded in NetworkX graphs [Pac15]. I call this Causal Bayesian NetworkX, or CBNX. I conclude by introducing a formal framework for generating causal Bayesian networks called theory based causal induction [GT09], out of which these utilities emerged. I discuss the background motivations for frameworks of this sort, their use in computational cognitive science, and the use of computational cognitive science for the machine learning community at large.

Index Terms—probabilistic graphical models, causal theories, Bayesian networks, computational cognitive science, networkx

Introduction and Aims

My first goal in this paper is to provide enough of an introduction to some formal/mathematical tools such that those familiar with `python` and programming more generally will be able to appreciate both why and how one might implement causal Bayesian networks. Especially to exhibit *how*, I have developed parts of a toolkit that allows the creation of these models on top of the `NetworkX` python package:cite:*networkx*. Given the coincidence of the names, it seemed most apt to refer to this toolkit as `Causal Bayesian NetworkX` abbreviated as CBNX[1].

In order to understand the tool-set requires the basics of probabilistic graphical models, which requires understanding some graph theory and some probability theory. The first few pages are devoted to providing necessary background and illustrative cases for conveying that understanding.

Notably, contrary to how Bayesian networks are commonly introduced, I say relatively little about inference from observed data. This is intentional, as is this discussion of it. Many of the most trenchant problems with Bayesian networks are found in critiques of their use to infer these networks from observed data. But, many of the aspects of Bayesian networks (especially causal Bayesian networks) that are most useful for thinking about problems of structure and probabilistic relations do not rely on inference from observed data. In fact, I think the immediate focus on inference has greatly hampered widespread understanding of the power and representative capacity of this class of models. Equally – if not more – importantly, I aim to discuss generalizations of Bayesian networks such as those that appear in [GT09], and inference in these cases requires a much longer treatment (if a comprehensive treatment can be provided at all). If you are dissatisfied with this approach and wish to read a more conventional introduction to (causal) Bayesian networks I suggest consulting [Pea00].

The current instantiation of the CBNX toolkit can be seen as consisting of two main parts: graph enumeration/filtering and the storage and use of probabilistic graphical models in a NetworkX compatible format [HSS08].

I focus first on establishing a means of building iterators over sets of directed graphs. I then apply operations to those sets. Beginning with the complete directed graph, we enumerate over the subgraphs of that complete graph and enforce graph theoretic conditions such as acyclicity over the entire graph, guarantees on paths between nodes that are known to be able to communicate with one another, or orphan-hood for individual nodes known to have no parents. We accomplish this by using closures that take graphs as their input along with any explicitly defined arguments needed to define the exact desired conditions.

I then shift focus to a case where there is a specific known directed acyclic graph that is imbued with a simple probabilistic semantics over its nodes and edges, also known as a Bayesian network. I demonstrate how to sample independent trials from these variables in a way consistent with these semantics. I discuss some of the challenges of encoding these semantics in dictionaries as afforded by NetworkX without resorting to `eval` statements.

I conclude by discussing Computational Cognitive Science as it relates to graphical models and machine learning in general. In particular, I will discuss a framework called **theory based causal induction** [GT09], or my preferred term: **causal theories**, which allows for defining problems of causal induction. The perspective expressed in this paper, the associated talk, and the CBNX toolkit developed out of this framework.

* *Corresponding author: mpacer@berkeley.edu*
‡ *University of California at Berkeley*

1. Static code can be found at [Pac15], and the most recent version can be found at CBNX. CBNX is licensed with the BSD 3-clause license.

Graphical Models

Graphs are defined by a set of nodes $(X, |X| = N)$ and a set of edges between those nodes $(E|e \in E \equiv e \in (X \times X))$.

Notes on notation

Nodes: In the examples in CBNX, nodes are given explicit labels individuating them such as $\{A,B,C,\ldots\}$ or {'rain','sprinkler','ground'}. Often, for the purposes of mathematical notation, it is better to index nodes with integers over a common variable label, e.g., using $\{X_1,X_2,\ldots\}$.[2]

Edges: Defined in this way, edges are all *directed* in the sense that an edge from X_1 to X_2 is not the same as the edge from X_2 to X_1, or $(X_1,X_2) \neq (X_2,X_1)$. An edge (X_1,X_2) will sometimes be written as $X_1 \to X_2$, and the relation may be described using language like "X_1 is the parent of X_2" or "X_2 is the child of X_1".

Directed paths: Paths are a useful way to understand sequences of edges and the structure of a graph. Informally, to say there is a path between X_i and X_j is to say that one can start at X_i and by traveling from parent to child along the edges leading out from the node that you are currently at, you can eventually reach X_j.

To define it recursively and more precisely, if the edge (X_i,X_j) is in the edge set or if the edges (X_i,X_k) and (X_k,X_j) are in the edge set there is a path from X_i to X_j. Otherwise, a graph has a path from node X_i to X_j if there is a subset of its set of edges such that the set contains edges (X_i,X_k) and (X_l,X_j) and there is a path from X_k to X_l.

Adjacency Matrix Perspective

For a fixed set of nodes X of size N, each graph is uniquely defined by its edge set, which can be seen as a binary $N \times N$ matrix, where each index (i,j) in the matrix is 1 if the graph contains an edge from $X_i \to X_j$, and 0 if it does not contain such an edge. We will refer to this matrix as $A(G)$.

This means that any values of 1 found on the diagonal of the adjacency matrix (i.e., where $X_i \to X_j, i = j$) indicate a self-loop on the respective node.

Undirected Graphs

We can still have a coherent view of *undirected* graphs, despite the fact that our primitive notion of an edge is that of a *directed* edge. If a graph is undirected, then if it has an edge from $X_i \to X_j$ then it has an edge from $X_j \to X_i$. Equivalently, this means that the adjacency matrix of the graph is symmetric, or $A(G) = A(G)^\top$. However from the viewpoint of the undirected graph, that means that it has only a single edge.

Directed Graphs

From the adjacency matrix perspective we've been considering, all graphs are technically directed, and undirected graphs are a special case where one (undirected) edge would be represented as two symmetric edges.

The number of directed graphs that can be obtained from a set of nodes of size n can be defined explicitly using the fact that they can be encoded as a unique $n \times n$ matrix:

$$R_n = 2^{n^2}$$

Directed Acyclic Graphs: A cycle in a directed graph can be understood as the existence of a path from a node to itself. This can be as simple as a self-loop (i.e., if there is an edge (X_i,X_i) for any node X_i).

Directed acyclic graphs(DAGs) are directed graphs that contain no cycles.

The number of DAGs that obtainable from a set of n noddes can be defined recursively as follows [MOR+04]:

$$R_n = \sum_{k=1}^{n} (-1)^{k+1} \binom{n}{k} 2^{k(n-k)} R_{n-k}$$

Note, because DAGs do not allow any cycles, this means that there can be no self-loops. As a result, every value on the diagonal of a DAG's adjacency matrix will be 0.

Probability Distributions: Conditional, Joint and Marginal

A random variable defined by a conditional probability distribution[3] has a distribution indexed by the realization of some other variable (which itself is often a random variable, especially in the context of Bayesian networks).

The probability mass function (pmf) for discrete random variable X with value x will be noted as $P(X = x)$. Often, when discussing the full set of potential values (and not just a single value), we leave out the $= x$ and just indicate $P(X)$.[4]

The conditional probability of X with value x given another variable Y with value y is $P(X = x\ |Y = y)$. Much like above, if we want to consider the probability of each possible event without specifying one, sometimes this will be written as $P(X|Y = y)$. If we are considering conditioning on any of the possible values of the known variable, we might use the notation $P(X|Y)$, but that is a slight abuse of the notation.

You *can* view $P(X|Y)$ as a function over the $X \times Y$ space. But do not interpret that as a probability function. Rather, this defines a probability function for X relative to each value of Y. Without conditioning on Y we have many potential probability functions for X. Equivalently, it denotes a *family* of probability functions on X indexed by the values $Y = y$.

The *joint probability* of X and Y is the probability that both X and Y occur in the event set in question. This is noted as $P(X,Y)$ or

2. Despite pythonic counting beginning with 0, I chose not to begin this series with 0 because when dealing with variables that might be used in statistical regressions, the 0 subscript will have a specific meaning that separates it from the rest of the notation. For example when expressing multivariate regression as $Y = \beta X + \varepsilon, \varepsilon \sim \mathcal{N}(0,\Sigma)$, β_0 refers to the parameter associated with a constant variable $x_0 = 1$ and X is normally defined as $x_1,x_2,x_3,\ldots$. This allows a simple additive constant to be estimated, which often is not of interest to statistical tests, acting as a scaling constant. This makes for a simpler notation than $Y = \beta_0 + \beta X + \varepsilon$, because that is equivalent to $Y = \beta X + \varepsilon$ if $x_0 = 1$. But, in other cases (e.g., [PG12]) 0 index will be used to indicate background sources for events in a system.

3. Rather than choose a particular interpretation of probability over event sets (e.g., Bayesian or frequentist), I will attempt to remain neutral, as those concerns are not central to the issues of graphs and simple sampling.

4. If one is dealing with continuous quantities rather than discrete quantities one will have to use a probability density function (pdf) which does not have as straightforward an interpretation as a probability mass function. This difficult stems from the fact that (under most cases) the probability of any particular event occurring is "measure zero", or "almost surely" impossible. Without getting into measure theory and the foundation of calculus and continuity we can simply note that it is not that any individual event has non-zero probability, but that sets of events have non-zero probability.As a result, continuous random variables are more easily understood in terms a cumulative density function (cdf), which states not how likely any individual event is, but how likely it is that the event in question is less than a value x. The notation usually given for a cdf of this sort is $F(X \leq x) = \int_{-\infty}^{x} f(u)du$, where $f(u)$ is the associated probability density function.

$P(X \cap Y)$ (using the set theoretic intersection operation). Similar to $P(X|Y)$, you *can* view $P(X,Y)$ as a function over the space defined by $X \times Y$. However, $P(X,Y)$ is a probability function in the sense that the sum of $P(X = x, Y = y)$ over all the possible events in the space defined by $(x,y) \in X \times Y$ equals 1.

The *marginal probability* of X is just $P(X)$. The term "marginalization" refers to the notion of summing over values of Y in their joint probability. When probabilities were recorded in probability tables, the sum would be recorded in the *margins*. Formally, this can be stated as $P(X) = \sum_{y \in Y} P(X,Y)$.

Relating conditional and joint probabilities

Conditional probabilities are related to joint probabilities using the following form:

$$P(X|Y = y) = \frac{P(X, Y = y)}{P(Y = y)} = \frac{P(X, Y = y)}{\sum_{x \in X} P(X = x, Y = y)}$$

Equivalently:

$$P(X, Y = y) = P(X|Y = y)P(X)$$

Bayes' Theorem

Bayes' Theorem can be seen as a result of how to relate conditional and joint probabilities. Or more importantly, how to compute the probability of a variable once you know something about some other variable.

Namely, if we want to know $P(X|Y)$ we can transform it into $\frac{P(X,Y)}{\sum_{x \in X} P(X=x,Y)}$, but then can also transform joint probabilities ($P(X,Y)$) into statements about conditional and marginal probabilities ($P(X|Y)P(X)$). This leaves us with

$$P(X|Y) = \frac{P(Y|X)P(X)}{\sum_{x \in X} P(Y|X = x)P(X = x)}$$

Probabilistic Independence

To say that two variables are independent of each other means that knowing/conditioning on the realization of one variable is irrelevant to the distribution of the other variable. This is equivalent to saying that the joint probability is equal to the multiplication of the probabilities of the two events.

If two variables are conditionally independent, that means that conditional on some set of variables, condition

Example: Marginal Independence $\neq$ Conditional Independence

Consider the following example:

$$\begin{aligned} X &\sim \text{Bernoulli}_{\{0,1\}}(.5),\ Y \sim \text{Bernoulli}_{\{0,1\}}(.5) \\ Z &= X \oplus Y, \oplus \equiv \text{XOR} \end{aligned}$$

Note that, $X \perp\!\!\!\perp Y$ but $X \not\perp\!\!\!\perp Y|Z$.

Bayesian Networks

Bayesian networks are a class of graphical models that have particular probabilistic semantics attached to their nodes and edges. This makes them probabilistic graphical models.

In Bayesian networks when a variable is conditioned on the total set of its parents and children, it is conditionally independent of any other variables in the graph. This is known as the "Markov blanket" of that node.[5]

5. The word "Markov" refers to Andrei Markov and appears as a prefix to many other terms. It most often indicates that some kind of independence property holds. For example, a Markov chain is a sequence (chain) of variables in which each variable depends only on the value of the immediately preceding and postceding variables in the chain. Properties like this make computation easier.

Common assumptions in Bayesian networks

While there are extensions to these models, a number of assumptions commonly hold.

Fixed node set: The network is considered to be comprehensive in the sense that there is a fixed set of n known nodes. This rules out the possibility of hidden/latent variables as being part of the network. From this perspective inducing hidden nodes requires postulating a new graph that is potentially unrelated to the previous graph.

Trial-based events, complete activation and DAG-hood: Within a trial, all events are presumed to occur simultaneously.There is no notion of temporal asynchrony, where one node/variable takes on a value before its children take on a value (even if in reality – i.e., outside the model – that variable is known to occur before its child). Additionally, the probabilistic semantics will be defined over the entirety of the graph which means that one cannot sample a proper subset of the nodes of a graph without marginalizing out and incorporating information from the ignored nodes into the subset in question.

This property also explains why Bayesian networks need to be acyclic. Most of the time when we consider causal cycles in the world the cycle relies on a temporal delay between the causes and their effects to take place. If the cause and its effect is simultaneous, it becomes difficult (if not nonsensical) to determine which is the cause and which is the effect — they seem instead to be mutually definitional. But, as noted above, when sampling in Bayesian networks simultaneity is presumed for *all* of the nodes.

Independence in Bayes Nets

One of the standard ways of describing the relation between the semantics (probability values) and syntax (graphical structure) of Bayesian networks is how graph encodes particular conditional independence assumptions between the nodes of the graph. Indeed, in some cases Bayesian networks merely play the role of a convenient representation for conditional and marginal independence relationships between different variables.

It is the perspective of the graphs as *merely* representing the independence relationships and the focus on inference that leads to the focus on equivalence classes of Bayes nets. The set of graphs $\{A \rightarrow B \rightarrow C,\ A \leftarrow B \rightarrow C,$ and $A \leftarrow B \leftarrow C\}$ represent the same conditional independence relationships, and thus cannot be distinguished on the basis of observational evidence alone. This also leads to the emphasis on finding V-structures or common-cause structures where (at least) two arrows are directed into the same child with no direct link between those parents(e.g., $A \rightarrow B \leftarrow C$). V-structures are observationally distinguishable because any reversing the direction of any of the arrows will alter the conditional independence relations that are guaranteed by the graphical structure.[6]

Though accurate, this eschews important aspects of the semantics distinguishing arrows with different directions when you consider the kinds of values variables take on.

Directional semantics between different types of nodes: The conditional distributions of child nodes are usually defined with parameter functions that take as arguments their parents' realizations for that trial. Bayes nets often are used to exclusively

6. A more thorough analysis of this relation between graph structures and implied conditional independence relations invokes the discussion of *d-separation*. However, d-separation (despite claims that "[t]he intuition behind [it] is simple") is a more subtle concept than it at first appears as it involves both which nodes are observed and the underlying structure.

represent discrete (usually, binary) nodes the distribution is usually defined as an arbitrary probability distribution associated with the label of it's parent's realization.

If we allow (for example) positive continuous valued nodes to exist in relation to discrete nodes the kind of distributions available to describe relations between these nodes changes depending upon the direction of the arrow. A continuous node taking on positive real values mapping to an arbitrarily labeled binary node taking on values $\{a, b\}$ will require a function that maps from $\mathbb{R} \to [0, 1]$, where it maps to the probability that the child node takes on (for instance) the value a[7].However, if the relationship goes the other direction, one would need to have a function that maps from $\{a, b\} \to \mathbb{R}$. For example, this might be a Gaussian distributions for a and b $((\mu_a, \sigma_a), (\mu_b, \sigma_b))$. Regardless of the particular distributions, the key is that the functional form of the distributions are radically different.

Sampling and semantics in Bayes Nets

The procedure we will use to sample from Bayesian networks uses an *active sample set*. This is the set of nodes for which we have well-defined distributions at the time of sampling.

There will always be at least one node in a Bayesian network that has no parents. We will call these nodes *orphans*. To sample a trial from the Bayesian network we begin with the orphans. Because orphans have no parents – in order for the Bayes net to be well-defined – each orphan will have a well-defined probability distribution available for direct sampling. The set of orphans is our first active sample set.

After sampling from all of the orphans, we will take the union of the sets of children of the orphans, and at least one of these nodes will have values sampled for all of its parents. We take the set of orphans whose entire parent-set has sampled values, and sample from the conditional distributions defined relative to their parents' sampled values and make this the *active sample set*.

After sampling the active sample set, we will either have new variables whose distributions are well-defined or will have sampled all of the variables in the graph for that trial.

Example: Rain, Sprinkler & Ground

In the sprinkler Bayesian network in Figure 1[8], there three discrete nodes that represent whether it *Rains* (yes or no), whether the *Sprinkler* is on (on or off) and whether the *Ground* is wet (wet or dry). The edges encode the fact that the rain listens to no one, that the rain can alter the probability of whether the sprinkler is on, and the rain and the sprinkler together determine how likely it is that the ground is wet.

Causal Bayesian Networks

Causal Bayesian networks are Bayesian networks that are given an interventional operation allowing for "graph surgery" by cutting nodes off from their parents[9]. Interventions are cases where a causal force is able to exogenously set the values of individual nodes, rendering intervened on nodes independent of their parents.

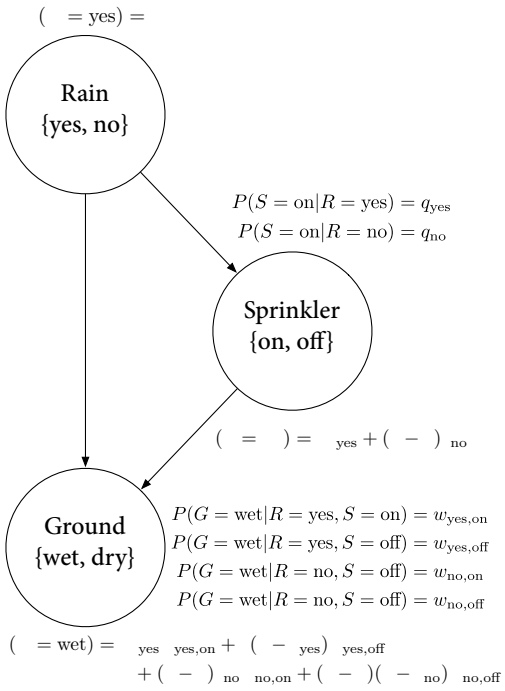


Fig. 1: *An Bayesian network describing the sprinkler example. Including both conditional and marginal distributions.*

NetworkX [HSS08]

NetworkX is a package for using and analyzing graphs and complex networks. It stores different kinds of graphs as variations on a "dict of dicts of dicts" structure. For example, directed graphs are stored as two dict-of-dicts-of-dicts structures[10].

Basic NetworkX operations

NetworkX is usually imported using the `nx` abbreviation, and you input nodes and edges as lists of tuples, which can be assigned dictionaries as their last argument, which stores the dictionary as the nodes' or edges' data.

```
import networkx as nx

G = nx.DiGraph() # init directed graph
G.add_edges_from(edge_list) # input edges
G.add_nodes_from(node_list) # input nodes
edge_list = G.edges(data=True) # output edges
node_list = G.nodes(data=True) # output nodes
```

7. If the function maps directly to one of the labeled binary values this can be represented as having probability 1 of mapping to either a or b.

8. This is an ill-specified Bayesian network, because while I have specified the states and their relations, I left open the potential interpretation of the parameters and how they relate to one another. I did so because it shows both the limits and strengths of what is encoded knowing only the structure, computing both conditional and marginal distributions for all variables.

9. This is technically a more general definition than that given in [Pea00] as in that case there is a specific semantic flavor given to interventions as they affect the probabilistic semantics of the variables within the network. This is related to his notion of a `do`-operator which deterministically sets a node to a particular value. Because here we are considering a version of intervention that affects the *structure* of a set of graphs rather than an intervention's results on a specific parameterized graph, this greater specificity is unnecessary.

10. It can also represent multi-graphs (graphs where multiple versions of "the same" edge from the adjacency matrix perspective can exist and will (usually) carry different semantics). We will not be using the multigraph feature of NetworkX, as multigraphs are not traditionally used in the context of Bayesian networks.

CBNX: Graphs

Here we will look at some of the basic operations described in the *ipython notebook* [JOP+] found at CBNX. For space and formatting reasons this code may differ slightly from that either in the variable names or comments, for the original version of these code snippets see graph-builder-code.

Other packages

In addition to networkX, we need to import numpy [VDWCV11], scipy [JOP+], and functions from itertools.

```
import numpy as np
import scipy
from itertools import chain, combinations, tee
```

Beginning with a max-graph

Starting with the max graph for a set of nodes (i.e., the graph with N^2 edges), we build an iterator that returns graphs by successively removing subsets of edges. Because we start with the max graph, this procedure will visit all possible subgraphs. One challenge that arises when visiting *all* possible subgraphs is the sheer magnitude of that search space (2^{N^2}).

```
def completeDiGraph(nodes):
    G = nx.DiGraph()
    G.add_nodes_from(nodes)
    edgelist = list(combinations(nodes,2))
    edgelist.extend([(y,x) for x,y in edgelist)
    edgelist.extend([(x,x) for x in nodes])
    G.add_edges_from(edgelist)
    return G
```

Preëmptive Filters

The graph explosion problem is helped by determining which individual edges are known to always be present and which ones are known to never be present. In this way we can reduce the size of the edgeset over which we will be iterating.

Filters can be applied by using `filter_Graph()`, which takes a graph and a filter_set as its arguments and returns a graph. A filter_set is a set of functions that take each take (at least) a graph as an argument and return a graph with a reduced edgeset according to the semantics of the filter.

```
def filter_Graph(G,filter_set):
    graph = G.copy()
    for f in filter_set:
        graph = f(graph)
    return graph
```

Example filter: remove self-loops

By default the graph completed by `completeDiGraph()` will have self-loops, often we will not want this (e.g., DAGs cannot contain self-loops).

```
def extract_remove_self_loops_filter():
    def remove_self_loops_filter(G):
        g2 = G.copy()
        g2.remove_edges_from(g2.selfloop_edges())
        return g2
    return remove_self_loops_filter
```

Conditions

The enumeration portion of this approach is defined in this `conditionalSubgraphs` function.[#]_ This allows you to pass in a graph from which you will want to sample subgraphs that meet the conditions that you also pass in.

```
def conditionalSubgraphs(G,condition_list):
    for edges in powerset(G.edges()):
        G_test = G.copy()
        G_test.remove_edges_from(edges)
        if all([c(G_test) for c in condition_list]):
            yield G_test
```

Example condition: requiring complete paths

This condition holds only if a graph has paths from the first node to the second node for each 2-tuple in the node-pair list.

```
def create_path_complete_condition(n_p):
    def path_complete_condition(G):
        return all([nx.has_path(G,x,y) for x,y in n_p])
    return path_complete_condition
```

Non-destructive conditional subgraph generators

Because `conditionalSubgraph` produces an iterator, applying a condition after that initial set is generated, requires splitting it into two copies of the iterator. This involves the `tee` function from the `itertools` core package.

```
def new_conditional_graph_set(graph_set,cond_list):
    graph_set_newer, graph_set_test = tee(graph_set,2)
    def gen():
        for G in graph_set_test:
            G_test = G.copy()
            if all([c(G_test) for c in condition_list]):
                yield G_test
    return graph_set_newer, gen()
```

Filters versus Conditions: which to use: The structural differences between filters and conditions highlight how they are to be used. Filters are intended to apply a graph to reduce its edge set in place; as such they return a graph. Conditions return truth values — they are applied to graph set reducing the size of that graph set.

CBNX: Representing probabilistic relations and sampling

We discuss an algorithm for sampling from Bayesian networks above (sampling). But, most of the difficult parts of encoding a sampling procedure prove (in this case) to do with the algorithm. Rather, the most pressing difficulties arise from attempting to store the relevant information within the NetworkX data dictionaries, so that a self-contained graphical object can be imported and exported. There is a general problem of a lack of standard storage format for Bayesian networks (and probabilistic graphical models in general). This is just one flavor of that problem.

A CBNX *implementation for sprinkler graph*

Below I will illustrate how to use NetworkX [HSS08] and node-associated attributes to define and sample from a parameterized version of the sprinkler Bayesian network represented in abstract, graphical form in Figure 1. for space reasons comments and formatting were reduced, if you wish to see the original code it can be found at sampling-code.

11. Note that powerset will need to be built (see CBNX for details).

Sampling infrastructure

```
def sample_from_graph(G,f_dict=None,k = 1):
    if f_dict == None:
        f_dict = {"choice": np.random.choice}
    n_dict = G.nodes(data = True)
    n_ids = np.array(G.nodes())
    n_states = [(n[0],n[1]["state_space"])
        for n in n_dict]
    orphans = [n for n in n_dict
        if n[1]["parents"]==[]]
    s_values = np.empty([len(n_states),k],dtype='U20')
    s_nodes = []
    for n in orphans:
        samp_f = str_to_f(n[1]["sample_function"],
            f_dict)
        s_states = n[1]["state_space"]
        s_dist = n[1]["dist"]
        s_idx = G.nodes().index(n[0])
        s_values[s_idx,:]  = samp_f(s_states,
            size=[1,k],p=s_dist)
        s_nodes.append(n[0])
    while set(s_nodes) < set(G.nodes()):
        nodes_to_sample = has_full_parents(G,s_nodes)
        for n in nodes_to_sample:
            par_indices = [(par,G.nodes().index(par))
                for par in G.node[n]["parents"]]
            par_vals = [(par[0],s_values[par[1],:])
                for par in par_indices]
            samp_index = G.nodes().index(n)
            s_values[samp_index,:] = cond_samp(G,n,
                par_vals,f_dict,k)
            s_nodes.append(n)
    return s_values

def has_full_parents(G,s_n):
    check_n = [x for x in G.nodes() if x not in s_n]
    nodes_to_be_sampled = []
    for n in G.nodes(data = True):
        if (n[0] in check_n) & (n[1]["parents"]<=s_n):
            nodes_to_be_sampled.append(n[0])
    if len(nodes_to_be_sampled)==0:
        raise RuntimeError("A node must be sampled")
    return nodes_to_be_sampled

def nodeset_query(G,n_set,n_atr=[]):
    if len(n_atr)==0:
        return [n for n in G.nodes(data = True)
            if n[0] in n_set]
    else:
        return_val = []
        for n in G.nodes(data=True):
            if n[0] in node_set:
                return_val.append((n[0],
                {attr:n[1][attr] for attr in n_atr}))
        return return_val

def cond_samp(G,n,par_vals,f_dict, k = 1):
    try: n in G
    except KeyError:
        print("{} is not in graph".format(n))
    output = np.empty(k,dtype="U20")
    for i in np.arange(k):
        val_list = []
        for p in par_vals:
            val_list.append(tuple([p[0],p[1][i]]))
        samp_dist = G.node[n]["dist"][tuple(val_list)]
        samp_f = str_to_f(
            G.node[n]["sample_function"],f_dict)
        samp_states = G.node[n]["state_space"]
        temp_output = samp_f(samp_states,
            size=1,p=samp_dist)
        output[i] = temp_output[0]
    return output

def str_to_f(f_name, f_dict=None):
    if f_dict == None:
        f_dict = {"choice": np.random.choice}
    try: f_dict[f_name]
    except KeyError:
        print("{} is not defined.".format(f_name))
    return f_dict[f_name]
```

Sampling from the sprinkler Bayes net with CBNX

The following encodes the sprinkler network from Figure 1 with parameters $p = .2, q_{\text{yes}} = .01, q_{\text{no}} = .4, w_{\text{yes,on}} = .99, w_{\text{yes,off}} = .8, w_{\text{no,on}} = .9$ and $w_{\text{no,off}=0}$. This distribution is meant to accord with our intuitions that rain and sprinklers increase the probability of the ground being wet, and that we are less likely to use the sprinkler when it has rained.

```
node_prop_list = [("rain",{
    "state_space":("yes","no"),
    "sample_function": "choice",
    "parents":[],
    "dist":[.2,.8]}),
    ("sprinkler",{
    "state_space":("on","off"),
    "sample_function": "choice",
    "parents":["rain"],
    "dist":{(("rain","yes"),):[.01,.99],
            (("rain","no"),):[.4,.6]}}),
    ("grass_wet",{
    "state_space":("wet","dry"),
    "sample_function": "choice",
    "parents":["rain","sprinkler"],
    "dist":{
        (("rain","yes"),("sprinkler","on")):[.99,.01],
        (("rain","yes"),("sprinkler","off")):[.8,.2],
        (("rain","no"),("sprinkler","on")):[.9,.1],
        (("rain","no"),("sprinkler","off")):[0,1]}})]


edge_list = [("sprinkler","grass_wet"),
    ("rain","sprinkler"),
    ("rain","grass_wet")]

G = nx.DiGraph()
G.clear()
G.add_edges_from(edge_list)
G.add_nodes_from(node_prop_list)
test = sample_from_graph(G,k=10)
```

Causal Theories and Computational Cognitive Science

Theory based causal induction is a formal framework arising out of the tradition in computational cognitive science to approach problems of human cognition with rational, computational-level analyses [GT09]. Causal theories form generative models for defining classes of parameterized probabilistic graphical models. They rely on defining a set of classes of entities (ontology), potential relationships between those classes of entities and particular entities (plausible relations), and particular parameterizations of how those relations manifest in observable data (or in how other relations eventually ground out into observable data). This allows Griffiths and Tenenbaum to subsume the prediction of a wide array of human causal inductive, learning and reasoning behavior using this framework for generating graphical models and doing inference over the structures they generate.

Rational analysis

Rational analysis is a technique that frees us from some of the problems inherent in mechanistic modeling in cognition. We specify the goals of the cognitive system, the environment in which it exists and minimal constraints on the computations available to the agent. We translate this into mathematically precise accounts

of "mechanism-free casting[s] of psychological [theories]" for optimal behavior. These formal models provide empirical predictions that can be evaluated by studying human cognitive behavior under different observable environmental conditions [And90][11]. If the model disagrees with the empirical data, we iterate — reëvaluating each component of the theory until we match a wide variety[12] of empirical data.

Computational-Level Analysis of Human Cognition

A computational-level analysis [Mar82] is one in which we model a system in terms of its functional role(s) and how they would be optimally solved. This is distinguished from algorithmic-level analysis by not caring how this goal achievement state is implemented in terms of the formal structure of the underlying system and from mechanistic-level analysis by not caring about the physical structure of how these systems are implemented (which may vary widely while still meeting the structure of the algorithmic-level which itself accomplishes the goals of the computational level).

A classic example [Mar82] of the three-levels of analysis are different ways of studying flying with the example of bird-flight. The mechanistic-level analysis would be to study feathers, cells and so on to understand the component subparts of individual birds. The algorithmic-level analysis would look at how these subparts fit together to form an active whole that is capable of flying often by flapping its wings in a particular way. The computational-level analysis would be a theory of aerodynamics with specific accounts for the way forces interact to produce flight through the particular motions of flying observed in the birds.

Causal theories: ontology, plausible relations, functional form

The causal theory framework generalizes specifying Bayesian network in the same way first-order logic generalizes specifying propositions in propositional logic. A causal theory requires elements necessary to populate nodes, those nodes with properties, and relations between the nodes, stating which of those relations are plausible (and how plausible), and a specific, precise formulation for how those relations manifest in terms of a probabilistic semantics. In the terms of [GT09]'s theory-based causal induction, this requires specifying an ontology, plausible relations over those ontologies, and functional forms for parameterizing those relations.

Ontology: This specifies the full space of potential kinds of entities, properties and relations that exist. This is the basis around which everything else will be defined. It is straightforward populate nodes with features using the data dictionary in NetworkX.

Plausible Relations: This specifies which of the total set of relations allowed by the ontology are plausible and how plausible. If you do not dramatically restrict the sets of relations you consider, there will be an explosion of possibilities. People, even young children, have many expectations about what sorts of things can can feasibly be causally related to one another. This sometimes has been interpreted as the plausible existence of a mechanism linking cause and effect. For example, we know that in most situations a fan is more likely than a tuning fork to blow out a candle.

Functional form:

> Even in the most basic cases of causal induction we draw on expectations as to whether the effects of one variable on another are positive or negative, whether multiple causes interact or are independent, and what type of events (binary, continuous, or rates) are relevant to evaluating causal relationships.
>
> —[GT09]

Of course, this allows for uncertainty about these functional forms and indeed, quite different judgments can be warranted depending on treats the underlying relation and structure of the data (e.g., continuous vs. binary data [PG11]).

Generalizations to other kinds of logical/graphical conditions

The causal theory framework is richer than the set of examples developed in [GT09]. It can express conditions of graphical connectivity, context-sensitive functional forms, substructures of constrained plausible relations, among others.

In [GT09], plausible relations are described in terms of sufficient conditions, implicitly suggesting that most relations are not plausible. However, we can also make necessary statements about the kinds of relations that *must* be there. And one can see this as selecting a subset of all the possible graphs implementable by the set of nodes defined by the ontology. It is for this purpose that I first arrived at the node enumeration.

One goal for CBNX is to enable causal theory programming. The utilities in `networkX`, plus the enumerating, filtering and conditioning functions in CBNX, ease implementing higher-order graphical conditions (e.g., a directed path necessarily existing between two nodes) than in the original notation described in [GT09]. These ideas were expressible in the original mathematical framework, but would have required a good deal more notational infrastructure to represent. CBNX not only provides a notation, but a programming infrastructure for expressing and using these kinds of conditions.

Uses in modeling human cognition

Using this framework, Griffiths and Tenenbaum were able to provide comprehensive coverage for a number of human psychology experiments. This allows them to model people's inferences in causal induction and learning regarding different functional forms, at different points in development, with different amounts of data, with and without interventions, and in continuous time and space (to name only a few of the different conditions covered).

They successfully modeled human behavior using this framework by treating people as optimal solvers of this computational problem[13] (at least as defined by their framework). Furthermore, by examining different but related experiments, they were able to demonstrate the different ways in which specific kinds of prior knowledge are called upon differentially to inform human causal induction resulting in quite different inferences on a rational statistical basis.

Cognition as Benchmark, Compass, and Map

People have always been able to make judgments that are beyond machine learning's state-of-the-art. In domains like object recognition, we are generally confident in people's judgments as

12. As Anderson notes, it is often the mathematization that proves to be the most difficult aspect of this procedure [And90].

13. Optimality in these cases is taken to mean on average approximating the posterior distribution of some inference problem defined by the authors in each case.

veridical, and – as such – they have been used as a benchmark against which to test and train machine learning systems. The eventual goal is that the system reaches a Turing point — the point at which machine performance and human performance are indistinguishable.

But that is not the only way human behavior can guide machine learning. In domains like causal induction, people's judgments cannot form a benchmark in the traditional sense because we cannot trust people to be "correct". Nonetheless, people *do* make these judgments and, more importantly, these judgments exhibit systematic patterns. This systematicity allows the judgments output by cognition to be modeled using formal, computational frameworks. Further, if we formally characterize both the inputs to *and* outputs from cognition, we can define judgments as optimal according to some model. Formal models of individual cognitive processes can then act as a compass for machine learning, providing a direction for how problems and some solutions can be computed.

Formal frameworks for generating models (e.g., causal theories) can be even more powerful. Data can often be interpreted in multiple ways, with each way requiring a model to generate solutions. Holding the data constant, different goals merit different kinds of solutions. Frameworks that generate models, optimality criteria and solutions not only provide a direction for machine learning, but lay out *sets* of possible directions. Generalized methods that use one system for solving many kinds of problems provide the ability to relate these different directions to each other. Formalizing the inputs, processes and outputs of human cognition produces a map of where machine learning could go, even if it never goes to any particular destination. From this, navigators with more details about the particular terrain can find newer and better routes.

Acknowledgements

Thank you to Jess Hamrick for aiding in the design of the underlying code, Katy Huff and Stéfan van der Walt for aiding in getting the bibliography working and helping me navigate github and the submission and review processes, Seb Benthall and Ankur Ankan for helping reviews, and Elizabeth Seiver for comments and support throughout the writing process.

REFERENCES

[And90] J. R. Anderson. *The adaptive character of thought*. Erlbaum, Hillsdale, NJ, 1990.

[GT09] T. L. Griffiths and J. B. Tenenbaum. Theory-based causal induction. *Psychological review*, 116(4), 2009.

[HSS08] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. Exploring network structure, dynamics, and function using NetworkX. In *Proceedings of the 7th Python in Science Conference (SciPy2008)*, pages 11–15, Pasadena, CA USA, August 2008.

[JOP+] Eric Jones, Travis Oliphant, Pearu Peterson, et al. SciPy: Open source scientific tools for Python. http://www.scipy.org/, 2001–. Online (accessed 2015-07-04).

[Mar82] D. Marr. *Vision*. W. H. Freeman, San Francisco, CA, 1982.

[MOR+04] Brendan D McKay, Frederique E Oggier, Gordon F Royle, NJA Sloane, Ian M Wanless, and Herbert S Wilf. Acyclic digraphs and eigenvalues of (0, 1)-matrices. *Journal of Integer Sequences*, 7:3, 2004.

[Pac15] M.D. Pacer. Causal-Bayesian-NetworkX. http://dx.doi.org/10.6084/m9.figshare.1471763, 2015. Online (accessed July 2, 2015). URL: http://dx.doi.org/10.6084/m9.figshare.1471763, doi:10.6084/m9.figshare.1471763.

[Pea00] J. Pearl. *Causality: Models, reasoning and inference*. Cambridge University Press, Cambridge, UK, 2000.

[PG11] M.D. Pacer and T.L. Griffiths. A rational model of causal induction with continuous causes. In *Advances in Neural Information Processing Systems*, volume 24, Cambridge, MA, 2011. MIT Press.

[PG12] M.D. Pacer and T.L. Griffiths. Elements of a rational framework for continuous-time causal induction. In *Proc. of the 34th Conf. of the CogSci Society*, 2012.

[VDWCV11] Stefan Van Der Walt, S Chris Colbert, and Gael Varoquaux. The numpy array: a structure for efficient numerical computation. *Computing in Science & Engineering*, 13(2):22–30, 2011.

Geodynamic simulations in HPC with Python

Nicola Creati‡*, Roberto Vidmar‡, Paolo Sterzai‡

https://www.youtube.com/watch?v=PTEgs7salEc

Abstract—The deformation of the Earth surface reflects the action of several forces that act inside the planet. To understand how the Earth surface evolves complex models must be built to reconcile observations with theoretical numerical simulations. Starting from a well known numerical methodology already used among the geodynamic scientific community, PyGmod has been developed from scratch in the last year. The application simulates 2D large scale geodynamic processes by solving the conservation equations of mass, momentum, and energy by a finite difference method with a marker-in-cell technique. Unlike common simulation code written in Fortran or C this code is written in Python. The code implements a new approach that takes advantage of the hybrid architecture of the latest HPC machines. In PyGmod the standard MPI is coupled with a threading architecture to speed up some critical computations. Since the OpenMP API cannot be used with Python, threading is implemented in Cython. In addition a realtime visualization library has been developed to inspect the evolution of the model during the computation.

Index Terms—HPC, numerical modelling, geodynamics

Introduction

The dynamics of surface and deep Earth processes, referred as geodynamics, is a challenging subject in geosciences since the establishment of plate tectonics in the late 1960s. The outer shell of the Earth is split into a number of thin, rigid plates that are in relative motion with respect to one another [Mor68]. Most of the earthquakes, volcanic eruptions and mountain buildings occur at plate boundaries [Tur02]. Geology started to move from a descriptive approach to a quantitative one [McK69], [Min70]. Numerical modeling in geodynamics is necessary because tectonic processes are too slow and too deep in the Earth to be observed directly. In the last 30 years numerical geodynamic modeling has developed very rapidly thanks to the introduction of new numerical techniques and the availability of powerful computers [Ger10]. Several of the known problems in computational science, such as the non-linear nature of rock rheology, the multicomponent nature of the systems and other thermodynamic variables, can now be managed. In the past years computer clusters of different architectures (shared memory systems, distributed memory systems and distributed shared memory systems) have become available to most researchers. To take full advantage of the power of these machines, parallel algorithms and software packages must be developed. However, geoscience researchers often do not have enough knowledge to build and debug parallel software. While in the last fifteen years several numerical methods and libraries have been developed to solve many of the equations needed to model geodynamic problems, almost all of them are written in C/C++ or Fortran. Sometimes geodynamic modeling is done with commercial software [Pas02], [Jar11] to reduce the effort of solving equations, as well as writing and debugging a new application. By exploiting several years of Python experience with remote sensing and geoscience topics, a numerical geodynamic modeling application, PyGmod, has been developed in the last twelve months. The development of PyGmod occurred within the PRACE-OGS research project, which is concerned with HPC applications for oceanographic and geophysical numerical simulations.

* Corresponding author: ncreati@inogs.it
‡ Istituto Nazionale di Oceanografia e di Geofisica Sperimentale, OGS

Scientific Context

PyGmod can simulate different geodynamic scenarios (e.g. plate subduction/collision, magma intrusion, continental drifting, etc. [Tur02]) and processes. The main target of PyGmod is the study of extensional geodynamic contexts. The application has been developed as a tool for understanding the genesis and evolution of extensional continental zones (rifts). Rifts and their final product, passive margins, are the expression of fundamental processes continually shaping planetary surfaces [Tur02]. They are sites of magmatic fluid and volatile transfer from the mantle to the surface trough flood basalt and alkaline magmatism, and from the surface to the mantle via surface weathering, hydrothermal systems and serpentinization. Sedimentary sequences contained within the segmented rift systems record the interplay between tectonics and climate throughout basin evolution, and they may sequester large volumes of CO2 and hydrocarbons. Like subduction margins, rifts may be sites of voluminous and explosive volcanism. Passive margins are sites of enormous landslides and destructive earthquakes [Jac97], [Buc04]. The poor understanding of rift initiation is partly due to the fact that extensive stretching, syn- and post-rift magmatism, and post breakup sedimentation usually overprint and bury the record of incipient extension at mature rifts and rifted margins. Understanding how, why and when rifts initiate and localize is important for defining factors controlling their dynamics. The relative importance of these factors during the inception and earliest development of a new rift is controversial.

Core Development

PyGmod is inspired by some examples available in the geodynamic literature [Ger10], [Deu08]. These codes have been studied, ported to Python and tested. The first Python version was serial,

was based on the Numpy [Van11] and Scipy [Jon01] packages to manage arrays and solved the governing differential equations. The parallel version of the algorithm has been developed on a multicore commodity PC with the target of the distributed shared memory architecture systems available at CINECA [Cin]. The porting of PyGmod from the commodity PC to a supercomputer was not straightforward beacuse every supercomputer has a different hardware architecture. The MPI message-passing system has been adopted while the Python multiprocessing module was avoided because it is not available on all supercomputers (e.g. the IBM Blue Gene series [Gil13]). The management of parallel distributed arrays has been possible thanks to the Global Array Toolkit (GA) library [Nie06] which is available in Python as ga4py. GA hides all the complexity of managing the distributed arrays between the nodes making them available simply as Numpy arrays. The parallel solution of the governing equations is done using the well known PETSc library provided by the petsc4py package [Dal11]. Simulation results are stored in an HDF5 file using the h5py [Col13] package. Ga4py, petsc4py and h5py are all depending on mpi4py [Dal05], [Dal11] and provide a higher level interface that greatly reduces the parallel programming effort. Practically all the MPI communication in PyGmod is hidden by the previous three packages and there are only few direct calls to MPI methods (Figure 1).

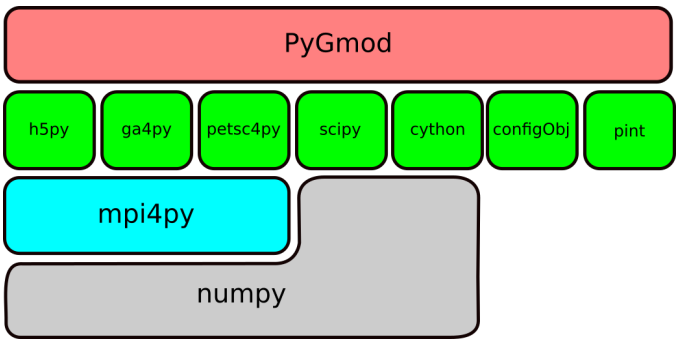


Fig. 1: *PyGmod layer diagram.*

PyGmod Structure

PyGmod is a 2D thermomechanical code based on an well known geodynamic modeling methodology characterized by the solution of conservative finite difference schemes [Pat80] coupled with the marker-in-cell method [Ger03], [Ger07], [Deu08]. The deformation of rocks is modeled as flow of variable viscous material as in computational fluid dynamics [Ver07]. The governing equations reflect the conservation laws of physics:

- conservation of mass,
- conservation of momentum (rate of change of momentum is equal to the sum of forces on the fluid particle, second law of Newton),
- conservation of energy (first law of thermodynamics).

These equations are coupled with rock rheological laws that take in account stress, strain-rate, viscosity, temperature, pressure and composition. The solution is achieved by a finite difference conservative schema and coupled with the the moving-marker Lagrangian approach [Bra86]. The equations are solved on an Eulerian grid while the markers are a cloud of points which covers the grid. The marker-in-cell methodology is characterized by several interpolation processes from the markers to the nodes of the grid and back [Ger03], [Deu08]. These are atomic calculations whose execution time depends on the number of markers and the type of interpolator (linear, bilinear, cubic, etc.). Implementation of this algorithm is usually done on shared memory architecture computers using the OpenMP API [Gor06]. These interpolations unfortunately cannot be vectorized by Numpy as they need the allocation of large temporary arrays for every MPI process and the memory available can be very little even if the number of processors is huge. For example, the IBM BG/Q at CINECA has only 1 Gb of RAM available to each MPI process even if the system has 160K cores [Gil13].

GA greatly helps to create, distribute and manage all the arrays, both 1D and 2D, providing a shared memory style programming environment in the context of a distributed array data structures. GA arrays are global and can be used as if they were stored in a shared memory environment. All details of the data distribution, addressing, and data access are encapsulated in the global array objects. The basic shared memory operations supported include get, put, scatter, and gather. These operations are truly one-sided/unilateral and will complete regardless of any action taken by the remote process(es) which own(s) the referenced data.

PyGmod uses a modified GA version which implements the ARMCI-MPI [Armci] RMA (Remote Memory Access) one-sided communication because the standard GA implementation, available at the time of the development of PyGmod, worked only on few hardware architectures.

The MPI topology implemented by a global array is used to split the 2D domain in Cartesian blocks along the vertical and horizontal axes and to assign to each block the markers which belong to it. Each block of data is then extended to partially overlap its neighbors to avoid border effects. Markers move inside the model domain at every time step iteration and the local portion of markers inside each Cartesian block must be extracted again. In each time iteration, most of the calculation is done on the local portion of the markers and on the grid nodes using only Numpy arrays and methods. Numerical calculation on local arrays has been vectorized by Numpy methods wherever possible. The following is an example of a block of code that has been vectorized to speed up computation (up to 75x) by removing a double for-loop:

```
# Original code
(r0, c0), (r1, c1) = ga.distribution(self.dexy)
for i in range(dexy.shape[0]):
    for j in range(dexy.shape[1]):
        dexy[i, j] = (
            0.5 * ((vx[i + 1, j] - vx[i, j]) /
            self.dyc[i + r0] +
            (vy[i, j + 1] - vy[i, j]) /
            self.dxc[j + c0]))

# Vectorized code
i = np.arange(dexy.shape[0])
j = np.arange(dexy.shape[1])
dexy[:] = (
    0.5 * ((vx[i + 1, :][:, j] -
    vx[i, :][:, j]) / self.dyc[i + r0, np.newaxis] +
    (vy[i, :][:, j + 1] - vy[i, :][:, j]) /
    self.dxc[j + c0]))
```

The governing equations are solved using the PETSc library provided by petsc4py. PyGmod uses direct equation solvers to achieve accurate solutions like MUMPS [Ame00] or Superlu [Li03] beacuse the problem is 2D and current supercomputers

```
[20150609-123405.046][0]: ---> Now starting step 6 - Time elapsed: 60.0k years <---
[20150609-123405.071][2]: Markers distribution done
[20150609-123405.073][0]: Markers distribution done
[20150609-123405.073][3]: Markers distribution done
[20150609-123405.074][1]: Markers distribution done
[20150609-123405.109][2]: Markers toNode done
[20150609-123405.109][1]: Markers toNode done
[20150609-123405.121][0]: Markers toNode done
[20150609-123405.121][3]: Markers toNode done
[20150609-123405.122][2]: Temperature BC update done
[20150609-123405.122][1]: Temperature BC update done
[20150609-123405.122][0]: Temperature BC update done
[20150609-123405.122][3]: Temperature BC update done
[20150609-123405.123][1]: Viscosity and stress done
[20150609-123405.123][2]: Viscosity and stress done
[20150609-123405.123][0]: Viscosity and stress done
[20150609-123405.123][3]: Viscosity and stress done
[20150609-123405.164][0]: x and y Stokes and Continuity right part done
[20150609-123405.164][2]: x and y Stokes and Continuity right part done
[20150609-123405.165][1]: x and y Stokes and Continuity right part done
[20150609-123405.165][3]: x and y Stokes and Continuity right part done
```

Fig. 2: *Example of on-screen log outputby four MPI tasks running with log level "info".*

provide enough memory. Unfortunately GA arrays cannot be directly passed to PETSc solvers so local processor ranges of PETSc sparse arrays and vectors must be extracted and the corresponding data block must be retrieved as Numpy arrays from the global array instance. The following is an example of the extraction of the local portion of a quantity from a global array needed later to fill PETSc arrays:

```
# Get local PETSC ranges
istart, iend = l.getOwnershipRange()

# Calculate equivalent local block of
# GA array ranges
c0 = istart / dofs / (ynum - 1)
c1 = iend / dofs / (ynum - 1) + 2
r0 = 0
r1 = ynum - 1

# From global GA array get needed block as
# Numpy array
local_array = ga.get(
    global_array, (r0, c0), (r1, c1))
```

In this example, *l* is a PETSc distributed bi-dimensional array, *dofs* is the degree of freedom of the system, and *ynum* is the total number of rows of the model. The local array is used to fill the local PETSc portion of the sparse arrays and vectors. The local solution of the equations, a Numpy array, is then put back in the corresponding global array.

PyGmod communicates with the user by a logging system, based on MPI, that helps the tracking of each step of the simulation and is of paramount importance in debugging the code. The *MPILog* class uses an *MPI.File* object's methods to write logging information to standard output (Figure 2) or to a file. Six log levels are defined: *critical*, *error*, *warning*, *notice*, *info* and *debug*, with warning as the default. Each MPI process writes its log to the same file in a different color. Log level as well as processor ID number can be filtered out. Log calls are invoked according to the following syntax:

```
log.info(...)
log.error(...)
log.critical(...)
```

Each of these is a pythonic shortcut to the write method of the *MPILog* object:

```
def write(self, inmsg, watch=['all'],
    rank=True, mono=False, level=INFO):
```

In this example *inmsg* is the message string, *watch* is the list of processors to which the message applies, *rank* is a switch to hide the processor rank from the message, *mono* disables colorized messages, and *level* defines the minimum level at which the message will be printed.

Each simulation is controlled by a single configuration file handled by the ConfigObj [Cfg] package. This file provides some general physical constants, modeling switches, PETSc equation solver options, mesh geometry and size, lithological geometry, initial distribution of temperature, boundary conditions, and topography. Units of measurements can be included in the configuration file because the parsing system implemented converts the units to the right ones needed by PyGmod checking also for dimensionality consistency. This has been accomplished adopting the Pint [Pint] package. The configuration file is organized in several sections as in the following condensed example:

```
# Physical constants
gx  =  0.  m / s**2
gy  =  9.81 m / s**2

# Ouput file
output_file = 'extension.hdf5'
log_file = 'extension.log'

# Stokes solver options
stokesSolver = """
    ksp_type=preonly
    pc_type=lu
    pc_factor_mat_solver_package=superlu_dist
    mat_superlu_dist_colperm=PARMETIS
    mat_superlu_dist_parsymbfact=1
    """
...
# Specific sections
[Mesh]
    model = "extension"
    SizeAlongX = 400000
    SizeAlongY = 300000
    NumberOfNodesAlongX = 161
    NumberOfNodesAlongY = 61
    NumberOfMarkersAlongX = 500
    NumberOfMarkersAlongY = 400
    DistributionOfNodesAlongX = """(
        'Variable(0.0, 100000.0, 2000.0, 30,
                  rtol=True)',
        'Constant(100000.0, 300000.0, 100)',
        'Variable(300000.0,    400000, 2000.0,
                  30, rtol=False)'
```

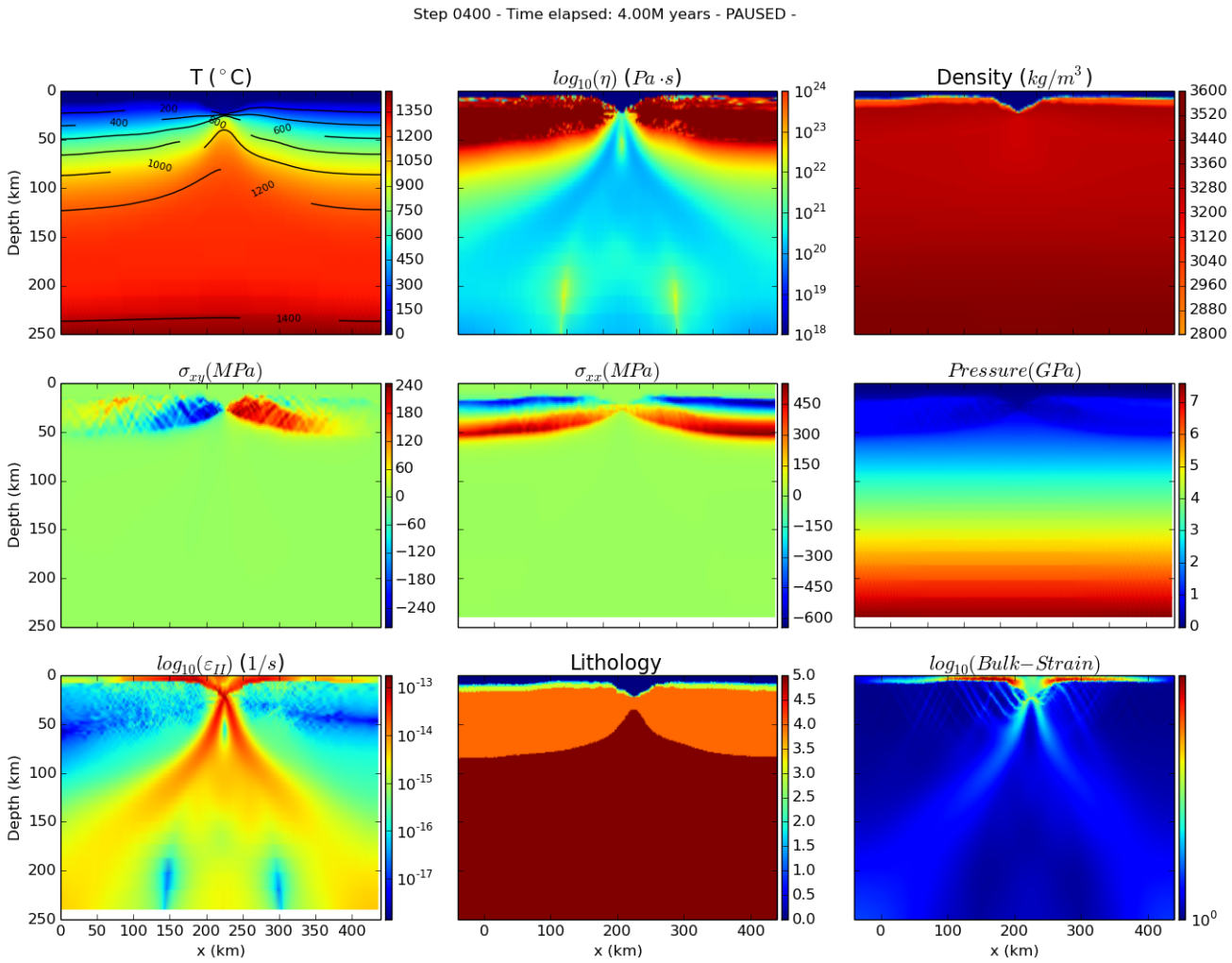


Fig. 3: *RTV screenshot of a rift simulation.*

```
        )"""
    DistributionOfNodesAlongY = """(
        'Constant(0.0, 80000.0, 40)',
        'Variable(80000.0, 300000, 2000.0,
                  20, rtol=False)'
        )"""


# Lithological/Rheological model
[Lithologies]

    [["Lithospheric mantle"]]
      density = 3300 * kg/m**3
      melt_density = 2700 * kg/m**3
      sinFI0 = 0.6 * dimensionless
      sinFI1 = 0.0 * dimensionless
      GAM1 = 0.1 * dimensionless

# Geometry of polygons where
# lithologies are defined
[Polygons]
    lid = """
            0 32
            0 95
           48 95
           48 32
           """
[Thermal Boundary Condition]
...
[Fluid Boundary Condition]
...
[Topography]
...
```

Modeling results are stored in HDF5 files created by the parallel (MPI) version of the h5py package. Each time iteration is saved in a different HDF5 file (evolution step) to avoid large files. A main output file also contains a copy of the configuration which generated the simulation for the entire evolution.

Results of the simulation can be explored by a viewer application module called Real Time Viewer (RTV). RTV code is based on Matplotlib [Hun07] and plots some of the quantities calculated in the simulation (Figure 3). Because the visualization of over a million markers as a cloud of points can be challenging, data are interpolated during the simulation using the power of MPI and saved in the HDF5 file as arrays. Thus, each processor interpolates only a small image patch from its own local markers pool. The interpolation uses the *griddata* module of Scipy with a nearest neighbors switch. RTV can plot data from a real-time simulation showing the current evolution step or historical data.

Each simulation can be interrupted by the user or by the operating system and restarted from the last completed time iteration without any data loss.

Performance

PyGmod was built using optimized third party libraries to speed up the computation and avoid the direct calls to MPI primitives needed for the parallelization wherever possible. Some sections (e.g. the mesh and topography objects) and some arrays are not yet parallel. These objects and arrays are replicated on all tasks since the size of the problems used to develop the code was not so big so as to require further optimization. Further parallelization should increase the speed and decrease the memory allocation. Tests proved that marker interpolation is a critical operation that can take a large amount of time. Interpolation is done in for-loops as the atomic nature of the algorithm used forbids the use of Numpy methods. Marker points contribute to the resolution of the model and they tend to be on the order of millions dramatically slowing down the computation. The following code is an example of one of the interpolations in PyGmod:

```
# Loop over markers
for mk in range(len(idx)):

    # Check if data is in the model domain
    if self.inDomain(...):

        # Find upper left node of the grid
        # from marker coordinates
        xn, yn, dx, dy = self.ul_node(...)

        # Linear interpolation method
        self.markerBint(...)
```

The loop operates over all the markers inside the block assigned to each processor and every time iteration step calls the interpolation methods several times. Because Python loops are inherently slow, Cython has been used to speed up markers interpolation. Most of the original Python code has been ported to Cython with minor modifications, just adding static typing and using pointers for arrays. The net increase of speed with this simple technique is almost three orders of magnitude (Table 1). The performance has been further improved by threading the interpolation methods. Thanks to Cython [Beh11], the Global Interpreter Lock (GIL) can be removed to make the threads concurrent. Loops are split into threads and each of them owns only a small section of the block of markers assigned locally to every processor. More tests are now taking place on the HPC facilities provided by CINECA to understand the scalability and further optimize the code.

Final Remarks

PyGmod shows that it is possible to build a simulation code that runs efficiently on HPC computers with a small programming effort. Available third party Python packages (Figure 1) greatly reduced the work needed to parallelize the algorithms. Petsc4py, ga4py, mpi4py and h5py are efficient and handle of all the necessary communication. Pure Python code can be optimized further by using different switches or methods provided by external packages (e.g. equation solvers). PyGmod is young code that works without any C or Fortran. It can be modified with minor effort, adapted to the needs of the research, and extended including other geodynamic phenomena like melting, fluid migration, phase changes, etc.. Open-source and efficient libraries and packages available in the Python universe overcome the myth that Python is only a scripting language not suited for computationally intensive purposes or that cannot be used on HPC facilities.

Interpolation	Speedup
Pure Python	1
Cython	725
Cython (2 Threads)	1187
Cython (4 Threads)	2056

TABLE 1: *Performance comparision between interpolation code adopting Cython and threading.*

References

[Ame00] P.R. Amestoy, I.S. Duff, J.Y. L'Excellent, *Multifrontal parallel distributed symmetric and unsymmetric solvers*, Comput. Methods in Appl. Mech. Eng., 184:501-520, 2000.

[Armci] ARMCI-MPI: https://github.com/jeffhammond/armci-mpi.

[Beh11] S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D. Sverre Seljebotn, K.Smith, *Cython: The Best of Both Worlds*, Computing in Science & Engineering, 13(2):31-39, 2011.

[Bra86] J.U. Brackbill, H.M. Ruppel, *FLIP: A method for adaptively zoned, particle-in-cell calculations of fluid flows in two dimensions*, Journal of Computational Physics, 65(2): 314-343, 1986.

[Buc04] W.R. Buck, *Consequences of Asthenospheric Variability on Continental Rifting*, in Rheology and Deformation of the lithosphere at comntinental margins, editors G.D. Karner, B. Taylor, N.W. Driscolland D.L. Kohlstedt, Columbia University Press, 1-31, 2004.

[Cfg] R. Dennis, E. Courtwright, https://github.com/DiffSK/configobj.

[Cin] Cineca, http://www.cineca.it.

[Col13] A. Collette, *Python and HDF5, Unlocking Scientific Data*, O'Riley ed., 152 pp, 2013.

[Dal11] L. Dalcin, P. Kler, R. Paz, A. Cosimo, *Parallel Distributed Computing using Python*, Advances in Water Resources, 34(9):1124-1139, 2011.

[Dal05] L. Dalcin, R. Paz, M. Storti, *MPI for Python*, Journal of Parallel and Distributed Computing, 65(9), 1108-1115, 2005.

[Deu08] Y. Deubelbeiss, B.J.P. Kaus, *Comparison of Eulerian and Lagrangian numerical techniques for the Stokes equations in the presence of strongly varying viscosity*, Physics of the Earth and Planetary Interiors, 171:92-111, 2008.

[Ger10] T.V. Gerya, *Introduction to Numerical Geodynamic Modelling*, Cambridge University Press ed., 345 pp, 2010.

[Ger07] T.V. Gerya, D.A. Yuen, *Robust characteristics method for modelling multiphase visco-elasto-plastic thermo -mechanical problems*, Phys. Earth Planet. Interiors, 163:83-105, 2007.

[Ger03] T.V. Gerya, D.A. Yuen, *Characteristics-based marker-in-cell method with conservative finite-differences schemes for modeling geological flows with strongly variable transport properties*, Phys. Earth Planet. Interiors, 140: 293-318, 2003.

[Gil13] M. Gilge, *IBM System Blue Gene Solution Blue Gene/Q Application Development*, IBM RedBook ed., 188 pp, 2013.

[Gor06] W. Gorczyk, T.V. Gerya, J.A.D. Connolly, D.A. Yuen, M. Rudolph, *Large-scale rigid-body rotation in the mantle wedge and its implications for seismic tomography*, G^3, 7, doi:10.1029/2005GC001075, 2006

[Hun07] J.D.Hunter, *Matplotlib: A 2D graphics environment*, Computing In Science & Engineering, 9(3):90-95, 2007.

[Jac97] J. Jackson, T. Blenkinsop, *The Bilila-Mtakataka fault in Malawi: An active, 100-km long, normal fault segment in thick seismogenic crust*, Tectonics 16(1):137-150, 1997.

[Jar11] M. Jarosinskia, F. Beekmanb, L. Matencob, S. Cloetingh, *Mechanics of basin inversion: Finite element modelling of the Pannonian Basin System*, Tectonophysics, 502:121-145, 2011.

[Jon01] E. Jones, T. Oliphant, E. Peterson, *SciPy: Open Source Scientific Tools for Python*, http://www.scipy.org/, 2001.

[Li03] X.S. Li, J. W. Demmel, *SuperLU_DIST: A Scalable Distributed-Memory Sparse Direct Solver for Unsymmetric Linear Systems*, CM Trans. Mathematical Software, 29(2):110-140, 2003.

[McK69] D. McKenzie, R.L. Parker, *The North Pacific: an example of tectonics on a sphere*, Nature, 216(5122): 1276–1280, 1967.

[Min70] J.W Minear, M.F. Toksoz, *Thermal regime of a downgoing slab and new global tectonics*, Tectonics, 75(8):1397–1419, 1970.

[Mor68] W. Morgan, *Rises, Trenches, Great Faults, and Crustal Blocks*, Journal of Geophysical Research, 73(6):1959-1982, 1968.

[Nie06] J. Nieplocha, B. Palmer, V. Tipparaju, M. Krishnan, H. Trease, E. Apra, *Advances, Applications and Performance of the Global Arrays Shared Memory Programming Toolkit*, International Journal of High Performance Computing Applications, 20(2):203-231, 2006.

[Pas02] C. Pascal, S. Cloetingh, *Rifting in heterogeneous lithosphere: Inferences from numerical modeling of the northern North Sea and the Oslo Graben*, Tectonics, 21(6):1-15, 2002.

[Pat80] S. Patankar, *Numerical Heat Transfer and Fluid Flow*, Hemisphere Series on Computational Methods in Mechanics and Thermal Science, CRC Press ed., 180 pp, 1980.

[Pint] H.E. Grecco, *Pint*, http://pint.readthedocs.org/en/0.6/.

[Tur02] D.L. Turcotte, S. G. Schubert, *Geodynamics*, Cambridge University Press ed., 456 pp, 2002.

[Van11] S. van der Walt, S.C. Colbert, G. Varoquaux, *The NumPy Array: A Structure for Efficient Numerical Computation*, Computing in Science & Engineering, 13:22-30, 2011.

[Ver07] H.K. Versteeg, M. Malalasekera, *An Introductio to Computational Fluid Dynamics*, Pearson Education ed., 503 pp, 2007.

Qiita: report of progress towards an open access microbiome data analysis and visualization platform

The Qiita Development Team‡*

https://www.youtube.com/watch?v=TQzXwQ9Vx08

Abstract—Advances in sequencing, proteomics, transcriptomics and metabolomics are giving us new insights into the microbial world and dramatically improving our ability to understand microbial community composition and function at high resolution. These new technologies are generating vast amounts of data, even from a single study or sample, leading to challenges in storage, representation, analysis, and integration of the disparate data types.

Qiita (https://github.com/biocore/qiita) aims to be the leading platform to store, analyze, and share multi-omics data. Qiita is BSD-licensed, unit-tested, and adherent to PEP8 style guidelines. New code additions are reviewed by multiple developers and tested using Travis CI. This approach opens development to the largest possible number of experts in "-omics" fields. The heterogeneous data generated by these disciplines led us to use a combination of Redis, PostgreSQL, BIOM ([Atr10]), and HDF5 for relational and hierarchical storage. The compute backend is provided by IPython's parallel framework (http://ipython.org/). In addition, the project depends on mature Python packages such as Tornado (http://www.tornadoweb.org/en/stable/), click (http://click.pocoo.org/4/), scipy (http://www.scipy.org), numpy (http://www.numpy.org), and scikit-bio (http://scikit-bio.org) among others. Most notably, the analysis pipeline is provided by QIIME (http://qiime.org), with EMPeror (http://emperor.microbio.me) serving as the visualization platform for high-dimensional ordination plots, which can be recolored interactively and manipulated using the sample metadata.

By providing the database and compute resources at http://qiita.microbio.me to the global community of microbiome researchers, Qiita alleviates the technical burdens, such as familiarity with the command line or access to compute power, that are typically limiting for researchers studying microbial ecology, while at the same time promoting an open access culture. Because Qiita is entirely open source and highly scalable, developers can inspect, customize, and extend it to suit their needs regardless of whether it is deployed as a desktop application or as a shared resource.

Index Terms—Microbiome, multi-omics, open science, metagenomics, metatranscriptomics, metaproteomics, metabolomics

Introduction

In recent years, the importance of microbes, including bacteria, archaea, fungi, and unicellular eukaryotes, in ecological communities has been extensively studied ([Atr11], [Atr06]). As the costs of analytical techniques such as DNA sequencing have continued their dramatic decline and samples become relatively easy to collect, large volumes of data and new data types have allowed for the characterization of the potent effects that microbial communities can impart on both host-associated ([Atr07], [Atr03]) and environmental ([Atr09]) health. The myriad techniques that can be used to characterize each individual sample allow researchers to understand these communities in previously unattainable detail, but also pose new challenges for integrating the results from multiple levels of observational data into a coherent picture. These techniques are colloquially called "omics" techniques and allow researchers to study the entire collections of genes (genomics), gene transcripts (transcriptomics), proteins (proteomics), and metabolites (metabolomics) represented in samples.

The genome of an organism is all of its genetic material; the "metagenome" of an environmental sample is the union of all of the genomes present in the sample. Since the genome of an organism defines the organism's biological capabilities, metagenomic analysis allows researchers to approach the question of what are the organisms in a sample capable of doing, collectively? Current techniques for performing metagenomic analysis fragment the metagenome into small pieces, which are then sequenced in massively parallel fashion, and genes are identified by comparison to references containing known genes. This technique results in a highly detailed view, but is relatively expensive due to the amount of sequencing that must be performed and the computational effort required ([Atr13]). A less detailed (but much cheaper and still very useful) characterization of a microbial community can be attained by performing targeted sequencing of marker genes. Sequences from marker genes are commonly grouped by similarity into operational taxonomic units (OTUs), groupings that might correspond to species, or genera, or classes, etc. A powerful way to identify the OTUs present in a sample is to amplify and sequence genes encoding components of the ribosome (rather than all of the genes). The ribosome is a cellular component that translates transcripts into proteins that is shared across the tree of life. Because it is believed to be under neutral evolution, mutations accrue at a relatively consistent rate, allowing it to be thought of as a "molecular clock" that provides phylogenetic information about the organism it came from ([Atr15]). In bacteria and archaea, amplicons of the 16S small subunit ribosomal gene are the most commonly used, while in eukaryotes the analogous 18S small subunit ribosomal gene is used (although for fungi, often parts of the internal transcribed spacer region are included for additional phylogenetic signal).

The central dogma of biology is that genes are transcribed into messenger RNAs (mRNAs), which are then translated into specific proteins. Tight regulation at each level is required for proper

* *Corresponding author: robknight@ucsd.edu*
‡ *University of California, San Diego*

cellular function. If amplicon sequencing and metagenomics help answer the questions of who's there, and what are they capable of doing, transcriptomics help answer the question what genes are actually being expressed right now? Genes that are "on" can be recognized by the presence of mRNA transcripts that identifiably correspond to the gene. Sequencing these transcripts elucidates which genes are actually being expressed ([Atr04]).

However, even the added depth provided by transcriptomic analyses does not paint the full picture. First, because transcripts are continuously being generated and degraded by cellular processes, only a snapshot of the transcriptome can be obtained from a single sample. Second, the regulatory mechanisms that govern the translation of transcripts into proteins do not treat all transcripts uniformly. Indeed, the abundances of proteins in a cell correlate only weakly with the abundances of their respective transcripts, as reviewed in [Atr08]. Therefore, protein levels must be measured directly using proteomics techniques to answer the question of how actively are observed transcripts actually being expressed as proteins? Proteins are inherently more complex molecules than DNA and RNA, and proteomics techniques fundamentally differ from genomics and transcriptomics techniques as they do not sequence nucleic acids. Instead of using genetic sequencers, instruments called mass spectrometers are used to fragment proteins and analyze the resulting charged peptides. The spectrum of peptides produced from a fragmented protein identifies it like a fingerprint (reviewed in [Atr01]).

The last "omics" technique considered here, metabolomics, provides an even more detailed view of cellular function by observing the presence of specific metabolites or all of the metabolites in a sample. Identifying the metabolites present in an organism (or group of organisms) helps answer the question, to what extent are these organisms interacting with and affecting their environments? Similar to proteomics, mass spectrometers are used to identify compounds and gauge metabolic interactions (reviewed in [Atr05]).

More and more commonly, studies are employing two or more of these techniques in "multi-omic" analyses of samples. Integrating these analyses and gaining biological insights from the preponderance of data resulting from each applied technique is a considerable challenge. For each technique, computational tools that process and digest raw data have been developed to varying levels of maturity, but orchestrating these tools into a coherent multi-omic analysis package has not yet been accomplished.

Moreover, the extremely rich datasets generated by each multi-omic study are valuable resources that can form the basis of subsequent "meta-analyses," wherein the original data are augmented with data from other new or existing studies. Meta-analysis has already been shown to be a powerful approach (Mason et al. 2014), and the potency of the approach increases as individual studies provide more and more detailed characterizations of their samples, enabling reuse of the data. The power of this approach underscores the scientific community's need for centralized resources for standardized, open access data.

Here, we present a progress report of Qiita, a multi-omic platform for meta-analysis that stresses standardization of data formats, open access to data and results, and methods for integrating samples across studies. As we design Qiita, we intend to account for the most common use-cases that a modern microbiome researcher will face. The following list briefly describes tasks that are streamlined using Qiita:

- Perform a microbiome analysis without any required knowledge of command line tools.
- Deposition of biological sequences into a public data repository, in specific the European Bioinformatics Institute's European Nucleotide Archive (ENA).
- Searching for studies based on sample and study metadata.
- Hosting of sequence data, sample metadata and processed files like BIOM tables.
- Provide a platform to collaboratively work on a dataset.
- Combine one or more studies into a single dataset to perform further specialized analyses.
- Analyze and organize different data types (16S, 18S, WGS, etc) into a single location where the sample metadata is enforced to be consistent across representations.

The list of tasks above, while not comprehensive, exemplifies some commonly encountered scenarios where Qiita is a powerful tool. Please also note that the last point regarding integration of multiple data types is a work in progress at this point. Currently, only portions of the 16S workflow are implemented, but there are plans for adding additional workflows (see future directions). Although other platforms and individual tools exist that are capable achieving one or more of these goals independently, such ad hoc pipelines are often troublesome, time consuming, and error prone.

Structure and Operation

Modular organization

Qiita follows a model-view-controller (MVC) architecture, with a Python module for each level (qiita_db, qiita_pet, and qiita_ware, respectively). Modularizing the platform in this way allows for flexibility in the case that different technologies are adopted as the project matures. It also permits customizability, since a user maintaining a deployment can choose to replace any of these modules with one of their own design as long as it operates using the same inter-module APIs.

Qiita-DB

The qiita_db module defines a database schema in PostgreSQL (http://www.postgresql.org/) that serves to store and relate study metadata as well as system data. The schema was designed in DBSchema (http://www.dbschema.com/), which provides a convenient GUI for defining the table structure, setting constraints, and generating documentation. Although the project is under heavy development, there are active deployments of Qiita (e.g., http://qiita.microbio.me and http://qiita.ibdmdb.org). As development progresses and modifications to the database schema are required, they must be implemented and deployed in a way that preserves active deployments' data. Therefore, migrations are performed using a combination of SQL- and Python-based patches. In order to facilitate brand new deployments as well as accommodate upgrading active deployments, Qiita's GitHub repository contains the schema definition ab initio as well as all patches needed to upgrade it (modifying data of active deployments as needed) to the most up-to-date version. The database itself contains information about the currently deployed patch version so that what patches need to be applied, if any, can easily be determined. Psycopg (http://initd.org/psycopg/) provides Python bindings for interacting with PostgreSQL.

Several aspects of the data model itself bear mentioning. Users are identified by an email address and a password supplied upon

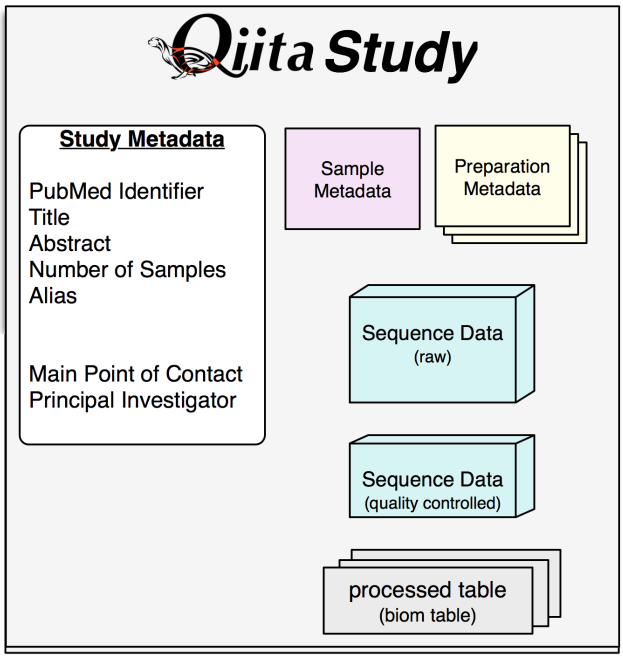


Fig. 1: *Core structure of a study in Qiita. The study metadata broadly describes information about the study, the sample and preparation metadata refer to the biological specimens and their preparation method, the raw data refers to the files as generated by the instrument, quality controlled sequence data is generated for convenience and is used to create the processed tables.*

account creation. Passwords are salted and hashed with hashlib using bcrypt (https://github.com/pyca/bcrypt/). After users verify their email addresses, they are free to create "studies" by supplying some basic information such as the title of the study, an abstract, and what kind of environment is being studied, et al. Most of this information can be edited at any time after creation. Each study serves as a logical container for its associated data, metadata, and results files.

Because the system was designed with multi-omic analyses in mind, a distinction is made between metadata associated with the samples themselves (sample metadata) and metadata associated with preparations of those samples for biological processing (preparation metadata). In other words, sample metadata is invariant information about the samples themselves (e.g., the gender or age of the subject that was sampled), while preparation metadata for a 16S amplicon analysis of those samples would differ from preparation metadata for a proteomic analysis of the same samples. Note that the set of samples in two different preparation metadata might not overlap (or might overlap only partially) since not all samples are analyzed using all available techniques (see Figure 1). For example, the database currently contains a public study of about 100 samples taken from the site of the Deepwater Horizon oil spill in 2011 (study ID 1197; [Atr09]) where both 16S data and metagenomic data were collected. Some of the metadata collected (including the amounts of dissolved inorganic nitrogen, dissolved phosphate, amount of toluene, etc.) is specific to the samples themselves and will not vary with preparation; this is the sample metadata. On the other hand, some of the metadata is specific to a particular preparation of the samples for 16S analysis (including the region that was amplified, the primers that were used, the date the sequencing was performed, etc.); this is one set of preparation metadata. The subset of the full ~100 samples that were prepared for 16S analysis would be represented in this preparation metadata. For the metagenomic preparation, a smaller subset of the full ~100 samples were analyzed, so the metadata for that preparation would only contain information on those samples, and the data tracked would differ from the preparation metadata for the 16S analysis (for example, the preparation metadata for the metagenomic analysis would not contain a column for the 16S region).

Qiita (and the administrator(s) in a multi-user system) attempts to standardize as many fields of the metadata as possible using controlled vocabularies and ontologies when available. However, users are permitted to supply whatever sample and preparation metadata they deem relevant to their studies. Since the data that is supplied by users cannot be predicted a priori, a dynamic approach to storing the metadata must be taken. New tables are created dynamically using a consistent naming convention to keep track of each study's sample metadata and various preparation metadata, and another table keeps track of what fields are available in each metadata table and what the datatype of the field is. Like metadata fields, processing parameters are also standardized in order to minimize the impact of technical effects that would arise from heterogeneous processing. Tables for each key processing step, including demultiplexing, quality filtering, and OTU picking, keep track of these standard sets of parameters.

The qiita_db module also contains Python objects and utility functions that mediate filesystem and database interactions, similar in many respects to an object-relational mapper (ORM). Uploaded metadata files and raw data files (e.g., sequence data from a sequencing instrument) are stored in a directory structure with indirection to support horizontal scaling of file systems. Unlike the information in metadata files, the contents of raw data files are not stored in the database. Instead, the filepaths are recorded. This design facilitates processing the raw data files using external programs (e.g., programs that are implemented or wrapped in qiita_ware; see below) that need filehandles.

Qiita-pet

The qiita_pet module defines components supporting a browser-based user interface. In a single-user deployment, tornado (http://www.tornadoweb.org/) handles all requests and serves all pages. In a multi-user deployment, nginx (http://nginx.org/) is required to serve downloads. While tornado is proficient at serving small or moderate files in small chunks, serving very large files can bog down the single-threaded server. Instead, tornado can be used to handle the initial request and to determine whether the file should be served (e.g., whether user has permission to access the file) before handing the request off to nginx to perform the actual file transfer. Another good use of nginx is as a load balancer sitting in front of several tornado web servers running on different ports.

Tornado templates provide a user interface that is based largely on bootstrap (http://getbootstrap.com/) and jQuery (https://jquery.com/). Other packages and extensions are used for various interface elements (for example, WTForms (https://github.com/wtforms/wtforms) is used for handling some form data, chosen (http://harvesthq.github.io/chosen/) provides improved select and multiple select form elements, and DataTables (https://www.datatables.net/) provides interactive and pleasantly formatted tabular displays). Asynchronous JavaScript and XML (AJAX) is used for the majority of asynchronous client-server communication,

although websockets are employed when push notifications are useful (for example, when the server wants to notify a client that a processing job has completed).

Qiita-ware

The qiita_ware module contains functions for manipulating input files, dispatching processing jobs, and performing operations on results files (e.g., submitting them to external data repositories like the European Bioinformatics Institute). Qiita is designed to be highly parallelizable through the use of IPython engines. Currently, the best supported workflow is for performing 16S amplicon analysis. For this workflow, scripts in the Quantitative Insights Into Microbial Ecology package (QIIME; [Atr02]) are executed from IPython engines to process users' input files and generate visualizations. Jobs are dispatched using mustached-octo-ironman (MOI; https://github.com/biocore/mustached-octo-ironman/), which serves the dual purpose of managing the submission of jobs and communicating their statuses to the browser-based interface through a websocket using pubsub calls with Redis as a message broker. Two packages are used to interface with Redis: redis-py (https://github.com/andymccurdy/redis-py) and toredis (https://github.com/mrjoes/toredis/), the latter of which provides a non-blocking mechanism for handling pubsub with Redis.

Command line interface

In addition to the browser-based interface provided by qiita_pet, a command line interface (CLI) is also available. Qiita's scripts directory contains Python scripts that provide a command line interface to many of the system's capabilities through the click framework (http://click.pocoo.org/4/). The top-level qiita click group has subgroups (db, ware, and pet) for interfacing with each of the aforementioned modules along with a maintenance subgroup for performing administrative actions and probing the system's status. Note that all of the CLI commands assume that the user executing the commands has administrator access to Qiita.

Data access control

Qiita can be deployed as either a single-user or multi-user system. A single-user deployment enforces virtually no data access restrictions; the sole user has ownership of all data in the system. The single-user deployment is intended for users who want a system that organizes their data and provides a graphical interface for performing analyses and meta-analyses. A multi-user deployment is more complex and depends on a group of administrators (at least one administrator is required) who moderate and curate additions and certain modifications to data in the system. Access to users' data is restricted based on the data's status, which can be one of sandboxed, private, or public.

Data that is sandboxed or private is visible only to its owner and other users with whom the owner explicitly chooses to share the data; data that is public is visible to all users of the system. Any user is free to upload, process, and explore his or her own sandboxed data using the full suite of tools provided, but the data is only minimally validated. The purpose of the sandboxed status is to allow users to get a quick look at their data -- and even rapidly compare it to other data in the system -- before expending a potentially large amount of time and effort detailing and correcting metadata-related minutiae.

Private data is assured to be maximally compatible with existing data in the system. Because computational validation can provide only a limited guarantee of compatibility, administrator approval is required to change a study from sandboxed to private status after a manual curation process. Manual curation helps ensure that new metadata uses controlled vocabulary and ontology terms where available, that applicable standards are followed (e.g., MIMARKS for marker gene sequence-related metadata), and that new user-defined metadata fields are introduced sparingly (for example, if there were already a field called "sex" in one or more existing studies, the curator would suggest amending a proposed "gender" field to avoid having multiple fields that contain the same class of information). It is possible but discouraged to revert data from private to sandboxed since another round of curation would be required to make it private again.

Once data is private, it is up to the user to decide if and when to make the data public at his or her discretion. At this stage, all users of the system are permitted to download and analyze the data, and the owner of the data can submit the data and metadata to a public repository such as the European Bioinformatics Institute (EBI; https://www.ebi.ac.uk). Reverting data from public to private has limited efficacy (since other users might have downloaded and/or performed analyses on the data) and requires administrator action.

Configuration

By default, Qiita will look for a configuration file in a default location where an example configuration file is supplied. This behavior can be overridden by setting the QIITA_CONFIG_FP environment variable. This configuration file controls the behavior of various aspects of Qiita and its dependencies, including Postgres, IPython (http://ipython.org; [Atr12]), Redis (http://redis.io/), and MOI.

Roadmap of future directions

Qiita is currently in alpha release and under active development. New functionality is continually being added, and these changes have the potential to affect all of the aforementioned submodules and interfaces, but any changes will maintain backwards compatibility with existing deployments. One planned enhancement will allow deployments to be "branded," so that not every Qiita deployment looks identical. In addition to supporting cosmetic changes, for example to logos or graphics, we will support the specification of multiple "portals" that coexist on one system and access a common database, but provide access to only desired subsets of the data. For example, we plan to introduce an Earth Microbiome Project ([Atr06]; EMP) portal that provides access to only EMP studies.

The most significant change currently planned will be the implementation of a plug-in system designed to support modular expansion of the system with new processing capabilities while maintaining a common user interface. We intend the plug-in system to support extensions to both the database schema and the Python framework by providing common interfaces to the main system. To demonstrate the feasibility of this approach, the current 16S analysis pipeline will be migrated to be the first plug-in. New users should note that right now, only portions of the 16S workflow are implemented. However, the data model and modularity that we have designed and built into the system will facilitate the addition of additional pipelines (including metagenomics, metaboloomics, and proteomics) through this upcoming plug-in system.

Another important change will affect data processing. Right now, in order to ensure consistent processing workflows, users can upload only raw data for processing on the system using standardized methods. However, the ability to enter the data processing

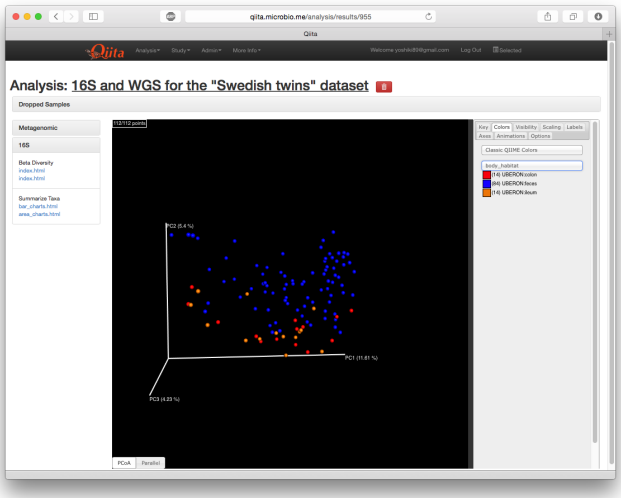


Fig. 2: *Embeded beta diversity plot displayed using EMPeror showcasing an example dataset where samples are colored by the body habitat from where they were collected.*

workflows at downstream steps is a frequently requested feature that we plan to support. For the 16S analysis pipeline, users will be able to upload sequence files that have already been demultiplexed and/or quality filtered (e.g., by the sequencing center) or even BIOM tables of OTU picking results. The downside to these alternative pipeline entry points is that the standardized processing that is applied to other studies in the system cannot be guaranteed. For this reason, processing results that do not originate from raw data cannot be made available for public use like other results.

Due to the size and complexity of this nascent project, Qiita's documentation for users and developers is continuously evolving. For developers, the Numpydoc-formatted docstrings (https://github.com/numpy/numpydoc) that have already been added, which describe the system's Python objects and functions, will be rendered using sphinx (http://sphinx-doc.org/) and supplemented by markdown documents that provide additional details or instructions. For users, separate documentation will be made available covering key design concepts and how to interact with the system through the web interface.

Interactive Visualizations

Allowing users to share, process, and combine their datasets easily does not ensure that interesting conclusions or insights will be generated. Only by carefully cross-examining results with sample metadata can correlations be observed and hypotheses developed. When working with large datasets (or combinations of datasets), effective visualizations are indispensable for presenting information in an intuitive manner and accelerating hypothesis generation. Collaborative efforts benefit greatly from visualizations that are portable and lightweight, qualities that allow researchers to communicate results and ideas to one another seamlessly.

One application that has proven useful to a large number of microbiome researchers is EMPeror ([Atr14]). While many existing tools are capable of displaying scatter plots, none of them actually integrates the sample metadata into the visualization on the fly while providing publication quality graphics. EMPeror accomplishes this integration, for example Figure 2 shows EMPeror executing within Qiita, meaning that users can interactively recolor points in space based on a metadata field using an intuitive browser-based interface. Other graphical manipulations of the points are also available, such as resizing or changing the opacity of arbitrary subsets of points. These capabilities shorten the gap between running a purely exploratory analysis and producing publication-quality figures.

As the development of EMPeror matures, other enhancements are being added, including the ability to view and interact with EMPeror plots from within an IPython notebook, supplementing textual descriptions with interactive plots. This feature is still in active development and will be available in a future release.

Since 2010, QIIME has provided the tools that utilize a sample's metadata to visualize taxonomic summaries, rarefaction curves, ordination plots, and even histograms of beta diversity distances. However these tools are usually limited, either because they are not extensible, lacking an interface that other web applications might use, or because they do not effectively provide both interactive and publication-quality static plots. The need for interactive, lightweight, and extensible browser-based visualization tools like EMPeror grows with the popularity of web-based scientific analysis platforms like BaseSpace (https://basespace.illumina.com/), Galaxy (https://galaxyproject.org/), iPlant (http://www.iplantcollaborative.org/), and KBase (https://kbase.us/), among others.

Conclusions

Qiita provides a centralized resource where researchers can add their multi-omic datasets and process them in a standardized manner that maximizes their utility in meta-analyses. Organizing data and results, managing computational work, and interacting with all of the available tools poses a significant technical burden for researchers to surmount. Single-user deployments of Qiita help ameliorate this burden for individuals. Meanwhile, multi-user deployments serve as hubs that coordinate research efforts by facilitating the sharing of data and communication between users. Furthermore, a large, centralized, multi-user deployment that is maintained by the Qiita developers and staff at the University of California, San Diego, is available at http://qiita.microbio.me, where free data storage and compute clusters are provided to users. Regardless of the mode of deployment, a growing set of interactive results visualizations are provided by browser-based tools like EMPeror to accelerate the generation and exploration of new hypotheses.

References

[Atr01] Aebersold R, Mann M, "Mass spectrometry-based proteomics," Nature 2003 Mar 13;422(6928):198-207.

[Atr02] Caporaso JG, Kuczynski J, Stombaugh J, Bittinger K, Bushman FD, Costello EK, Fierer N, Pea AG, Goodrich JK, Gordon JI, Huttley GA, Kelley ST, Knights D, Koenig JE, Ley RE, Lozupone CA, McDonald D, Muegge BD, Pirrung M, Reeder J, Sevinsky JR, Turnbaugh PJ, Walters WA, Widmann J, Yatsunenko T, Zaneveld J, Knight R, "QIIME allows analysis of high-throughput community sequencing data," Nature Methods 2010 May 7;7(5):335-6.

[Atr03] Costello EK, Lauber CL, Hamady M, Fierer N, Gordon JI, Knight R, "Bacterial community variation in human body habitats across space and time," Science. 2009 Dec 18;326(5960):1694-7. doi: 10.1126/science.1177486.

[Atr04] Creecy JP and Conway T, "Quantitative bacterial transcriptomics with RNA-seq," Curr Opin Microbiol. 2015 Feb;23:133-40. doi: 10.1016/j.mib.2014.11.011. Epub 2014 Dec 5.

[Atr05] Dettmer K, Aronov PA, Hammock BD, "Mass spectrometry-based metabolomics," Mass Spectrom Rev. 2007 Jan-Feb;26(1):51-78.

[Atr06] Gilbert JA, Jansson JK, Knight R, "The Earth Microbiome project: successes and aspirations," BMC Biology 2014, 12:69 doi:10.1186/s12915-014-0069-1.

[Atr07] Goodrich JK, Di Rienzi SC, Poole AC, Koren O, Walters WA, Caporaso JG, Knight R, Ley RE, "Conducting a microbiome study," Cell 2014,158(2):250-62. doi:10.1016/j.cell.2014.06.037.

[Atr08] Maier T, Güell M, Serrano L, "Correlation of mRNA and protein in complex biological samples," FEBS Lett. 2009 Dec 17;583(24):3966-73. doi: 10.1016/j.febslet.2009.10.036.

[Atr09] Mason OU, Scott NM, Gonzalez A, Robbins-Pianka A, Bælum J, Kimbrel J, Bouskill NJ, Prestat E, Borglin S, Joyner DC, Fortney JL, Jurelevicius D, Stringfellow WT, Alvarez-Cohen L, Hazen TC, Knight R, Gilbert JA, Jansson JK, "Metagenomics reveals sediment microbial community response to Deepwater Horizon oil spill," ISME J. 2014 Jul;8(7):1464-75. doi: 10.1038/ismej.2013.254.

[Atr10] McDonald D, Clemente JC, Kuczynski J, Rideout JR, Stombaugh J, Wendel D, Wilke A, Huse S, Hufnagle J, Meyer F, Knight R, Caporaso JG, "The Biological Observation Matrix (BIOM) format or: how I learned to stop worrying and love the ome-ome," Gigascience 2012 Jul 12;1(1):7. doi: 10.1186/2047-217X-1-7.

[Atr11] NIH HMP Working Group, Peterson J, Garges S, Giovanni M, McInnes P, Wang L, Schloss JA, Bonazzi V, McEwen JE, Wetterstrand KA, Deal C, Baker CC, Di Francesco V, Howcroft TK, Karp RW, Lunsford RD, Wellington CR, Belachew T, Wright M, Giblin C, David H, Mills M, Salomon R, Mullins C, Akolkar B, Begg L, Davis C, Grandison L, Humble M, Khalsa J, Little AR, Peavy H, Pontzer C, Portnoy M, Sayre MH, Starke-Reed P, Zakhari S, Read J, Watson B, Guyer M, "The NIH Human Microbiome Project," Genome Res. 2009 Dec;19(12):2317-23. doi: 10.1101/gr.096651.109.

[Atr12] Pérez F, Granger B, "IPython: A System for Interactive Scientific Computing," Computing in Science and Engineering, vol. 9, no. 3, pp. 21-29, May/June 2007, doi:10.1109/MCSE.2007.53. URL: http://ipython.org

[Atr13] Scholz MB, Lo CC, Chain PS, "Next generation sequencing and bioinformatic bottlenecks: the current state of metagenomic data analysis," Curr Opin Biotechnol. 2012 Feb;23(1):9-15. doi: 10.1016/j.copbio.2011.11.013.

[Atr14] Vázquez-Baeza Y, Pirrung M, Gonzalez A, Knight R, "EMPeror: a tool for visualizing high-throughput microbial community data," Gigascience 2013 Nov 26;2(1):16. doi: 10.1186/2047-217X-2-16.

[Atr15] Woese CR, "Bacterial evolution," Microbiol Rev. 1987 Jun; 51(2): 221–271.

Python in Data Science Research and Education

Randy Paffenroth[§*], Xiangnan Kong[‡]

https://www.youtube.com/watch?v=EUEHOYl0mRg

Abstract—In this paper we demonstrate how Python can be used throughout the entire life cycle of a graduate program in Data Science. In interdisciplinary fields, such as Data Science, the students often come from a variety of different backgrounds where, for example, some students may have strong mathematical training but less experience in programming. Python's ease of use, open source license, and access to a vast array of libraries make it particularly suited for such students. In particular, we will discuss how Python, IPython notebooks, scikit-learn, NumPy, SciPy, and pandas can be used in several phases of graduate Data Science education, starting from introductory classes (covering topics such as data gathering, data cleaning, statistics, regression, classification, machine learning, etc.) and culminating in degree capstone research projects using more advanced ideas such as convex optimization, non-linear dimension reduction, and compressed sensing. One particular item of note is the scikit-learn library, which provides numerous routines for machine learning. Having access to such a library allows interesting problems to be addressed early in the educational process and the experience gained with such "black box" routines provides a firm foundation for the students own software development, analysis, and research later in their academic experience.

Index Terms—data science, education, machine learning

Introduction

Data Science is a burgeoning field of study that lies at the intersection of statistics, computer science, and numerous applied scientific domains. As is common within such *interdisciplinary* domains of study, Data Science education, mentoring, and research draws ideas from, and is inspired by, several other domains such as the mathematical sciences, computer science, and various businesses and application domains. Perhaps just as importantly, students who wish to pursue education and careers in Data Science come from similarly diverse backgrounds. Accordingly, the challenges and opportunities of being an educator in such a domain requires one to reflect on appropriate tools and approaches that promote educational success. It is the authors' view, and experience, that the Python scripting language can be an effective part of the Data Science curriculum for several reasons such as its ease of use, its open source license, and its access to a vast array of libraries covering many topics of interest to Data Science.

Worcester Polytechnic Institute (WPI) has recently (fall 2014) begun admitting students into its new Data Science Master's degree program and, as of spring 2015, has also initiated a Data Science Ph.D. program. Even at this early stage, the program has been quite fortunate to receive many more applications from students then it can reasonably admit. The authors have the pleasure of being faculty members in the program and have the honor of teaching a number of the courses on offer. In addition, as long time Python users (for one of them since 1997 in fact [Paf99]), the authors were intrigued by the possibility of leveraging Python in the graduate Data Science curriculum and this monograph describes some of the experiences, both successes and challenges, gained from that effort. Of course, it is much too early to make any comments on the sustained effect of using Python for graduate Data Science education, however, perhaps the reader will find some value in the authors' experiences, even at this early date.

Of course, we are not the first to suggest Python's effectiveness in an education and research environment. In fact, the Python scripting language is quite popular in numerous problem domains and Python has seen wide used in education, see e.g., [Mye07] and [Sta00]. In fact, it ranks quite highly in many surveys of programming language popularity [OGr14], it is seeing substantial growth within the Data Science community [Sin14], and is generally speaking quite easy to learn [Lut13].

However, it is not our purpose here to focus on Python in general, but rather to focus on its use in *Data Science* education and research. With that in mind, herein we will focus on a small number of case studies that provide insights into how we have leveraged Python in that domain.

In particular, herein we will discuss the use of Python at three different levels of Data Science education and research. First, one of the courses that is offered as part of our Data Science curriculum, and in which the authors and others have leveraged Python, is DS501–"Introduction to Data Science". The idea of DS501 is to provide an introductory overview of the many fields that comprise Data Science and it is intended that DS501 be one of the first classes a new student takes when entering the program. Second, one of the authors has also used Python to support MA542–"Regression Analysis". MA542 is a somewhat more advanced class that is a (core) elective in the Data Science program as well as being a class taken by many students who are seeking degrees in the Mathematical Sciences department. Finally, the authors mentor a number of students' research projects within the Data Science Program, the Mathematical Sciences Department, and the Computer Science Department. Many of these research projects leverage Python in various ways, and having access to a common code base allows the various student projects to build off of one another.

Two key themes will permeate our discussion in the following

* *Corresponding author: rcpaffenroth@wpi.edu*
§ *Worcester Polytechnic Institute, Mathematical Sciences Department and Data Science Program*
‡ *Worcester Polytechnic Institute, Computer Science Department and Data Science Program*

sections. First, the Python community provides easy access to a vast array of libraries. Even though Data Science education and research draws from many other domains, Python was always there with a library ready to support our work. Second, and perhaps more subtly, having access to a language which is easy to use, but provides access to many advanced libraries, allows one to carefully craft the difficulty and scope of homework assignments, class projects, and research problems. In particular, Python allows students to tackle specific aspects of real world problems, without being overly burdened with details that are extraneous to their particular learning objectives. Both properties make Python particularly advantageous.

Finally, in an effort to assist the reader who is not steeped in Python, we will attempt to provide a range of references so that the interested reader can learn more about the specific libraries we leverage in our work. While in certain circles the libraries we mention are well known, we thought it would be useful to collect these references together into a single document.

DS501 Introduction to Data Science

DS501–"Introduction to Data Science" is intended to be one of the first classes a new student takes when entering the Data Science program at WPI, and the goal is to provide a high level overview of a wide swath of the material that a burgeoning Data Scientist should know. In particular, the course is described as:

> This course provides an overview of Data Science, covering a broad selection of key challenges in and methodologies for working with big data. Topics to be covered include data collection, integration, management, modeling, analysis, visualization, prediction and informed decision making, as well as data security and data privacy. This introductory course is integrative across the core disciplines of Data Science, including databases, data warehousing, statistics, data mining, data visualization, high performance computing, cloud computing, and business intelligence. Professional skills, such as communication, presentation, and storytelling with data, will be fostered. Students will acquire a working knowledge of data science through hands-on projects and case studies in a variety of business, engineering, social sciences, or life sciences domains. Issues of ethics, leadership, and teamwork are highlighted. – http://www.wpi.edu/academics/catalogs/grad/dscourses.html

As one might imagine from such an ambitious description, finding the right level of detail for the course can be quite challenging. One must consider the fact that many of the students have quite varied backgrounds. Some students are experts in mathematics and have less training in computer science or software development, while others find themselves in the opposite situation.

Space does not allow for a fulsome description of the class content and, in any event, such a discussion would distract us from our focus on Python. However, in the authors' view, one important feature of such a class is that the students should be able to *get "their hands dirty" playing with real data* both early and often. Students can often find inspiration by seeing the ideas developed as part of the lectures being put to use on problems of practical interest.

With all of the above in mind, it was decided to have four interconnected *case studies* as major learning activities for the class. Each case study is intended to build upon the previous one with the students solving interesting and pertinent problems in Data Science at every step. Accordingly, our focus here will be on these case studies and the substantial role that Python had to play in their development.

Case Study One

The idea of the first case study in DS501 is to perform basic data gathering, cleaning, and collection of statistics. For this case study we choose our data source to be the Twitter Data Streaming API [Rus13], [Twi15]. Already, Python begins to demonstrate its usefulness, since it allows ready access to the Twitter API through the python-twitter library [Ptw15].

Another key feature of the case studies in DS501 is that we chose to use IPython notebooks [Per07] both to provide the assignments to the students and to have the students submit their results. Using IPython notebooks for both of these tasks provided a number of advantages. First and foremost, it let the instructors provide the students with skeleton implementations of their assignments and allowed the students to focus on their learning objectives. Second, it provide a uniform and easy to use development environment for the students' efforts. As DS501 is not a programming class, per se, leveraging IPython notebooks made the introduction of Python to those students unfamiliar with it substantially easier.

For example, in the IPython notebooks we are able to provide code examples to get the students started with their development work. For example, we could provide code similar to the following as a launching pad for their efforts (see [Twi15] for details and code example is based upon [Rus13]):

```
import twitter
#-----------------------------------------------

# Define a Function to Login Twitter API
def oauth_login():
    # Go to http://twitter.com/apps/new to create an
    # app and get values for these credentials that
    # you'll need to provide in place of these empty
    # string values that are defined as placeholders.
    # See https://dev.twitter.com/docs/auth/oauth
    # for more information on Twitter's OAuth
    # implementation.

    CONSUMER_KEY = '<Insert your key>'
    CONSUMER_SECRET ='<Insert your key>'
    OAUTH_TOKEN = '<Insert your token>'
    OAUTH_TOKEN_SECRET = '<Insert your token>'

    auth = twitter.oauth.OAuth(OAUTH_TOKEN,
                               OAUTH_TOKEN_SECRET,
                               CONSUMER_KEY,
                               CONSUMER_SECRET)

    twitter_api = twitter.Twitter(auth=auth)
    return twitter_api

#-----------------------------------------------
# Your code starts here
# Please add comments or text cells in between
# to explain the general idea of each block of the
# code.  Please feel free to add more cells below
# this cell if necessary.
```

In this example we provide a skeleton that allows the students to focus on the objective of analyzing tweets and hashtags with frequency analysis and not have to struggle with the details of Twitter authentication. Using Python, and the skeleton code provided by the instructors, the student were able to gather and

analyze many thousands of tweets and learn important lessons about data gathering, data APIs, data storage, and basic analytics.

Case Study Two

Building upon the skills gained in the first case study, the second case study asks the students to analyze the MovieLens 1M Data Set [Mov15], which contains data about how users rate movies. The key learning objectives are to analyze the data set, make conjectures, support or refute those conjectures with data, and use the data to tell a compelling story. In particular, the students are not only asked to perform several technical tasks, but they must also propose a business question that they think this data can answer. In effect, they are expected to play the role of a Data Scientist at a movie company and they must convince "upper management", who are not presumed to be technically minded, that their conjecture is correct.

While a seemingly tall order for only the second case study, Python again shows its utility. In particular, just as in case study 1, the assignment is provided in an IPython notebook, and the student is required to submit their work in the same format, thereby leveraging the skills learned in the first case study.

However, in this case study we introduce several important Python libraries that support Data Science including Numpy [Wal11], matplotlib [Hun07], and, perhaps most importantly, pandas [McK10]. As is perhaps well known to the readers of this text, Numpy provides a vast selection of routines for numerical processing, including powerful array and matrix/vector classes, while matplotlib allows for plotting of data and generation of compelling figures. Finally, pandas provides many tools for data processing, including a structure called a DataFrame (inspired by a data structure with the same name in the R language [RCT13]), which facilitates many data manipulations. Note, we are certainly not the first to consider this collection of libraries to be important for Data Science, and this particular case study was inspired by the excellent book "Python for Data Analysis: Data Wrangling with Pandas, NumPy, and IPython", by Wes McKinney [McK12] (which is required reading for this particular assignment).

Many of the tasks in this case study revolve around question like:

- How many movies have an average rating over 4.5 overall?
- How many movies have an average rating over 4.5 among men? How about women?
- How many movies have a *median* rating over 4.5 among men over age 30? How about women over age 30?
- What are the ten most popular movies given a reasonable, student derived, definition of "popular"?

and the visualization of the data by way of:

- Plotting a histogram of the ratings of all movies.
- Plotting a histogram of the *number* of ratings each movie received.
- Plotting a histogram of the *average rating* for each movie.
- Plotting a histogram of the *average rating* for movies which are rated more than 100 times.
- Making a scatter plot of men versus women and their mean rating for every movie.
- Making a scatter plot of men versus women and their mean rating for movies rated more than 200 times.

Note, there are a number of important learning objectives that we wish to support. First, several terms are, intentionally, only vaguely defined in the assignment. For example, the precise definition of "popular" is left to the student to derive. As is often the case is real world Data Science, one of the key first steps of analysis is to decide precisely what the question of interest is. Second, the student is expected to make hypotheses or conjectures based upon the definitions they come up with. For example, the student might conjecture that men's and women's rating for certain genres are highly correlated, while for other genres their ratings more independent. Finally, the students must try to either prove, or just as interestingly, disprove their conjectures based upon the data.

Diving a bit more deeply into some of the specific functionality that we leverage in Python, we note that pandas [McK10] is particularly useful for these kinds of data analysis questions. In particular, to any Python aficionado, it is likely to be clear that there are many ways to process the data to answer the questions above, ranging from the brute force to the elegant.

To begin, we note that the MovieLens 1M Data Set itself is actually provided in three different files. First is a file containing the information regarding individual users, indexed by a unique *user_id*. Second is a file containing the information regarding each movie, indexed by a unique *movie_id*. Finally, and perhaps most importantly, is a file which contains ratings (and time stamps) indexed by a pair of *user_id* and *movie_id*.

Already we can perceive a thorny issue. Clearly, the questions of interest can only be answered by appropriate cross referencing between these three files. For example, all three files must be referenced to answer a question as seemingly straight forward as "how many action movies do men rate higher than 4?" While perhaps not too troublesome for students who are adept programmers, the cross referencing between the files presents an unnecessary impediment to less proficient students and overcoming this sort of impediment does not support the learning goals for this assignment.

Of course, a straightforward answer would be for the instructors to preprocess the data appropriately. However, using the power of Python one can easily arm the students with a general tool, while at the same time avoiding unnecessary hurdles. In particular, pandas has a merge function [PMe15] that provides exactly the required functionality in a quite general framework. In particular, one can use the code below to easily merge the three data files into a single DataFrame.

```
import pandas as pd
#-----------------------------------------------

# Read in the user data into a DataFrame
unames = ['user_id', 'gender', 'age',
          'occupation', 'zip']
users = pd.read_table('ml-1m/users.dat',
                      sep='::', header=None,
                      names=unames)

# Read in the rating data into a DataFrame
rnames = ['user_id', 'movie_id',
          'rating', 'timestamp']
ratings = pd.read_table('ml-1m/ratings.dat',
                        sep='::', header=None,
                        names=rnames)

# Read in the movie data into a Data Frame
mnames = ['movie_id', 'title', 'genres']
movies = pd.read_table('ml-1m/movies.dat',
                       sep='::', header=None,
                       names=mnames)

# Merge all the data into one DataFrame
data = pd.merge(pd.merge(ratings,
```

```
                                users),
                    movies)
```

Of course, even once the data files have been merged, there are many places where a student might fall astray. Fortunately, pandas provides another tool which allows for elegant and compact code, namely the *pivot-table*. For example, one can imagine writing complicated loops and conditionals to perform the task of printing out all movies that have a median rating of 5 by men or women. However, using pivot-tables, such a question can be answered with just three lines of code (using the Python 2 "print" statement versus the Python 3 "print()" function):

```
# Create a pivot table to aggregate the data
mean_ratings = data[data['age'] > 30].\
     pivot_table(values='rating',
                 rows='title',
                 cols='gender',
                 aggfunc='median')
# Only print out movies with at least one rating
print (mean_ratings[mean_ratings['M'].notnull()].\
     sort('M',ascending=False)['M'] > 4.5).nonzero()
print (mean_ratings[mean_ratings['F'].notnull()].\
     sort('F',ascending=False)['F'] > 4.5).nonzero()
```

Of course, one might be tempted to argue that having students develop their own code, rather than leveraging such *black box* routines leads to a deeper learning experience. While we certainly appreciate this point of view, we wish to emphasize that the class in question is an introductory Data Science class, and not a programming or data structure class. Accordingly, using Python, and the powerful features of libraries such as Pandas, allows us to focus on the Data Science learning goals, while at the same time allowing the students to utilize large scale, real world, and sometimes messy data sources. This theme of using Python to allow for focused learning goals, using real world data, is a key message of this text.

Case Study Three

The third case study is substantially more challenging than the second case study, but builds on the foundations already laid down. While case study two focused on analyzing *numerical* movie reviews, case study three focuses on detecting positive and negative reviews from raw text using natural language processing.

In particular, in case study three, the class turns its attention to the Movie Review Data v2.0 from http://www.cs.cornell.edu/people/pabo/movie-review-data. This data set contains written reviews of movies divided into positive and negative reviews, and the goal is to learn how to automatically distinguish between the two cases.

Of course, tackling such problems is well known to be difficult, and there are many open research problems in this domain. On the other hand, such problems are clearly of importance in many domains, and it is not at all difficult to get students interested in solving them. The question remains, how can students in their very first Data Science class be expected to approach such difficult and important problems, and still be able to make meaningful progress? Of course, the answer is, again, Python.

In particular, we base this case study on the excellent scikit-learn [Ped11] Python library. Scikit-learn provides easy to use and efficient tools for data analysis. Most importantly, it provides routines for many important Data Science concepts such as machine learning, cross validation, etc. In fact, this case study is inspired by the scikit-learn tutorial "Working With Text Data" which can be found at http://scikit-learn.org/stable/tutorial/text_analytics/working_with_text_data.html.

Following our theme of leveraging Python to quickly get to interesting Data Science problems, the students in case study three are encouraged to start their work based upon various examples provided in the scikit-learn library. In particular, the students leverage the exercise_02_sentiment.py files from the directories:

- doc/tutorial/text_analytics/skeletons/
- doc/tutorial/text_analytics/solutions/

One version of the file is merely a skeleton of a natural language processing example, while the other contains the full source code.

For DS501 there are two key learning goals for this case study. First, the students need to derive *features* from the raw text that they feel would be useful in predicting positive and negative sentiments. Second, they must make predictions by processing these features using a variety of supervised machine learning algorithms.

Feature Generation: Classically, rather than attempting to do machine learning on raw text, Data Science practitioners will first process the raw text to derive features for downstream processing. A detailed description of text feature generation is beyond the scope of the current text (the interested reader may see [Raj11], and references therein, for more details). However, Python and scikit-learn [Ped11] provide easy access to the exact functionality required by the students by way of the TfidVectorizer class which implements the term frequency–inverse document frequency (TF-IDF) statistic [Raj11]. For our purposes we merely observe that there are several parameters that the student can explore to get a feel for feature generation from raw text, including *min_df* and *max_df* parameters (which control thresholds on document frequencies) and ngram_range (which controls how many words are conglomerated into a single token). Experimenting with these parameters provide many important insights for feature generation from real world text data, not the least of which is that large values of ngram_range may take a long time to run.

Supervised Machine Learning: Now, given a collection of reviews, each represented by a set of features sometimes called *predictors*, one can imagine many interesting problems. For example, a classic problem in machine learning involves using a set of reviews which have appropriate labels (in this case positive or negative) to *predict* labels of other reviews which do not already have labels. This process is called *supervised* machine learning. The idea is that the labeled data is used to *supervise* the training of an algorithm which, after training, can attempt to compute labels just from the raw features. Again, supervised machine learning is a vast subject, and space does not allow us to treat the subject even at the more superficial level here (the interested read may see [Fri01], [Jam13], [Bis06], and references therein, for more details). However, we will note that scikit-learn provides functions and classes for many standard algorithms, allowing the students to become familiar with important machine learning and Data Science concepts, without being expected to have too many prerequisites. For example, scikit-learn provides access to classic and powerful algorithms such as K-nearest neighbors, support vector classifiers, and principal component analysis [Fri01], [Jam13], [Bis06].

Using such routines, several important learning objectives can be supported, such as error estimation, by way of techniques such as cross-validation and confusion matrices. In fact, one particularly effective learning experience revolved around the following challenge. Using their favorite technique the student is asked to

find a two dimensional plot of the data where the positive and negative reviews are separated. While easy to state, practitioners of natural language processing will recognize that actually solving the problem is exceptionally difficult, and the instructors admit that they are not in possession of an actual solution. For some students this may be the first time they have been presented with a problem they are expected to tackle for which their instructor *does not know the solution*. The student's ability to begin thinking about such open problems so early in their Data Science career is substantially supported by a language such as Python and the libraries it provides.

Case Study Four

The final case study, and in some sense the capstone of the class, revolves around the Yelp Dataset Challenge http://www.yelp.com/dataset_challenge. This case study involves a large data set with approximately 42,153 business, 252,898 users, and 1,125,458 reviews in Phoenix, Las Vegas, Madison, Waterloo and Edinburgh.

Again, building off of the previous case studies, the students are expected to process the data, generate statistics, process reviews using TfidVectorizer, etc. However, for this case study the students are also expected to process the data using MapReduce [Dea08]. As is well known in certain circles, MapReduce is a programming model (with various implementations) for distributed processing of large scale data sets. Distributed processing models, and MapReduce in particular, are essential elements of modern Data Science and we would have felt remiss if students in a class such as DS501 were not able to experience, at least at some level, the beauty and power of such methods.

Fortunately, and we fear that we are repeating ourselves, Python provides precisely the functionality we required. In particular, there are several MapReduce interfaces for Python, and the mrjob package [MrJ15] was chosen to support the students learning objectives. This package is especially useful in a classroom environment since it can be used locally on a single computer (for testing) and in a cluster environment. Accordingly, the students can learn about MapReduce with the need for access to large scale computing resources.

Introductory Data Science: Final Thoughts

Of course, Python is not the only choice for an Introductory Data Science course. For example, the scripting language R [RCT13] is also a popular choice which has also used successfully in the Data Science curriculum. In particular, R offers much, if not all, of the functionality mentioned above, including interfaces to MapReduce [Usu14]. Accordingly, the choice of language for such a class may be considered a matter of taste.

However, there is mounting evidence of Python's growing popularity within the Data Science community [Sin14] and the software development community at large [OGr14]. Perhaps, if we may be forgiven a small measure of Python bias, we will merely emphasize that Python's popularity cuts across many problem domains. For example, the authors are not aware of any customer relationship management applications, system administration tools, or web servers[1], to name just a handful of areas outside of statistical and data analysis, currently being developed in R, nor many other domains in which Python has made inroads. The fact that Python is as generally applicable as it is, while perhaps still being just as popular as R for Data Science, is a testament to its advantages.

MA542 Regression Analysis

Leaving aside introductory classes, we now make brief mention of Python's usefulness in more advanced classes. In particular, one of the authors recently taught a Regression Analysis class (using the text *Applied linear regression models* [Kut04]), for the first time, with all of the development in the class being Python focused. Regression Analysis is a more advanced class with a greater concentration of mathematically focused students who take the class. In addition, many students were first time Python users, with the majority of the exceptions being Data Science students who had taken DS501–"Introduction to Data Science" previously.

Just as in DS501, Numpy [Wal11], matplotlib [Hun07], and pandas [McK10] provided almost all of the functionality the students required for the learning objectives in the class. Also as in DS501, the instructor can use Python and its vast array of libraries to carefully control the difficulty and scope of assignments. In fact, one of the challenges in this class was that Python perhaps does *too good* of a job providing functionality to the students.

In particular, Python provides so many libraries that, for example, many of the computationally oriented homework questions are trivially answerable if the students look hard enough. Accordingly, as an instructor, one needs to be careful that the ground rules are set correctly so that the learning objectives are achieved. For example, if the learning objective is for the student to understand the details of a particular mathematical concept, say the *normal equations*, rather than just a numerical procedure, such as *linear regression* on a particular data set, then the expectations for the assignment need to be carefully delineated.

Accordingly, to maintain the integrity of the learning objectives, a tactic used by the authors was to carefully delineate what parts of the assignment are allowed to be Python "black boxes" and which parts must be hand coded. In addition, we require the students to hand in their Python code, even though the code itself is *not* graded. The learning objectives of the class are mathematical, and not programming. Accordingly, the quality of the implementations is not a focus. However, having access to the code allows the instructor to verify that the desired learning objectives are being met.

As one final note, one tactic that was quite successful was to encourage the students to check their hand coded results against those provided by any black box routine they are able to use. It was quite useful for the students in debugging their own implementations and understanding of the mathematical concepts. It was quite empowering for the students when their answers would exactly match those of the black box. They then appreciated that they understood, in a deep way, what the "professionals" were doing.

Student research projects and theses

Python has had an important part to play in the authors' research since 1997 [Paf99]. Currently, we perform research involving, and mentor students in, several topics revolving around semi-supervised and unsupervised machine learning applied to several

1. We would be remiss not to at least mention the quite beautiful R web application framework Shiny [Shi14]. However, we believe our point still stands.

different domains, with a focus on cyber-defense (see, for example, [Paf13]). Accordingly, one of our key goals is to support the training of the next generation of researchers in these domains. We will not burden the reader with the mathematical details of our research directions, but just observe that our work, and the work of our students, draws from a laundry list of ideas from mathematics, statistics, and Data Science, including convex optimization [Boy04], deep learning [Den14], graphical models [Lau96], and scientific visualization [War10].

For the current purpose, it is merely important to note that Python libraries are available that support *all of these subject areas*. For example, we have:

- Statistical modeling: Statsmodels [StM15]
- Convex optimization: cvxopt [Dah06], CVXPY [Dia14]
- Deep learning: Theano [Ber11]
- Graphical models: libpgm [Kar14], pgmpy [Pgm15]
- Scientific visualization: Mayavi [Ram11], Matplotlib [Hun07], Bokeh [Bok15], Seaborn [Was14]

Accordingly, students who are trained in classes such as DS501 and MA542 can leverage that training to get a running start on their research subjects. Perhaps this is the single biggest advantage of using a language such as Python from the earliest stages of Data Science education. In addition to being easy to learn [Lut13], and providing access to many libraries that support Data Science education, Python provides ready access to a broad swath of cutting edge Data Science research.

We use all of these libraries in our work, where we are especially interested in large scale robust principle component analysis [Can11], [Paf13] and non-linear dimension reduction problems [Lee07]. These problem domains are mathematically subtle, computationally intensive, and lead to, in the authors' opinion, rather intriguing visualization problems, which are also supported by Python through libraries such as Mayavi, as shown in the figure below.

Beyond the mathematical research that Python supports, there are a vast array of computational resources that are at the fingertips of those well versed in Python. For example, our research group is interested in developing algorithms for modern distributed supercomputers that leverage GPUs to accelerate computations. Again, Python displays its usefulness with the pycuda [Klo12] and mpi4py [Dal08] libraries.

As one can see, Python is an effective tool for cutting edge Data Science research. Of course, there are many such tools, and often the specific choice of language for Data Science research is a matter of taste. However, we would respectfully submit that few languages have the broad range of support for Data Science research that Python provides.

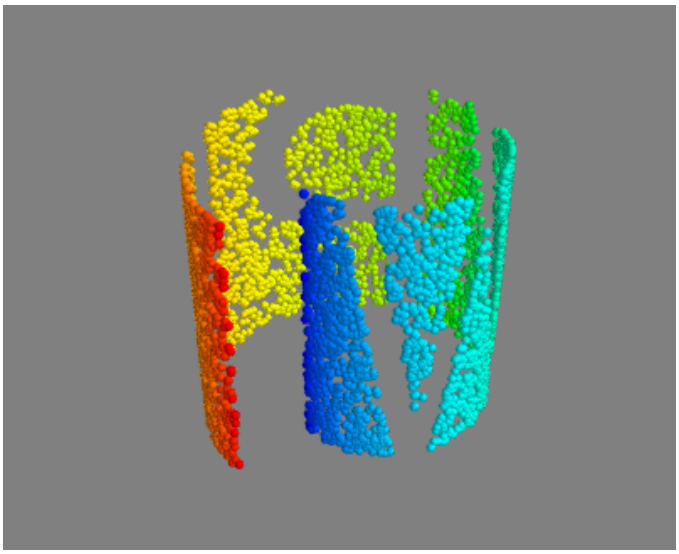


Fig. 1: *An example of a 3D visualization of a manifold using Mayavi [Ram11]. In our work we attempt to detect the non-linear dependencies in such data, even when the data is noisy and unevenly distributed. In this synthetic example we see data which is intrinsically two-dimensional (since it is a flat surface) embedded in a three-dimensional space. The two-dimensional structure is non-trivial to detect based upon the non-linear nature of the data, noise, and regions with no data points.*

Conclusion

We have discussed how Python can be used throughout the entire life cycle of a graduate program in Data Science. Python is easy to learn and use, but it also provides access to a vast array of libraries for cutting edge Data Science research. In particular, IPython notebooks, scikit-learn, NumPy, SciPy, and pandas can be used to support many aspects of the Data Science education. These libraries allow instructors to focus on desired learning objectives, while leaving many of the less important details to the libraries. Having access to such libraries allow interesting problems to be addressed early in the educational process and the experience gained with such Python libraries supports the student's own software development, analysis, and research throughout their academic career and beyond.

Acknowledgments

We wish to gratefully acknowledge several people without whom this monograph would not have been possible. In particular, the authors are deeply grateful to the other members of Data Science Steering Committee at WPI:

- Prof. Elke Angelika Rundensteiner (Director of Data Science)
- Prof. Mohamed Eltabakh
- Prof. Eleanor T. Loiacono
- Prof. Joseph D. Petruccelli
- Prof. Carolina Ruiz
- Prof. Diane M. Strong
- Prof. Andrew C. Trapp
- Prof. Domokos Vermes
- Prof. Jian Zou

without whose tireless efforts the WPI Data Science program would not be what it is today.

References

[Beh11] Stefan Behnel, Robert Bradshaw, Craig Citro, Lisandro Dalcin, Dag Sverre Seljebotn and Kurt Smith. *Cython: The Best of Both Worlds*, Computing in Science and Engineering, 13, 31-39 (2011), DOI:10.1109/MCSE.2010.118

[Ber11] Bergstra, James, et al. *Theano: Deep learning on gpus with python*. NIPS 2011, BigLearning Workshop, Granada, Spain. 2011. http://deeplearning.net/software/theano/ [Online; accessed 2015-06-08].

[Bis06] Bishop, Christopher M. *Pattern recognition and machine learning*. Springer, 2006.

[Bok15] *Bokeh* (2015), http://bokeh.pydata.org/en/latest/ [Online; accessed 2015-06-17].

[Boy04] Boyd, Stephen, and Lieven Vandenberghe. *Convex optimization*. Cambridge university press, 2004.

[Can11] Candès, Emmanuel J., Li, Xiaodong, Ma, Yi, and Wright, John. *Robust principal component analysis?*. Journal of the ACM (JACM) 58.3 (2011): 11.

[Dah06] Dahl, Joachin, and Lieven Vandenberghe. *Cvxopt: A python package for convex optimization.* Proc. eur. conf. op. res. 2006. http://cvxopt.org [Online; accessed 2015-06-08].

[Dal08] Dalcín, Lisandro, Paz, Rodrigo, Storti, Mario, and D'Elía, Jorge (2008). *MPI for Python: Performance improvements and MPI-2 extensions*. Journal of Parallel and Distributed Computing, 68(5), 655-662.

[Dea08] Dean, Jeffrey, and Sanjay Ghemawat. *MapReduce: simplified data processing on large clusters*. Communications of the ACM 51.1 (2008): 107-113.

[Den14] Deng, Li, and Dong Yu. *Deep learning: methods and applications*. Foundations and Trends in Signal Processing 7.3–4 (2014): 197-387.

[Dia14] Diamond, Steven, Eric Chu, and Stephen Boyd. *CVXPY: A Python-embedded modeling language for convex optimization*, version 0.2." (2014). http://cvxpy.org [Online; accessed 2015-06-08].

[Fri01] Friedman, Jerome, Trevor Hastie, and Robert Tibshirani. *The elements of statistical learning*. Vol. 1. Springer, Berlin: Springer series in statistics, 2001.

[Jam13] James, Gareth, et al. *An introduction to statistical learning*. New York: springer, 2013.

[Hun07] John D. Hunter. *Matplotlib: A 2D Graphics Environment*, Computing in Science & Engineering, 9, 90-95 (2007), DOI:10.1109/MCSE.2007.55

[Kar14] Karkera, Kiran R. *Building Probabilistic Graphical Models with Python*. Packt Publishing Ltd, 2014.

[Klo12] Andreas Klöckner, Nicolas Pinto, Yunsup Lee, Bryan Catanzaro, Paul Ivanov, Ahmed Fasih, *PyCUDA and PyOpenCL: A scripting-based approach to GPU run-time code generation*, Parallel Computing, Volume 38, Issue 3, March 2012, Pages 157-174.

[Kut04] Kutner, Michael H., Chris Nachtsheim, and John Neter. *Applied linear regression models*. McGraw-Hill/Irwin, 2004.

[Lau96] Lauritzen, Steffen L. *Graphical models*. Oxford University Press, 1996.

[Lee07] Lee, John A., and Michel Verleysen. *Nonlinear dimensionality reduction*. Springer Science & Business Media, 2007.

[Lut13] Lutz, Mark. *Programming python*. 5th edition, O'Reilly Media, Inc., 2010.

[McK10] McKinney, Wes. *Data Structures for Statistical Computing in Python*, Proceedings of the 9th Python in Science Conference, 51-56 (2010)

[McK12] McKinney, Wes. *Python for data analysis: Data wrangling with Pandas, NumPy, and IPython* .O'Reilly Media, Inc., 2012.

[PMe15] *Merge, join, and concatenate* (2015), http://pandas.pydata.org/pandas-docs/stable/merging.html [Online; accessed 2015-06-08].

[Mil11] K. Jarrod Millman and Michael Aivazis. *Python for Scientists and Engineers, Computing in Science & Engineering*, 13, 9-12 (2011), DOI:10.1109/MCSE.2011.36

[Mov15] *MovieLens* (2015), http://grouplens.org/datasets/movielens/ [Online; accessed 2015-06-08].

[MrJ15] *mrjob* (2015), https://pythonhosted.org/mrjob/ [Online; accessed 2015-06-08].

[Mye07] Myers, Christopher R., and James P. Sethna. *Python for education: Computational methods for nonlinear systems.* Computing in Science & Engineering 9.3 (2007): 75-79.

[OGr14] O'Grady, Stephen. *The RedMonk Programming Language Rankings: January 2014* (2014), http://redmonk.com/sogrady/2014/01/22/language-rankings-1-14/ [Online; accessed 2015-06-08].

[Oli01] Jones, Erid, Oliphant, Travis, Peterson, Pearu, et al. *SciPy: Open Source Scientific Tools for Python*, 2001-, http://www.scipy.org/ [Online; accessed 2015-05-31].

[Oli07] Travis E. Oliphant. *Python for Scientific Computing*, Computing in Science & Engineering, 9, 10-20 (2007), DOI:10.1109/MCSE.2007.58

[Paf99] Paffenroth, Randy C. *VBM and MCCC: Packages for objected oriented visualization and computation of bifurcation manifolds.* Object Oriented Methods for Interoperable Scientific and Engineering Computing: Proceedings of the 1998 SIAM Workshop. Vol. 99. SIAM, 1999.

[Paf13] Paffenroth, Randy, Du Toit, Philip, Nong, Ryan, Scharf, Louis, Jayasumana, Anura. P., and Bandara, Vidarshana. (2013). *Space-time signal processing for distributed pattern detection in sensor networks*. Selected Topics in Signal Processing, IEEE Journal of, 7(1), 38-49. Chicago

[Ped11] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, Édouard Duchesnay. Scikit-learn: Machine Learning in Python, Journal of Machine Learning Research, 12, 2825-2830 (2011)

[Per07] Fernando Pérez and Brian E. Granger. IPython: A System for Interactive Scientific Computing, Computing in Science & Engineering, 9, 21-29 (2007), DOI:10.1109/MCSE.2007.53

[Pgm15] *Python Library for Probabilistic Graphical Models (pgmpy)* (2015), https://github.com/pgmpy/pgmpy [Online; accessed 2015-06-24].

[Ptw15] *Python Twitter* (2015), https://code.google.com/p/python-twitter/ [Online; accessed 2015-06-08].

[Raj11] Rajaraman, Anand and Jeffrey David Ullman. *Data Mining*, Mining of Massive Datasets. 1st ed. Cambridge: Cambridge University Press, 2011. pp. 1-17. Cambridge Books Online. http://dx.doi.org/10.1017/CBO9781139058452.002 [Online; accessed 2015-06-08].

[Ram11] Ramachandran, Prabhu and Varoquaux, Gael, *Mayavi: 3D Visualization of Scientific Data* IEEE Computing in Science & Engineering, 13 (2), pp. 40-51 (2011)

[RCT13] R Core Team (2013). R: A language and environment for statistical computing. R Foundation for Statistical Computing, Vienna, Austria. http://www.R-project.org/.

[Rus13] Russell, Matthew A. *Mining the Social Web: Data Mining Facebook, Twitter, LinkedIn, Google+, GitHub, and More*. O'Reilly Media, Inc., 2013.

[Shi14] RStudio, Inc. *shiny: Easy web applications in R* (2014), http://shiny.rstudio.com [Online; accessed 2015-06-08].

[Sin14] Singh, Harpreet. *Is Python Becoming the King of the Data Science Forest?* (2014), http://www.experfy.com/blog/python-data-science/ [Online; accessed 2015-06-08].

[Sta00] Stajano, Frank. *Python in education: Raising a generation of native speakers.* Proceedings of 8th International Python Conference. 2000.

[StM15] *Statsmodels* (2015), http://statsmodels.sourceforge.net/ [Online; accessed 2015-06-17].

[Twi15] *The Streaming APIs Overview* (2015), https://dev.twitter.com/streaming/overview [Online; accessed 2015-06-08].

[Usu14] Usuelli, Michele. *An Example of MapReduce with rmr2* (2014), http://www.milanor.net/blog/?p=853 [Online; accessed 2015-06-08].

[Wal11] 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), DOI:10.1109/MCSE.2011.37

[War10] Ward, Matthew, Georges Grinstein, and Daniel Keim. *Interactive Data Visualization: Foundations, Techniques, and Applications*, AK Peters, Ltd., Natick, MA (2010).

[Was14] Waskom, Michael et al.. (2014). *seaborn: v0.5.0* (November 2014). Zenodo. 10.5281/zenodo.12710

Relation: The Missing Container

Scott James‡*, James Larkin‡

Abstract

The humble mathematical *relation*[1], a fundamental (if implicit) component in computational algorithms, is conspicuously absent in most standard container collections, including Python's. In this paper, we present the basics of a relation container, and why you might use it instead of other methods. The concept is simple to implement and easy to use. We will walk through with code examples using our implementation of a relation (https://pypi.python.org/pypi/relate)

Background: It's the Little Things

In our work in surface and aviation traffic simulation we deal with many moving pieces, terabytes of streaming information. Managing this much information pieces requires, unsurprisingly, some significant computational machinery: clusters of multiprocessors; different interworking database topologies: HDF5, NoSQL and SQL; compiled code, scripted code; COTS tools, commercial and open source code libraries. For the Python components of our work, we are fortunate to have data crunching libraries: numpy, pandas etc... However, we kept finding that, despite this wealth of machinery, we would get caught up on the little things.

There may be thousands of flights in the air at any one time, but there are far fewer *types* of aircraft. There may be millions of vehicles on the road, but only a handful of vehicle categories. Whereas we could place these mini-databases into our data crunching tools as auxiliary tables, we didn't. It didn't make sense to perform a table merge with streaming data when we could do a quick lookup, on-the-fly, when we needed to. We didn't want to create a table with ten rows and two columns when we could easily put that information into a dictionary, or a list. We didn't want to implement our transient, sparse table with a graph database or create tables with an 'other' column which we would then have to parse anyhow. And besides the traffic specific information, there were all those other pesky details: file tags, user aliases, color maps.

Instead we cobbled together our mini-databases with what we had within easy mental reach: lists, sets and dictionaries. And when we needed to do a search, or invert keys/values, or assure uniqueness of mappings, we would create a loop, a list comprehension or a helper class.

* *Corresponding author: scott.james@noblis.org*
‡ *Noblis*

After some time it occurred to us that what we were really doing with our less-than-big data was reinventing a mathematical relation, ... over and over again. Once we realized that, we replaced the bookkeeping code managing our mini-databases with relation instances. This resulted in a variety of good things: reduced coding overhead, increased clarity of purpose and, oddly, improved computational efficiency.

What is a relation and what is it good for?

A relation a simply a pairing of elements of one set, the *domain*, with another, the *range*. Rephrasing more formally, a relation is a collection of tuples (x,y) where x is in the domain and y is in the range. A relation, implemented as code, can perform a variety of common tasks:

- **Inversion**: quickly find the values(range) associated with a key(domain)
- **Partitioning**: group values into unique buckets
- **Aliasing**: maintain a unique pairing between keys and values
- **Tagging**: associate two sets in an arbitrary manner

These roughly correspond to the four *cardinalities* of a relation:

- **Many-to-one (M:1)**: a function, each range value having possibly multiple values in the domain
- **One-to-many (1:M)**: a categorization, where each element in the domain is associated with a unique group of values in the range
- **One-to-one (1:1)**: an isomorphism, where each element in the domain is uniquely identified with a single range value
- **Many-to-many (M:N)**: an unrestricted pairing of domain and range

What is it not good for?

The relation, at least as we have implemented it, is a chisel, not a jack-hammer. It is meant for the less-than-big data not the actually-big data. When computational data is well-structured, vectorized or large enough to be concerned about storage, we use existing computational and relational libraries. A relation, by contrast, is useful when the data is loosely structured, transient, and in no real danger of overloading memory.

The API

Using a relation should be easy, as easy as using any fundamental container. It should involve as little programming friction as possible. It should feel natural and familiar. To accomplish these

Method	Comment
`__init__`	establish the cardinality and ordering of a Relation
`__setitem`	assign a range element to a domain element
`__getitem`	retrieve range element(s) for a domain element
`__delitem`	remove a domain element and all associated range pairings. If the range element has no remaining pairings, delete it.
`extend`	combine two Relation objects
`values`	return the domain
`keys`	returns list of domains
`__invert__`	swap domain and range

TABLE 1

goals, we created our Relation class by inheriting and extending *MutableMapping*:

In essence, it will look and feel like a dictionary, but with some twists.

Example 1 (Many-to-many)

For example, suppose we need to map qualitative weather conditions to dates:

```
weather = Relation()
weather['2011-7-23']='high-wind'
weather['2011-7-24']='low-rain'
weather['2011-7-25']='low-rain'
weather['2011-7-25']='low-wind'
```

Note that in the last statement the assignment operator performs an append not an overwrite. So:

```
weather['2014-7-25']
```

Produces a *set* of values:

```
{'low-rain','low-wind'}
```

Relation also provides an inverse:

```
(~weather)['low-rain']
```

Also producing a set of values:

```
{'2014-7-25','2014-7-24'}
```

For our work, other many-to-many relations include:

- Flight numbers and airports
- Auto makers and vehicle classes
- Neighboring planes (or autos) at an instant of time

Cardinality

Relations look like a dictionary but also provide the ability to

1) Assign many-to-many values
2) Invert the mapping directly

Relations become even more valuable when we have the ability to enforce the degree of relationship, i.e. cardinality. As mentioned, there are four cardinalities used in the relation object class:

Many-to-one assignment is already supported by Python's built-in dictionary (minus the inversion); however, the remainder of the cardinalities are not[2].

Relationship	Shortcut	Pseudonyms
many-to-one	M:1	function, mapping, assignment
one-to-many	1:M	partition, category
one-to-one	1:1	aliasing, isomorphism
many-to-many	M:N	general

TABLE 2

Example 2 (One-to-One)

```
airport = Relation(cardinality='1:1')
airport['ATL'] = 'Hartsfield-Jackson Atlanta International'
airport['KORD'] = 'Chicago O'Hare International'
```

When the relation is forced to be 1:1, the results are no longer sets:

```
airport['ATL']
> 'Hartsfield-Jackson Atlanta International'
```

And assignments overwrite *both* the domain and the range:

```
# use the full four-letter international code ...
# not the US 3-letter code
airport['KATL'] = 'Hartsfield-Jackson Atlanta International'
airport['ATL']
> KeyError: 'ATL'
```

Note that, similar to a dictionary silently overwriting a key-value pair, a 1:1 relation silently overwrites a value-key pair, and in this case, removes the stranded key. Also worth noting, for cardinalities M:1 and 1:1, a dictionary literal can also serve as syntactic sugar for an initializer:

```
airport = Isomorphism(
    {'KATL':'Hartsfield-Jackson Atlanta International',
    'KORD':'Chicago O'Hare International'})
airport['KATL']
> 'Hartsfield-Jackson Atlanta International'
```

For our work, other 1:1 mappings include:

- User names and company id
- Automobile manufacturers and their abbreviations
- Color codes and their representations in various simulation tools (using a chain of 1:1 containers)

Comparing Relation Implementations

The relation container is fast, as fast as a dictionary. It should be; it is implemented by two dictionaries: one for each mapping direction. However, there are other ways to implement many-to-many relations. In this section we compare the relation against two other implementations: a Pandas data frame and a NetworkX graph.

Our test data will consist of 200 two-digit alphanumeric values (domain) and 1 million numeric values (range) for a total of approximately 10 million unique entries. We describe the implementations of the lookups and then compare speeds.

Data Frame

To implement a M:N relationship using a data frame, we create a two-column table:

A forward search can performed as follows:

```
df['range']==887267]['domain']
```

And a reverse search as:

```
df[df['domain']=='YR']['range']
```

Each of these searches can be accelerated by indexing.

Domain	Range
UF	423423
OP	3242
FD	887267
YR	343
...	...

TABLE 3

Method	Forward (ms)	Reverse (ms)
Pandas	7.34e2	7.94e1
Pandas (indexed)	1.97e2	7.81e-2
Graph	9.47e0	6.84e-4
Relate	3.76e-4	4.58e-4

TABLE 4

NetworkX

To implement an M:N relation using a NetworkX Graph we use a bipartite graph, that is, a graph connecting two disjoint sets, creating the relations by linking the nodes from one set (domain) to another set (range)

Both forward and reverse searches are performed in the same manner:

```
# forward, using domain nodes
G.neighbors('YR')
# reverse, using range nodes
G.neighbors(887267)
```

Timings

We collect timings using the Python's `timeit` function:

In all cases, Relate is faster, most significantly when searching on strings as opposed to numeric values. Of course, data frames and graphs have many more features than a relation. Also, the two-dictionary Relation implementation is cheating: it precomputed the only two searches it was built to handle; moreover, it did so at a cost of doubling the memory footprint. But this is precisely the use-case for which the relation was created: space at non-critical levels but economy of code and code performance crucial.

Sparse Matrix

One other implementation worth mentioning is a sparse matrix. Viewing the nonzero elements of a sparse matrix as a connection between the row (domain) and column (range) indices also produces an M:N relationship. The power of the sparse matrix is in its suitability to large-scale numerical computations. The relation container proposed, however, is designed to match general datatypes, including non-numerical. Providing a direct comparison between the two is thus somewhat difficult as the two are used for different purposes.

More Examples

The relation object is a basic concept, and as such useful in limitless contexts. A few more examples are worth mentioning.

Tags (Many-to-Many)

Over the last decade, we've seen *tags* invade our previously hierarchical organized data. Tags are now ubiquitous, attached to our: photos, files, URL bookmarks, to-do items etc ...

Tags are also exactly a many-to-many relationship:

```
files = Relation()

files['radar-2011-7-23.png'] = 'image'
files['radar-2011-7-23.png'] = 'KATL'
files['departure-procedures.doc'] = 'KATL'
files['departure-procedures.doc'] = 2015

#find the files associated with Atlanta
(~files)['KATL']
> {'radar-2011-7-23.png','departure-procedures.doc'}

# find the attributes for particular file
files['departure-procedures.doc']
> {2015,'KATL'}
```

We tag our simulation products to allow flexible retrieval and searching. With an in-code tagging scheme we can automatically attach tags at the file system level and then query these tags with both in-code and operating system level tools.

Taxonomies (One-to-Many)

We mentioned earlier that the 1:M relation is a partition, a way to categorize objects into groups. Nesting 1:M relations creates a backward-searchable taxonomy. An example in our work are en-route air traffic sectors, the nested polyhedrons through which aircraft fly:

```
sectors=Relation(cardinality='1:M')
sectors['ZNY'] = 'ZNY010'
sectors['ZNY'] = 'ZNY034'
sectors['ZNY010'] = 'ZNY010-B'
sectors['ZNY010'] = 'ZNY010-2'
sectors['ZNY034'] = 'ZNY034-B'
sectors['ZNY034'] = 'ZNY034-11'

(~sectors)['ZNY034-B']
> 'ZNY034'

(~sectors)[(~sectors)['ZNY034-B']]
> 'ZNY'
```

Using a taxonomy of sectors as above allows us to quickly access aggregate information at different granularities as the flight progresses.

When to Use What for What

Modern high-level computing languages provide us with a robust set of containers. We feel, of course, that a relation container is a valuable addition but, we also feel one should use the most economical container for the task. Asking questions about the type of data being stored and the relationship between an element and its attributes is crucial, even for the less-than-big data:

Choosing the best matching structure for your data set doesn't just help with the code, it helps with the intent, providing the next programmer touching the code with your vision of the structure, and also some safety belts in case they didn't see it the first time.

Conclusion

The relation object provides an easy-to-use invertible mapping structure supporting all four relationship cardinalities: 1:1, 1:M, M:1 and M:N. Using the relation library can simplify your

Content	Structure
unordered set of unique objects	set
ordered set of non-unique objects	list
ordered set of unique objects	OrderedDict
unidirectional mapping	dictionary
bidirectional mapping	relation
mapping with restricted cardinalities	relation
multiple, fixed attributes per element	data frame/table
variant attributes per element	relation

TABLE 5

code and eliminate the need for repeated, ad hoc patterns when managing your less-than-big working data structures.

One of the best things about the relation data container is its ease of implementation within Python. For a simple, yet complete example, see our implementation at https://pypi.python.org/pypi/relate.

1. http://www.purplemath.com/modules/fcns.htm

2. For 1:1 mapping, however we also recommend the excellent bidict package https://bidict.readthedocs.org/en/master/intro.html#intro

Testing Generative Models of Online Collaboration with BigBang

Sebastian Benthall[‡*]

https://www.youtube.com/watch?v=AQFS_ES7rT0

Abstract—We introduce BigBang, a new Python toolkit for analyzing online collaborative communities such as those that build open source software. Mailing lists serve as critical communications infrastructure for many communities, including several of the open source software development communities that build scientific Python packages. BigBang provides tools for analyzing mailing lists. As a demonstration, in this paper we test a generative model of network growth on collaborative communities. We derive social networks from archival mailing list history and test the Barabási-Alpert model against this data. We find the model does not fit the data, but that mailing list social networks share statistical regularities. This suggests room for a new generative model of network formation in the open collaborative setting.

Index Terms—mailing lists, network analysis, assortativity, power law distributions, collaboration

Introduction

Open source software communites such as those that produce many scientific Python packages are a critical part of contemporary scientific organization. A distinguishing feature of these communities is their heavy use of Internet-based infrastructure, such as mailing lists, version control systems, and issue trackers, for managing communications and organizing work on distributed teams. This data is often deliberately publicly accessible as open source best practices include the "conspicuous use of archives" [Fogel]. The availability of these digital records are also an excellent resource for the researcher interested in sociotechnical organization and collaboration within science.

This paper introduces BigBang, a Python project whose purpose is the collection, preprocessing, and analysis of data from open collaborative communities. Built for the use case of studying the Scientific Python communities in particular, it generalizes to other communities and supports fruitful comparisons between them.

To demonstrate the potential of this approach, this paper will explore the structure of mailing list discussions in the context of open collaborative projects. We extract social network data from the archives of public mailing lists and test the plausibility that these graphs were generated by Barabási-Alpert network model. We find that of the mailing lists we've analyzed, none exhibit two features of Barabási-Alpert networks: power law degree distribution and zero degree assortativity. Instead the data indicates that these networks have a log-normal degree distribution and have negative degree disassortivity. This result suggests the possibility of future work of scientifically developing a generative model of collaboration.

* Corresponding author: sb@ischool.berkeley.edu
‡ UC Berkeley School of Information

BigBang Overview

Launched in 2014, BigBang is a software project that aims to provide researchers a complete toolkit for the scientific analysis of open online collaborative communities. Though applicable to many domains, research into online collaboration has special relevance to practicioners of computationally intensive open science. Through it scientific programming communities such as Scientific Python can achieve a quantitative understanding of their own work and innovation process.

Thorough study of these kinds of communities requires the collection and rationalization of many heterogenous and high-dimensional data sources, including but not limited to mailing lists, version control systems such as Git, and issue trackers such as GitHub and Bugzilla.

This data is complex in that it has many dimensions that afford very different kinds of analysis:

- **Time.** All data from online collaboration infrastructure is timestamped, affording use of time series methods.
- **Text.** Email message bodies, issue contents, and commit messages in version control are all text data suitable for study with natural language processing techniques.
- **Social network.** Participants in the project are individuals linked by relational ties of communication. Hence these data afford study through social network analysis techniques.
- **Software static analysis.** Source code in version control is complex data containing the definition of many interelated variables, functions, classes, and modules. Static analysis and compilation techniques from computer science can be used to study these entities within the software itself.

The richness and granularity of the data from open source software communities and other open on-line collaborative projects promise the answers to many research questions about software engineering, innovation, social organization, and more. The catch is that with data that is so multifaceted, preprocessing the data is an engineering-intensive endeavor.

The versatility and scope of open source Scientific Python packages makes building a generic research infrastructure for analyzing these communities a possibility within reach. Architecturally, BigBang is a Python package that includes Scientific Python libraries for time series analysis, natural language processing, network analysis, and software analysis as dependencies.

BigBang includes methods for collecting research data from sources on the web about the activity of open collaborative communities. At the time of this writing, BigBang supports data collection from Mailman, the mailing list service, as well as other *.mbox* formatted email archives. It also supports data collection from Git repositories. Future versions will have methods for collecting data from issue trackers.

The BigBang repository contains an *examples* directory of Jupyter notebooks demonstrating its functionality and exploring lines of research inquiry. Researchers can contribute to the project by submitting Jupyter notebooks to the repository for review through the GitHub Pull Request system. Source code that performs preprocessing that is usable by multiple computational experiments is in a separate source code directory that can be imported as a Python module.

In the context of Scientific Python, BigBang is deliberately recursive. It is a Python project that depends on many other scientific Python projects. It is designed to study, among other things, dependencies and interactions between the Scientific Python technologies and communities. Our goal is for BigBang to provide a new means for these communities to engage in scientific self-management.

Testing Generative Models of Online Collaboration

As a demonstration of BigBang's capabilities, in this paper we will test a well known generative model of network formation against social network data derived from public mailing list discussions. A generative model is a formal model that describes a process through which data is generated. A principle benefit of a formal generative model is that the statistical properties of data it generates can be compared with the statistical properties of empirical data. Such comparisons are one way to get empirical purchase on the mechanism behind even purely observational data. Discovering a concise generative model that fits data from online collaboration would give us insight into the mechanism of collaboration itself.

In this paper, we will test one well known generative model of network data, the Barabási-Alpert model. This model describes a process by which new nodes, as they join a network, form edges with other nodes with probability proportional to their degree. This process is called *preferential attachment*. Very roughly speaking, in social networks preferential attachment is suggestive of a network dominated by attachments to a small number of luminaries. In its basic form, this model generates networks with two notable statistical properties:

- The degree distributions of Barabási-Alpert networks are *scale-free*, meaning that the fraction of nodes of degree k falls asymptotically according to a power law distribution. $P(k) \sim k^{-\gamma}$ for some positive γ.
- The correlation between the degrees of adjacent nodes converges to zero (from below) as the network grows.

We discover in our empirical data that neither of these properties hold for the social networks fo public mailing list discussions. This suggests that preferential attachment is not a mechanism that dominates the social interactions on the collaborative projects represented in our data. On the contrary, the statistical properties of public mailing list discussions suggest that participation is more widely distributed than in many other social networks, and that interaction with new participations is a priority.

Preferential attachment model

An early result in the study of complex networks was the observation that many networks existing in nature exhibit a scale-free degree distribution. [BarabásiAlbert] This means that the tail of the distribution of the number of edges of each node in the network (the node's *degree*) converges to a power law function:

$$y = ax^k$$

(Scale-free refers to the scale invariance of the power law distribution.)

The prevalence of scale-free networks in nature has raised the question of what generative processes produce networks with this property. What was at one point the most well-known random graph model, the Erdős-Rénri model, produces networks with binomial degree distribution. Barabási and Alpert [BarabásiAlbert] have proposed a widely cited and studied model of network generation that produces graphs with scale-free degree distribution.

The attractiveness of the Barabási-Alpert model is due in part to its being a generative model that describes a process for creating data of an observed distribution, as opposed to being simply a description of the distribution itself. This gives the Barabási-Alpert model explanatory power.

In particular, the Barabási-Alpert model attributes the scale-free distribution of node degree to a *preferential attachment* mechanism, parameterized by m_0 and m. The network is formed by beginning with a small number m_0 of nodes and adding new nodes, connecting each new node to $m < m_0$ nodes, where the probability of connecting to node i is proportional to the prior degree of that node, k_i.

$$P(k_i) = \frac{k_i}{\sum_j k_j}$$

Here, $\sum_j k_j$ is the sum of all degrees of all nodes in the graph. The parameter m is fixed across all iterations. [AlbertBarabási]

The Barabási-Alpert model is favored for its simplicity, its intuitively clear mechanism of preferential attachment, and for its analytic tractability. Intuitively, a social process driven by preferential attachment is one in which "the rich get richer". Consider the social graph from an on-line social network such as Facebook or Twitter. New entrants to the network will 'friend' or 'follow' existing nodes. If they preferentially attach, the will be much more likely to connect to celebrities who already dominate the network than to new entrants such as themselves. The most highly connected participants will likely owe their position in the network to their seniority. Studies have supported the role of a preferential attachment mechanism in social network formation [Zhou2011], [Tinatti2012]. It is an empirical question whether the preferential attachment mechanism explains the data from collaborative communities such as those that develop Scientific Python packages.

Power law or log-normal?

An implicit challenge to the Barabási-Alpert model comes from [Clauset2007], who argue that many conventionally accepted techniques for fitting power law distributions to empirical data are biased and unsound. Specifically, they critique the common method of plotting the histogram of the data on a log-log axis and testing for linearity by perfoming a least-squares linear regression, and variations of this. They propose an alternative Bayesian technique for testing power law distributions. By computing the likelihood of the data being generated by a power law distribution and comparing it with the likelihood of it being generated by other heavy-tail distributions, such as the log-normal distribution, they provide a statistically sound basis for model comparison.

The Clauset et al. method considers only the tail of the data, picking a cutoff value x_{min} below which data are ignored. They argue that picking this value is of critical importance: to pick too high an x_{min} is to fit a power law to non-power law data. to pick too low a value is to throw out legitimate data, which can lead to bias. They propose selecting the x_{min} that minimizes the Kolmogorov-Smirnov distance between the best fitting power law distribution and the empirical data above the mininum.

We refer the reader to [Clauset2007] for the in-depth defense of this method as an alternative to those based on testing for linearity on a log-log scale. Besides its statistical soundess, an advantage of this method is that it has been implemented in Python in the *powerlaw* package by [Alstott2014], which is what is used for the computational results below.

From a Bayesian perspective, the ratio of likelihoods represents how much one should update one's beliefs based on observation of data. In this case, the computed likelihood ratio of the data being generated by a power law over a log-normal distribution would be interpreted as how much the data should persuade that it came from a power law distribution independent of one's prior untested belief.

This leaves open the question of the prior probability of a distribution being generated by a power law producing process, or a log-normal producing process. [Mitzenmacher2003] surveys a century of scientific disagreement over the prevalance of each distribution across many disciplines. Different processes are expected to produce different distributions.

Processes through which "the rich get richer" systematically, such as the preferential attachment process described above, will produce power law distributed data.

Log-normal distributions are produced by what [Mitzenmacher2003] calls *multiplicative processes*. A multiplicative process occurs when independent random variables are multiplied together. Contrast this with the preferential attachment process, where the possibility of attachement is distinctly not independent of prior conditions. When a series of independent and identically distributed variables is multiplied together, the product's distribution converges on a log-normal distribution by the Central Limit Theorem.

Mitzenmacher argues that subtle variations in generative processes can turn their results one way or another. Ideally one can look more deeply at the structure of data, not just its distribution, to determine the process behind a heavy-tailed data set. Despite this difficulty, the statistical consequences of different processes will become more apparent asymptotically as more data is generated.

In summary, a process of network growth according to which degree is the result of a independent multiplicative process will assymptotically produce a log-normal distribution. A process of network growth driven by non-independent preferential attachment will approach a power law degree distribution. A test of the log likelihood of the best fit of either distribution on an empirical data set provides empirical support for the data's being produced by one process or the other.

Degree assortativity

Another graph theory concept that we will use in our analysis of collaborative mailing lists is *degree assortativity*. Degree assortativity is the correlation between degrees of adjacent nodes in the network. In the context of social networks, it is a measure of a special case of *homophily*, the tendency of people to be connected to others who are similar to them. Degree assortativity means that the most connected members of the network are connected with each other.

Following the mathematical definition of [Newman2003], the degree assortativity coefficient is

$$r = \frac{\sum_{jk} jk(e_{jk} - q_j q_k))}{\sigma_q^2}$$

In the above formula, e_{jk} is the fraction of edges that connect vertices of degree $j+1$ and $k+1$, i.e. the degrees of the connected vertices not including the connecting edge itself. [Newman2003] calls this *excess degree*. The value q_k is the distribution of excess degree.

$$q_k = \sum_j e_{jk}$$

The value σ_q is the standard deviation of q_k.

[Newman2002] studied degree assortativity in complex networks and introduced an intriguing hypothesis. Observed social networks, such as those of academic coauthorship networks and business director associations, exhibit positive degree assortativity. Technical and biological networks, such as connections between autonomous systems on the Internet, protein interactions, and neural networks, exhibit negative degree assortativity, or *disassortivity*. Our own speculative interpretation is that the organization of technical and biological networks evolves for a functional purpose facilitated better by having highly connected hubs distributed widely, whereas many social networks are organized more according to the self-interest or homophilic tendencies of the participants.

[Noldus2015] reviews the extensive scholarship on assortativity in networks since Newman's work in 2002. They note that Barabási-Alpert networks are only slightly disassortive, converging on zero assortativity as the number of nodes increases. [Noldus2015] also surveys work such as [Newman2003] and [Foster2009] that define and analyze directed degree assortativity. In directed variations, degree assortativity is computed as above except using either the in-degree or out-degree of the source and targets nodes. In our empirical work below, we report directed assortativity in its in/in and out/out variations. We have observed little difference between these and the computed values for the in/out and out/in variations in our data, though there are theoretical graph structures for which these values can vary greatly.

According to the survey by [Noldus2015], assortivity in weighted networks is not well explored either theoretically or empirically. The weighted assortativity of a a network is the correlation between the weighted degree of its adjacent nodes, where weighted degree is the sum of the weights of all edges of a node. Directed weighted assortativity is computed from

weighted in- and out-degrees. [Networkx] provides functions for computing these values on networks. We will compare weighted and unweighted directed assortivity in empirical networks below.

Methods

We collected archival data of 13 mailing lists from open collaborative communities. From these data we derived an *interaction graph* of who replied to whom. We then computed the weighted and unweighted degree assortativity of these networks. We also used the Alstott package to test the degree distribution of these networks using the Clauset method.

Email data collection

BigBang supports collection of email data. It can do this either by scraping the archival pages of a Mailman 2 instance, or by importing an *.mbox* formatted file. Internally, BigBang parses this data into a Pandas DataFrame [McKinney] and stores parsed and normalized email data in *.csv* format.

For the purpose of this study, we scraped data from public Mailman 2 instances associated with the following projects:

- SciPy: http://mail.scipy.org/mailman/listinfo/
- WikiMedia: http://lists.wikimedia.org/mailman/listinfo
- OpenStreetMap: http://lists.openstreetmap.org/listinfo

We selected mailing lists from the SciPy Mailman instance primarily for their relevance to the SciPy community. We also selected some mailing lists from other projects for comparison.

A limiting factor for our analysis is that every new data set introduces new edges cases BigBang's processing logic must take into account. For example, misformatted timestamps cause errors in many archival email data sets. In future work we hope to sample data more systematically in order to establish general principles of collaboration. This preliminery study is merely descriptive.

Deriving interaction graphs

Email is archived in the same text format that email is sent in, as specified in RFC2822 [RFC2822]. Every email is comprised of a message body and a number of metadata headers that are essential for email processing and presentation.

For our study, we have been interested in extracting the implied social network from an email archive of a public mailing list. To construct this network, we have used the *From*, *Message-ID*, and *In-Reply-To* headers of the email.

The *From* field of an email contains data identifying the mailbox of the message author. This often includes both a full name and an email address of the sender. As this is set by the email client and a single person may use many different mailboxes, a single person is often represented differently in the *From* field across many emails. See *Entity Resolution* for our strategies for resolving entities from divergent email headers.

The *Message-ID* header provides a globally unique identification string for every email. The uniqueness of the identifier must be guaranteed by the host that generates the message. It is recommended in [RFC2822] that email hosts accomplish this by including their domain name and combination of the exact date and time, as well as some other unique identifier (such as a process ID number) from the host system. The *In-Reply-To* header is set when an email is sent as a reply to another email. The reply's *In-Reply-To* header will match the *Message-ID* of the original email.

Formally, we construct the directed *interaction graph* G from a set of emails indexed by $i \in I$. Each email consists of a tuple (f_i, r_i), where f_i identifies the mailbox of the sender (corresponding to the *From* header) and $r_i \in I \cup \{\varepsilon\}$ (corresponding to the *In-Reply-To* header) may be a null value ε or be the index of another email.

- For every email i, if there is not one already add a node with label f_i to G corresponding and set its *sent* attribute 1. If such a node already exists, increment its *sent* attribute by 1.
- Iterating again through every email i, if $r_i \neq \varepsilon$, and if there is not one already, then create a directed edge between nodes f_i and f_{r_i} with a *weight* attribute set to 1. If the edge already exists, increment the *weight* attribute by 1.

In sum, the final graph G has a node for every email author annotated by the number of emails from that sender in the data set. There is an edge from f_i to f_j if author f_i ever wrote a reply to an email authored by f_j. The weight of an edge corresponds to the number of these replies in the data set.

The motivation for constructing interaction graphs in this way is to build a concise representation of the social network implied by email data. We posit that building a social network representation based on actual messages sent between people provides a more granular and faithful description of social relationships than one based on higher-level descriptions of social relationships or ties from web services such as Facebook 'friends' and Twitter 'followers'

BigBang implements this interaction graph creation using Python's native email processing libraries, *pandas*, and *networkx*. [Networkx] The following code builds the interaction graph representations.

```
import networkx as nx

def messages_to_interaction_graph(messages):
    """
    *messages* is a Pandas DataFrame, each row
    containing the body and header metadata for
    an email from the archive.
    Messages should be in chronological order.

    Returns a NetworkX DiGraph (directed graph),
    the nodes of which are mailing list participants.

    Nodes have a 'sent' attribute indicating number
    of emails they have sent within the archive.

    Edges from i to j indicate that i has sent at least
    one reply to j. The weight of the edge is equal
    to the number of replies sent from i to j.
    """

    IG = nx.DiGraph()

    from_dict = {}

    sender_counts = {}
    reply_counts = {}

    for m in df.iterrows():
        m_from = m[1]['From']

        from_dict[m[0]] = m_from
        sender_counts[m_from] = \\
            sender_counts.get(m_from, 0) + 1
        IG.add_node(m_from)

        if m[1]['In-Reply-To'] is not None:
            reply_to_mid = m[1]['In-Reply-To']
```

```
            if reply_to_mid in from_dict:
                m_to = from_dict[reply_to_mid]
                reply_counts[m_from][m_to] = \\
                    reply_counts[m_from].get(m_to, 0) + 1

    for sender, count in sender_counts.items():
        IG.node[sender]['sent'] = count

    for m_from, edges in reply_counts.items():
        for m_to, count in edges.items():
            IG.add_edge(m_from, m_to, weight=count)

    return IG
```

One potential objection to this approach is that since the data we are considering comes from public mailing lists where every message has a potentially large audience, it may be misleading to build a network representation on the assumption that a reply is directed primarily at the person who was replied to and not more broadly to other participants in a thread or, even more broadly, to the mailing list as a whole. While this is a valid objection, it points to the heart of what is distinctive about this research. While there have been many studies of social network formation in conventional settings, the conditions of open collaboration are potentially quite different. Theoretically, we expect them to be explicitly and collectively goal-directed, self-organized for efficient action as opposed to positional social advantage, and designed around an archiving system for the sake of efficiency. Understanding the statistical properties of this particular form of social organization, as opposed to others, is the very purpose of this empirical work.

Entity Resolution

Empirically, over the extent of a mailing list's archival data it is common for the *From* fields of emails to vary even when the email is coming from the same person. Not only do people sometimes change their email address or use multiple addresses to interact with the same list, but also different email clients may represent the same email address in the *From* header in different ways. BigBang includes automated techniques for resolving these entities, cleaning the data for downstream processing.

Data from the *From* header of messages stored by Mailman is most often represented in a form that includes both a full name representation and an email representation. Unfortunately these fields can vary widely for one person. Table 1 shows some of the variability that might appear for a single prolific sender. Variation in entity representation is a source of noise in our research and an ongoing area of development for BigBang.

For the study in this paper, we have implemented a heuristic system for entity matching.

- First we standardize the data by converting it to lower case and normalizing " at " and "@".
- Then we construct a similarity matrix between each entry. Each entry is parsed into email and full name subfields. The value of the similarity matrix at cell (i, j) is 1 if there is an exact match of *either* the email address or the full name, and 0 otherwise.
- We then construct a graph from the similarity matrix and treat each *connected component* (group of nodes that are connected to each other by at least one path) as an entity.

Under this procedure, all of the above email addresses would be collapsed into a single entity. These heuristics were developed through informal but thorough investigation of mailing list data we have analyzed for this paper. We leave it to future work to formally test and improve this method with respect to a sufficiently large and labeled test data set.

Variations
tyrion.lannister at gmail.com (Tyrion Lannister)
Tyrion.Lannister at gmail.com (Tyrion Lannister)
Tyrion.Lannister at gmail.com (Tyrion.Lannister@gmail.com)
Tyrion.Lannister at gmail.com (Tyrion.Lannister at gmail.com)
Tyrion.Lannister@gmail.com (Tyrion Lannister)
Tyrion.Lannister@gmail.... (Tyrion Lannister)
Tyrion.Lannister@gmail.com
Tyrion.Lannister at gmail.com (Tyrion)
tyrion at lanister.net (Tyrion Lannister)
halfman@council.kings-landing.gov (Tyrion Lannister)
halfman@council.kings-landing.gov (Tyrion Lannister, Hand of the King)
halfman@council.kings-landing.gov (halfman@council.kings-landing.gov)
tyrion+hand at lanister.net (Tyrion Lannister)
tyrion.lannister at gmail.com (=?UTF-8?B?RGF2aWQgQWJpw6Fu?=)
"Tyrion Lannister" <Tyrion.Lannister@gmail.com>

TABLE 1: *Examples of variations in* From *header values corresponding to the same person in an email archive. Some of these changes reflect changes of email address. Others are artifacts of the users' email clients and the mailing list software.*

In our interaction graph study, this has the effect of combining several nodes into a single one in a way that's similar to the *blockmodel* technique. The edges to and from the derived node are weighted by the sum of the edges of the original nodes. The *sent* attribute of the new node is also set as the sum of the *sent* attribute of the original nodes.

Results

We computed the (unweighted) degre distribution and the weighted and unweighted degree assortativities of each of the mailing lists for which we collected data. We also aggregated the interaction graphs of each list into a single graph that we have called *total* and ran the same analysis.

Every mailing list of the 13 we analyzed exhibits degree disassortivity better fit to log-normal instead of power law distribution. This is the meaning of the negative R value given in Table 2.

The p values require special explanation. The value given is computed by the [Alstott2014] package *in the direction of the best fitting distribution*. Since the best fitting distribution is log-normal, the null hypothesis used for computing the p-values is that the data was generated from a power law distribution. That the p-value for no individual mailing list is beneath a threshold of statistical significance (such as $p < .05$ speaks to the similarity between these two distributions that is the source of such confusion and debate, as outlined previously. Especially for lists with low n, the [Clauset2007] test can be entirely inconclusive as to which distribution is more likely.

List name	List Source	n	R value	p	in,in,weighted	out,out,weighted	in,in,unweighted	out,out,unweighted
total	All sources	9576	-7.62	**0.01**	-0.13	-0.12	-0.21	-0.17
numpy-discussion	SciPy	2973	-0.76	0.40	-0.22	-0.20	-0.29	-0.26
scipy-user	SciPy	2735	-0.02	0.31	-0.11	-0.11	-0.19	-0.18
wikimedia-l	WikiMedia	1729	-3.65	0.07	-0.15	-0.15	-0.21	-0.20
ipython-user	SciPy	1085	-0.33	0.23	-0.27	-0.26	-0.29	-0.26
scipy-dev	SciPy	1056	-0.33	0.58	-0.28	-0.26	-0.31	-0.29
ipython-dev	SciPy	689	-0.52	0.08	-0.25	-0.24	-0.36	-0.36
hot	OpenStreetMap	524	-0.85	0.40	-0.19	-0.20	-0.24	-0.24
astropy	SciPy	404	-0.08	0.77	-0.16	-0.20	-0.16	-0.16
gendergap	WikiMedia	301	-0.86	0.40	-0.15	-0.18	-0.20	-0.21
apug	SciPy	121	-0.01	0.52	-0.20	-0.20	-0.21	-0.22
maps-l	WikiMedia	118	-0.00	0.95	-0.19	-0.18	-0.27	-0.26
design	WikiMedia	111	-3.62	0.10	-0.18	-0.17	-0.21	-0.21
potlatch-dev	OpenStreetMap	75	-0.00	0.97	-0.01	-0.08	-0.45	-0.34

TABLE 2: *Results of analysis. For each mailing list archive, number of participants n, loglikelihood ratio R and statistical significance p in the direction of the best fit. In all cases, the log-normal distribution is a better fit, though only in the case of the aggregated graph is the power-law distribution ruled out with statistical significance. We compute weighted and unweighted variations of (in,in) and (out,out) degree assorativity.*

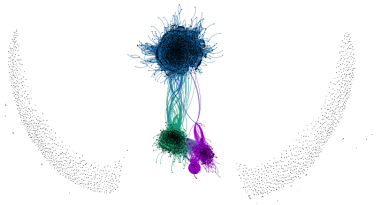


Fig. 1: *Interaction graph of all participants across all mailing lists explored in this study, rendered with [Gephi]. The large blue module is roughly the SciPy community. The green module is the Wikimedia community. The purple module is the OpenStreetMap community. Notably, these communities are not completely disjoint. There are several bridge nodes, meaning there are some mailboxes that have participated in two or more of the communities represented. Singleton points on either side of the central component indicate email authors to whom nobody ever replied.*

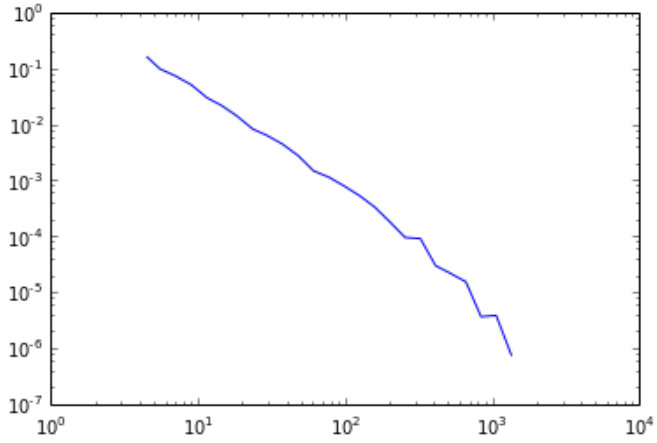


Fig. 2: *A common technique for testing whether a distribution fits a log-normal or power law distribution is to plot its density function on log-log axes and observe whether the tail of the distribution drops below the line of best fit. As an illustration, this is the log-log plot of the probability density function for the unweighted degrees of the aggregated* total *interaction graph. In this paper, we eschew this technique on the grounds that it is biased for reasons discussed in depth in [Clauset2007].*

In the *total* interaction graph, we can rule out that the data was generated from a power law distribution because $p < .05$. One reason for this may be simply because the graph size n is much larger than for any of the individual graphs. Another may be because of the sampling method of aggregating otherwise mainly separate networks.

We find in all cases that interaction graphs are disassortative. We have presented here the results of computing both weighted and unweighted variations of directed (in,in) and (out,out) assortativity. (In all cases, (out,in) and (out,in) were similar enough to the values given that we felt they provided no additional insight to the reader). We observe that the disassortativity of the interaction graphs appears to be insensitive to graph size n. We tentatively conclude that this disassortativity is therefore not of the residual sort found in small Barabási-Alpert graphs. A more thorough analysis of this point may be the subject of future work.

In most (but not all) cases, unweighted disassortativity in interaction graphs is more extreme than its weighted variation. As little work has been done on weighted degree disassortativity, we find this notable.

Discussion

We have found no empirical support for email interaction networks having power law degree distribution, as opposed to a log-normal degree distribution. Interpretation of this result will vary depending on the "prior" probability on assigns to finding power law and log-normal distributions in social processes like this. As similar processes may generate both kinds of heavy-tail distributions, we can say only that our study suggests we should not be tethered to models that guarantee scale-free distributions such as Barabási-Alpert when explaining the interaction network data. We consider the development of a network generation model whose degree is determined by a multiplicative process as a direction for future work.

The statistical strength of the rejection of the power law

hypothesis in the case of the aggregate interaction graph is noteworthy because it suggests that other social network analysis may suffer from a kind of myopia. Recall that preferential attachment requires that new nodes attach according to a probability distribution that is a global property of the network. But considering the growth of largely disjoint communities of collaborators, it is *prima facie* for one participant to understand the aggregate network structure. A network formation process that is more sensitive to this modularity may be a better fit for aggregated collaboration data.

A possible explanation for the disassortativity of these interaction graphs is a community norm of inclusiveness. If community leaders (who have high degree) make it a common practice to respond to new or infrequent participants in an effort to encourage them to contribute further, that would lead to disassortative mixing of degree. On the other hand, this mixing pattern could be the result of a much more generic statistical process.

It is our good fortune that the network data we study is granularly time-stamped. Since the total network structure is derived from an archive in which every email is annotated with a particular time, we see an opportunity to test generative models for there predictions during the whole duration of network growth.

Though anecdotally there is a difference between typical behavior on an open source project's developer list (e.g. *ipython-dev* and *scipy-dev* in our data set) and a projects user list (e.g. *ipython-user* and *scipy-user*), these behavior differences do not surface as a clear statistical pattern in our study. A direction for future work is to more carefully operationalize and test for these behavioral differences.

We anticipate that research supported by BigBang will contribute to discourse on social roles in on-line communities [SocWik], [SocRole], measurement of digital labor [LaborWik], and the relationship between social structure and technical modularity [Zanetti2012].

We have also built BigBang and conducted this preliminary analysis with a number of applications in mind. One is anomaly detection in the open source ecosystem as a method of supply chain risk management. An statistical understanding of the typical patterns of collaborative behavior in open source software development could form the foundation for techniques that detect deviations from those patterns. If non-adherence to these patterns were correlated with propensity for software to be buggy or brittle, then detecting non-adherence could play a useful role in community self-management.

Another potential application of this research is in the appropriate incentivization of participation in open source development. Supposing, as seems likely, that open source software development is truly a collective effort and not merely the sum of many individual efforts, the question of how to best incentivize contributions to open source software is not an easy one. An understanding of how the network structure of collaboration relates to collective productivity could inform incentive plans that are sensitive to participants unique role within the network.

I gratefully acknowledge the helpful comments of Christine Choirat, Allen Downey, Thomas Kluyver, and Skipper Seabold.

REFERENCES

[Alstott2014] Alstott J, Bullmore E, Plenz D (2014) powerlaw: A Python Package for Analysis of Heavy-Tailed Distributions. PLoS ONE 9(1): e85777. doi:10.1371/journal.pone.0085777

[AlbertBarabási] Reka Albert and Albert-László Barabási. 2002 Statistical mechanics of complex networks. Reviews of Modern Physics, vol 74

[BarabásiAlbert] Albert-László Barabási & Reka Albert. Emergence of Scaling in Random Networks, Science, Vol 286, Issue 5439, 15 October 1999, pages 509-512.

[Benthall2013] Benthall, S. 2013. "Reflexive Data Science: An Overview". http://dlab.berkeley.edu/blog/reflexive-data-science-overview

[Clauset2007] A. Clauset, C.R. Shalizi, and M.E.J. Newman. Power-law distributions in empirical data. arXiv:0706.1062, June 2007.

[Fogel] Fogel, K. 2013 *Producing Open Source Software.* http://producingoss.com/

[Foster2009] Foster, J, Foster, D, Grassberger, P, and Paczuski, M. 2010 "Edge direction and the structure of networks" PNAS 2010 107 (24) 10815-10820; published ahead of print May 26, 2010, doi:10.1073/pnas.0912671107

[Gephi] Bastian M., Heymann S., Jacomy M. (2009). Gephi: an open source software for exploring and manipulating networks. International AAAI Conference on Weblogs and Social Media.

[LaborWik] R. Stuart Geiger and Aaron Halfaker. 2013. *Using edit sessions to measure participation in wikipedia.* In Proceedings of the 2013 conference on Computer supported cooperative work (CSCW '13). ACM, New York, NY, USA, 861-870.

[McKinney] Wes McKinney. Data Structures for Statistical Computing in Python, Proceedings of the 9th Python in Science Conference, 51-56 (2010)

[Mitzenmacher2003] Mitzenmacher, M. 2003. "A Brief History of Generative Models for Power Law and Lognormal Distributions." Internet Mathematics Vol. 1, No. 2: 226-251

[Networkx] Aric A. Hagberg, Daniel A. Schult and Pieter J. Swart, "Exploring network structure, dynamics, and function using NetworkX", in Proceedings of the 7th Python in Science Conference (SciPy2008), Gäel Varoquaux, Travis Vaught, and Jarrod Millman (Eds), (Pasadena, CA USA), pp. 11–15, Aug 2008

[Newman2002] Newman, M. E. J. 2002. "Assortative mixing in networks."

[Newman2003] Newman, M. E. J. 2003. "Mixing patterns in networks." Phys. Rev. E 67, 026126

[Noldus2015] Noldus, R and Mieghem, P. 2015. "Assortativity in Complex Networks" Journal of Complex Networks. doi: 10.1093/comnet/cnv005

[RFC2822] Resnick, P. 2001. "Internet Message Format". Network Working Group, IETF.

[SocWik] Howard T. Welser, Dan Cosley, Gueorgi Kossinets, Austin Lin, Fedor Dokshin, Geri Gay, and Marc Smith. 2011. *Finding social roles in Wikipedia.* In Proceedings of the 2011 iConference (iConference '11). ACM, New York, NY, USA, 122-129.

[SocRole] Gleave, E.; Welser, H.T.; Lento, T.M.; Smith, M.A., *"A Conceptual and Operational Definition of 'Social Role' in Online Community,"* System Sciences, 2009. HICSS '09. 42nd Hawaii International Conference on , vol., no., pp.1,11, 5-8 Jan. 2009

[Tinatti2012] Tinati, R., Carr, L., Hall, W. and Bentwood, J. (2012) Scale Free: Twitter's Retweet Network Structure. At Network Science 2012, Evanston, US.

[Zanetti2012] Zanetti, M. and Schweitzer, F. 2012. "A Network Perspective on Software Modularity" ARCS Workshops 2012, pp. 175-186.

[Zhou2011] Zhou T, Medo M, Cimini G, Zhang Z-K, Zhang Y-C (2011) Emergence of Scale-Free Leadership Structure in Social Recommender Systems. PLoS ONE 6(7): e20648.

Visualizing physiological signals in real-time

Sebastián Sepúlveda‡*, Pablo Reyes‡, Alejandro Weinstein‡

https://www.youtube.com/watch?v=6WxkOeTuX7w

Abstract—This article presents an open-source Python software package, dubbed RTGraph, to visualize, process and record physiological signals (electrocardiography, electromyography, etc.) in real-time. RTGraph has a multiprocess architecture. This allows RTGraph to take advantage of multiple cores and to be able to handle data rates typically encountered during the acquisition and processing of biomedical signals. It also allows RTGraph to have a clean separation between the communication and visualization code. The paper presents the architecture and some programming details of RTGraph. It also includes three examples where RTGraph was adapted to work with (i) signals from a Inertial Measurement Unit (IMU) in the context of a biomechanical experiment; (ii) electromyography signals to estimate muscle fatigue; and (iii) pressure signals from a device used to monitor nutrition disorders in premature infants.

Index Terms—real-time processing, visualization, signal processing

Introduction

A common task in biomedical research is to record and visualize physiological signals in real-time. Although there are several options to do this, they are commonly based on proprietary tools, associated with a particular signal acquisition device vendor. This article presents RTGraph, an open-source software package (under MIT license) written in Python, to visualize and record physiological signals in real-time, such as electrocardiography, electromyography and human movement. RTGraph is also capable of doing real-time processing, such as filtering and spectral estimation. RTGraph is open-source,[1] extensible, and has been tested on different Linux distributions, including the RaspberryPi (ARM architecture). RTGraph has a modular design, with a clear separation among its different functionalities, making it easy to add new signal processing tasks, to use different communication protocols (serial, Bluetooth, Sockets, etc.), and customize the user interface for the specific needs of the application.

The main aim of RTGraph is to display multiple signals in real-time and to export them to a file. In the current implementation, the communication between RTGraph and the acquisition device is through the serial port, and it is implemented using the PySerial library. Other communication protocols can be easily added. The real-time display of the signals is implemented using the PyQtGraph library.[2] RTGraph has a multiprocess architecture, based on the multiprocessing Python standard library. This allows having concurrent processes for receiving, processing, and displaying the data. Signal processing tasks, such as spectral estimation, are based on the SciPy/NumPy stack [Ste11]. This architecture makes it possible to ensure that no data is lost and that the user interface has a fast response.

* Corresponding author: ssepulveda.sm@gmail.com
‡ Escuela de Ingeniería Civil Biomédica, Facultad de Ingeniería, Universidad de Valparaíso

1. Available at https://github.com/ssepulveda/RTGraph.

Software architecture

The applications described in this article can be classified as a "data logger". A data logger needs to acquire a stream of data, add a time stamp to the data (if required), and export the time-stamped data to a file in a known file format, such as comma separated value (CSV) format. Optionally, the application can do some processing (filtering, spectral estimation, etc.) before saving the data. In addition, it is also useful to be able to visualize, in real-time, the stream of data.

When developing, evaluating, or validating new hardware or software, it is important to control the outcome of the algorithms and the fidelity and performance of the data acquisition process. In particular, in the field of Biomedical Engineering, the acquisition and processing of biological signals need to be reliable and with a tight control over the sampling frequency. It is also fundamental to ensure that no data is lost during the acquisition and logging process. From a practical point of view, having to wait for the data to be stored before visualizing it (possibly in another program) is cumbersome, slowing down the development process. For these reasons, in this article we present a program capable of: receiving data from a variety of sources (serial port, Bluetooth, Zigbee, Sockets, etc.); processing and visualizing the data in real-time; and saving the data in a file.

The first version of this program was developed for biomechanical engineering research. In our case, this research involves logging, processing and the display in real-time of the signals generated by a nine degrees of freedom inertial measurement unit (9DOF-IMU) [Roe06]. This requires acquiring nine signals with a sampling rate of at least 100 Hz. Six additional signals are computed through a sensor fusion algorithm [Mad11]. A total of 15 signals are displayed and exported as a CSV file. We designed the architecture of the program with these requirements in mind.

Real-time graphics library

Real-time visualization is a key component of our program. To satisfy our requirements we needed a fast and portable graphics library. Since we implemented the GUI in PyQT, we also required that the graphics library should be embeddable in this framework.

2. Available at http://www.pyqtgraph.org.

We used Matplotlib [Hun20] in the first version of the program. This option worked out of the box. We were able to embed a Matplotlib plot in the GUI and interact with it through other elements of the UI without major complications. Although this approach worked for displaying one signal with a sampling rate of 30 Hz, we started to notice a degradation on performance as we increased the number of signals. It is important to note that this is not a flaw of Matplotlib, since the main focus of the library is the production of publication of quality figures, and not the display of real-time data.

Next, we tried PyQtGraph [Cam15]. It is a pure Python implementation, with a focus on speed, portability and a rich set of features. Unlike Matplotlib, PyQtGraph is designed to do real-time plotting and interactive image analysis. It is built on top of PyQt4/PySide, giving easy integration and full compatibility with the Qt framework. This allows using tools like Qt Designer to design the GUI. Using Qt Designer and the examples provided with the PyQtGraph library, it is easy to configure and customize the widgets. PyQtGraph is also built on top of NumPy, facilitating and improving the performance of the manipulation of numerical data. In addition, PyQtGraph wraps up some NumPy/SciPy signal processing functions such as the Fast Fourier Transform and some linear and non-linear filters.[3]

Threading versus Multiprocessing

After using PyQtGraph to its limits in a multithreaded architecture, we could not reliably achieve the desired performance. The limitations of threads in Python [Bea10] combined with the interaction between the UI (main thread) and communication thread, resulted in data losses when the data rate was too high. The Global Interpreter Lock (GIL) [Bea10] prevents threads from taking advantage of multicore systems. In short, it means that a mutex controls threads access to memory. There are ways to work around this limitation. For instance, many of the NumPy primitives take advantage of multiple cores.[4] However, in our case we need to parallelize the reception of the data, the visualization, the processing, and the logging.

To overcome the GIL limitations we used the multiprocessing module, belonging to the Python Standard Library. This module provides an API similar to the threading module, but it uses subprocesses instead of threads [Pyt15]. By letting the OS control the subprocesses, it is possible to take advantage of the multiple cores available on the platform.

Putting it all together

After selecting the key components of the program, the remaining problem is to orchestrate the communication among the processes. We pay special attention to data synchronization, since there are specific considerations that should be taken into account when working with multiple processes.

Figure 1 shows the architecture of RTGraph. The architecture allow us to: (1) Have a multiplatform program; (2) have a separation between the reception and parsing of input data stream and the plotting and logging tasks. The following is a description of each process.

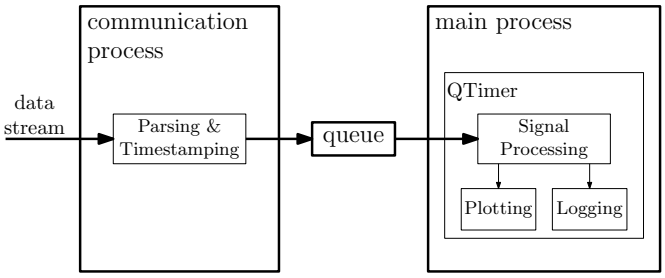


Fig. 1: *Diagram of the software architecture. There are two independent processes. The communication process reads the incoming data stream, parses it, adds a time-stamp (if necessary), and puts the processed data into a queue. The main process reads the data from the queue, processes the data, and then updates the plot and logs the data to a file.*

1) Communication process: This process is responsible for receiving and parsing the data stream sent by the device. The implementation consists of an abstract class, that subclasses the `Process` class from the `multiprocessing` library. Therefore, the methods `__init__` and `run` are overwritten. We also added methods `start` and `stop` to properly start and stop the subprocesses. The class also has methods common to different communication protocols (serial, sockets, etc.). The details of each protocol are implemented in each subclass. This process is also responsible of validating the data and adding the time-stamp to the data, in case the device does not provide it. This guarantees that the data is always time-stamped.
2) Main process: The main process is responsible for initializing the different subprocesses and for coordinating the communication between them. As shown in figure 1, this process instantiates the components that will allow the communication between the subprocesses and also manage the different UI elements. A `Queue`, as implemented by the `multiprocessing` module, is used to connect the communication process with the main process. A `QtTimer` is set to update the real-time plot. By updating the plot at a known frequency, we can control the responsiveness of the program under different conditions. Each time the `QtTimer` triggers a plot update (30 times per second), the `queue` is processed. The queue is read until it is empty and then the plot is updated.

Figure 2 shows the processes viewed by `htop` during the execution of the program. The first process (PID 3095) corresponds to the process initiated by the application. The second one is the communication process (PID 3109).[5]

Programming details

The template for the communication process is implemented through the `CommunicationProcess` class. This template allows for processing data streams coming from a variety of

3. We also evaluated the PyQwt library (http://qwt.sourceforge.net/). This library provides a Python interface to the Qwt library. It is a light implementation with an easy QT integration. It is fast enough to support real-time display of the data. However, this library is not currently maintained, and its author recommended using PyQtGraph (see http://comments.gmane.org/gmane.comp.graphics.qwt.python/506).

4. See http://wiki.scipy.org/ParallelProgramming for details.

5. By default `htop` shows the processes and threads together. Pressing the H key while the program is running shows or hides the threads. In figure 2, the screen is configured to show only the processes.

protocols (serial, sockets, bluetooth, etc.). The design of the class also allows changing some of the communication parameters during run-time. In addition, since the class inherits from the `Process` class, it is trivial to run several instances of the class to receive from multiple devices simultaneously. For instance, it is possible to instantiate the class twice to receive data form two different serial ports at the same time. The following code snippet shows the basic structure of the class.

```
class CommunicationProcess(Process):
    def __init__(self, queue):
        Process.__init__(self)
        self.exit = Event()
        self.queue = queue
        # Initialize the process ...
        # Initialize the acquisition method ...

    def run(self):
        self.init_time = time()
        try:
            while not self.exit.is_set():
                # do acquisition
                # and add time stamp ...
        except:
            raise
        finally:
            self.closePort()

    def openPort(self, port):
        # Port configuration to open

    def closePort():
        self.exit.set()
```

One of the key methods of the `CommunicationProccess` class is `run`. The following code snippets is an example of how to write a serial port interface.

```
class SerialProcess(Process):
    # ...
    def run(self):
        self.init_time = time()
        try:
            while self.ser.isOpen() and \
                  not self.exit.is_set()
            data = self.ser.readline().strip()
            try:
                data = map(float, data.split(','))
                self.queue.put([time() -
                                self.init_time] + data)
            except:
                pass
        except:
            raise
        finally:
            self.closePort()
        # ...
```

In this case, `run` computes the time stamp, then checks if the serial port is open and if the process is not exiting. If both statements are true, a line is read from the serial port. Then, the data is parsed (in this example, the data stream consists of CSV floats). Finally, if the data is valid it is placed in the queue.

The main process is implemented through the `MainWindow` class. It is a subclass of the `QtGui.QMainWindow` class. Inside this class we define the proper acquisition method (serial, sockets, bluetooth, etc.) and the basic plot configurations, and we configure the timers used to update the plots, which trigger the `update_plot` method. The following code snippet shows the basic structure of the class.

```
class MainWindow(QtGui.QMainWindow):
    def __init__(self):
        QtGui.QMainWindow.__init__(self)
```

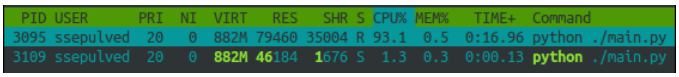


Fig. 2: *Screenshot of `htop` showing the processes associated with the program. The first process (PID 3095) corresponds to the process initiated by the application. The second one is the communication process (PID 3109).*

```
        self.ui = Ui_MainWindow()
        self.ui.setupUi(self)
        # initialize plots ...
        self.ui.plt.setBackground(background=None)
        self.plt1 = self.ui.plt.addPlot(row=1, col=1)

        # initialize variables ...
        # initialize timers ...
        QtCore.QObject.connect(self.timer_plot_update,
                               ...)

    def start(self):
        self.data = CommunicationProcess(self.queue)
        self.data.openPort(...)

        self.timer_plot_update.start(...)
        self.timer_freq_update.start(...)

    def update_plot(self):
        while self.queue.qsize() != 0:
            data = self.queue.get(True, 1)

        # draw new data ...
        self.plt1.clear()
        self.plt1.plot(...)

    def stop(self):
        self.data.closePort()
        self.data.join()
        self.timer_plot_update.stop()
```

The `start` method initializes the communication process. This method is triggered every time the *Start* button is pressed. This allows to change the communication parameters (port name, bauds, etc.) during execution time.

The plot details are also defined in the `MainWindow` class. The following code snippets shows how to customize some PyQtGraph options, such as titles, labels, and line colors.

```
class MainWindow(QtGui.QMainWindow):
    def __init__(self):
        # ...
        # Initializes plots
        self.ui.plt.setBackground(background=None)
        self.plt1 = self.ui.plt.addPlot(row=1, col=1)
        self.plt2 = self.ui.plt.addPlot(row=2, col=1)
        # ...
        self.configure_plot(self.plt1, "title1",
                            "unit1")
        self.configure_plot(self.plt2, "title2",
                            "unit2")

    @staticmethod
    def configure_plot(plot,title, unit,
                       y_min=0, y_max=0,
                       label_color='#2196F3',
                       label_size='11pt'):
        label_style = {'color': label_color,
                       'font-size': label_size}
        plot.setLabel('left', title,
                      unit, **label_style)
        plot.setLabel('bottom', 'Time',
                      's', **label_style)
        plot.showGrid(x=False, y=True)
        if y_min != y_max:
            plot.setYRange(y_min, y_max)
```

Fig. 3: *Screenshot of RTGraph customized and modified to display 3 signals: an EMG signal (first panel), an estimation of the fatigue level (second panel) based on the acquired EMG signal, and three acceleration signals (third panel).*

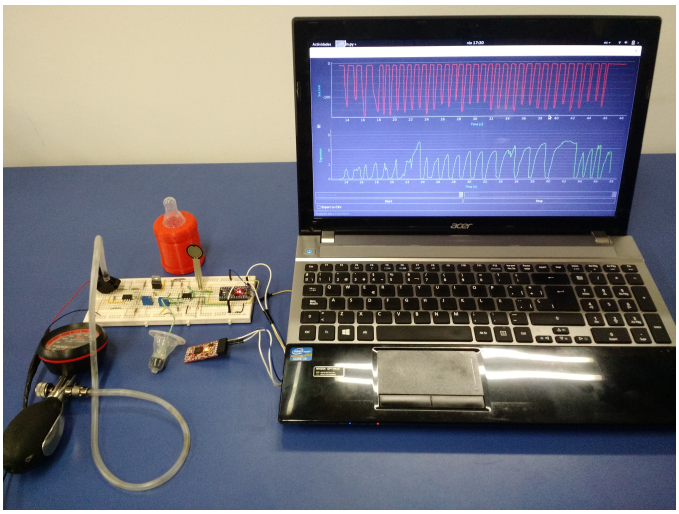


Fig. 4: *Photo of the prototype device used to monitor nutrition disorders in premature infants. An Arduino development platform is used to acquire the signals (two pressure measurements). These signals are acquired by a computer running a modified version of RTGraph.*

```
    else:
        plot.enableAutoRange(axis=None,
                             enable=True)
    plot.setMouseEnabled(x=False, y=False)
```

The class sets the layout of the plots through calls to `self.ui.plt.addPlot` methods. Then, each plot is configured by the `configure_plot` method, where details such as title, range, color, and font sizes are set.

Results

We have used RTGraph with a serial port data stream corresponding to a signal with a sampling frequency of 2 kHz. We have also used it with a data stream from a TCP/IP socket corresponding to 20 signals with a sampling frequency of 500 Hz.

In a biomechanical study we used our program to evaluate a prototype of a wearable device used to estimate muscle fatigue through the EMG signal. RTGraph was customized to acquire and record these data. We also incorporated some steps of a fatigue estimation algorithm [Dim03] in the processing pipeline. We found that having real-time feedback of the signal simplified the procedure to position the wearable device correctly, drastically reducing the amount of time required by the experiments. Figure 3 shows a screenshot of the program while acquiring an EMG signal using a wearable device to study muscle fatigue. The figure shows an EMG signal (first panel), a real-time estimation of the fatigue level (second panel) based on the acquired EMG signal, and three acceleration signals (third panel). See the following links for a video of RTGraph being used to acquire these signals: https://www.youtube.com/watch?v=sdVygxpljII, https://www.youtube.com/watch?v=6WxkOeTuX7w.

An important feature of our program is the ease with wich it can be customized to a specific application. For instance, RTGraph is being used to acquire a set of pressure signals from a device (as seen in figure 4) used to monitor nutrition disorders in premature infants. The customization included: (1) modifying RTGraph to acquire two pressure signals using bluetooth; and (2) to perform some specific signal processing before the visualization. In this example it is important to emphasize that the changes to the program were made by a researcher other than the main developer of our program. We claim that this is possible because our program is written in Python. This makes it easier to understand and modify the code compared to a program written in a lower-level language.

The software package presented in this article has been tested with different devices, communication protocols, platforms and operating systems (OSs). The initial development was done and tested on the platforms x86, x64 and ARM (RaspberryPy) running Linux. However, this version of RTGraph did not work as expected on OS X and Windows, due to some restrictions of the multiprocessing library in these OSs. Despite the fact that OS X is a Unix-like OS, there are some multiprocessing methods not implemented in the multiprocessing library. In particular, the method `qsize`, used to get the approximate size of the queue, is not implemented in OS X. The lack of the `os.fork()` call in Windows adds some extra limitations when running a program on this OS. Since in this case a child process can not access the parent resources, it is necessary that subclasses of the `Process` class must be picklable. Although the documentation of the library contains some suggestions to overcome these restrictions, currently we are not able to run our program on Windows.

Conclusions

In this article we presented a program developed to record, process and visualize physiological signals in real-time. Although many people consider Python as a "slow" language, this article shows that it is possible to use Python to write applications able to work in real-time. At the same time, the clarity and simplicity of Python allowed us to end up with a program that it is easy to modify and extend, even by people who are not familiar with the base code.

We also believe that our solution is a contribution to the open-source and Do It Yourself (DIY) communities. Typically, programs to receive and manipulate data in real-time are developed using proprietary tools such as LabView or MATLAB. The cost of these tools denies members of these communities access to

solutions like those described in this article. As we showed in the results section, in many cases we have used the program with an Arduino acting as an acquisition device. This is a common situation, and we believe that our program can be extended to be used in other fields in need of similar tools.

In the future our first priority is to make our program work on platforms running OS X and Windows. We are currently investigating how to overcome the restriction imposed by the multiprocessing platform on these OSs. Next, we will focus on improving the UI. In particular, we will add an option to change some plotting and processing parameters on the fly, instead of requiring a change in the source code. Finally, we will refactor the architecture of the program to improve the performance, so we can handle higher data rates. In this respect, the main change we plan to do is to move the signal processing computation to another process, leveraging the existence of multi-core machines.

Acknowledgments

This research was partially supported by the Advanced Center for Electrical and Electronic Engineering, Basal Project FB0008, Conicyt.

REFERENCES

[Bea10] D. Beazley. *Understanding the Python GIL*, In PyCON Python Conference. Atlanta, Georgia, 2010.

[Cam15] L. Campagnola. *PyQtGraph. Scientific Graphics and GUI Library for Python*, http://www.pyqtgraph.org/

[Dim03] N. Dimitrova and G. Dimitrov. *Interpretation of EMG changes with fatigue: facts, pitfalls, and fallacies.* Journal of Electromyography and Kinesiology 13.1 (2003): 13-36.

[Hun20] J. D. Hunter. *Matplotlib: A 2D graphics environment*, Computing In Science & Engineering, 9(3):90-95, IEEE COMPUTER SOC, 2007.

[Mad11] S. Madgwick, Andrew JL Harrison, and Ravi Vaidyanathan. *Estimation of IMU and MARG orientation using a gradient descent algorithm.*, Rehabilitation Robotics (ICORR), 2011 IEEE International Conference on. IEEE, 2011.

[Pyt15] Python Software Foundation, *16.6 multiprocessing - Process-based "threading" interface*, https://docs.python.org/2/library/multiprocessing.html

[Roe06] D. Roetenberg, *Inertial and magnetic sensing of human motion*. University of Twente, 2006.

[Ste11] S. van der Walt, S.C. Colbert and G. Varoquaux, *The NumPy Array: A Structure for Efficient Numerical Computation*, Computing in Science & Engineering, 13, 22-30, 2011.

Building a Cloud Service for Reproducible Simulation Management

Faical Yannick Palingwende Congo[‡*]

https://www.youtube.com/watch?v=euMLYw7SNdk

Abstract—The notion of capturing each execution of a script and workflow and its associated metadata is enormously appealing and should be at the heart of any attempt to make scientific simulations repeatable and reproducible.

Most of the work in the literature focus in the terminology and the approaches to acquire those metadata. Those are critical but not enough. Since one of the purposes of capturing an execution is to be able to recreate the same execution environment as in the original run, there is a great need to investigate ways to recreate a similar environment from those metadata and also to be able to make them accessible to the community for collaboration. The so popular social collaborative *pull request* mechanism in Github is a great example of how cloud infrastructures can bring another layer of public collaboration. We think reproducibility could benefit from a cloud social collaborative presence because capturing the metadata about a simulation is far from being the end game of making it reproducible, repeatable or of any use to another scientist that has difficulties to easily get them.

In this paper we define a reproducibility record atom and the cloud infrastructure to support it. We also provide a use case example with the event based simulation management tool *Sumatra* and the container system *Docker*.

Index Terms—metadata, simulations, repeatable, reproducible, Sumatra, cloud, Docker.

Introduction

Reproducibility in general is important because it is the cornerstone of scientific advancement. Either done manually or automatically; reusability, refutability and discovery are the key proprieties that make research results repeatable and reproducible.

One will find that in the literature many research have been done in defining the terminology (repeatability, reproducibility and replicability) [Slezak2011] and investigating approaches regarding the recording of simulations metadata using workflows [Oinn2006], libraries [Langer2014] or event control systems [Guo2012]). These research are critical because they focus on getting to the point where the metadata about a simulation execution have been captured in a qualitative and reliable way. Yet the use of these metadata to recreate the proper execution environment is challenging and is not only extremely valuable to the scientist that ran the simulation. It is more valuable to other scientists that share the same interest and could benefit an easy way to at least get the same results consistently. This is why we think that reproducibility can gain from a more active presence in the cloud through infrastructures that bring an easy access and collaboration around those captured metadata. The social collaborative *pull request* mechanism from Github [MacDonnell2012] is a great example about the importance of cloud infrastructures in enhancing collaboration. In fact many scientific projects from SciPy [Oliver2013] got some interest and contribution because of their exposure on Github and its ease for collaboration.

In this paper we discuss on a structure of a reproducible record atom. It is a record that we propose to ease the reconstruction of the execution environment and allow an easy assessment of its reproducibility by comparing it to others. Then we propose a cloud platform to deliver an online collaborative access around these record atoms. And finally we present an integration use case with the data driven simulation management tool Sumatra [Davidson2010].

A reproducible record atom

Defining what are the requirements that have to be recorded to better enforce the reproducibility of a simulation is of good interest in the community. From more general approaches like defining rules that have to be fulfilled [Sandve2013], to more specific approaches [Heroux2011], we can define a set of metadata that are useful to determine the reproducibility of a simulation. To do so, we have to go from the fact that the execution of a simulation involves mostly five different components: the source code or executable, the input files, the output files, the dependencies and the hosting system. The source code or executable gives all the information about what the simulation is, where it can be found (repository) and how it was run. The input files are all the files being loaded by the simulation during its execution. The output files are all the files that the simulation produced during its execution. The dependencies are all the libraries and tools that are needed by the simulation to run. The hosting system is the system in which the simulation is being ran. These components can be classified into two groups regarding repeatability as a goal. To repeat a simulation execution, the source code and the inputs are part of a group of components that are kept as the same. The dependencies and the host system on the other end are part of the components that will most likely change from the original executing system to another that is attempting a repeat. We think of them as a cause of uncertainties that lead to variations in the outputs when the source

* *Corresponding author: yannick.congo@gmail.com*
‡ *LIMOS - UMR CNRS 6158, Blaise Pascal Univerity, Campus Universitaire des Cezeaux, 2 Rue de la Chebarde, TSA 60125 - CS, 60026, 63178 Aubière CEDEX FRANCE*

code and inputs are still the same. To assess a reproducibility property on a simulation, we provide the Table 1. It defines the reproducibility properties involved (repeatable, reproducible, non-repeatable, non-reproducible or unknown) when comparing the source code, inputs and outputs of two simulations. This table is used in conjunction with the models presented later to assess the reproducibility property of any record atom in the system compared to another through a requesting mechanism that will be detailed further.

One thing is to be able to gather crucial information about a simulation yet another challenging one is to be able to recreate the same execution context as when the simulation was done the first time. It is impossible to consistently reproduce a simulation across platforms and machines if we do not have a uniform and portable way to bundle the whole simulation execution environment.

We think that container based systems [Bottomley2014] are a possible solution to ensure the consistency of the operating system and dependencies on which the simulation runs. Building and sharing a container that will deliver a runnable image in which the simulation execution is well scoped and controlled will ensure that across machines and platforms we get closer to a consistent execution environment [Melia2014].

Thus we propose here a container based recording alternative along with the captured metadata as a set of four models that combined together should be enough to deliver a reproducible record atom storage. We show here the project model in Table 2.

It describes the simulation. Its *history* field is the list of container images that have been built each time that the project source code changes. The container is setup directly from the source code of the simulation. We also propose a container model that is as simple as shown in the Table 3.

Based on the project's model in Table 2, we designed a record atom model shown in Table 4. A record is related to a project and a container in the history of the project containers. When a record atom is created, its container is the last container in the the project's history at that time. Thus, a record atom that will be done on a modified project source code has to be performed after the new container for this modified version of the project get pushed to the history field. This way we ensure that two records with different containers are from two different sources codes and also two records with the same containers are from the same source code.

A record atom reproducibility property assessment is done through a differentiation process. A differentiation process is a process that allows the resolution of a record atom reproducibility property compared to another. In this situation, the two record atoms are considered being from simulations that try to achieve the same goals. It is quite hard to know at a high level standpoint if two record atoms are the same because it will most likely be a domain related decision that proves that both records support the same claims. We focus here in an approach that provides some basic differentiation methods and allow the definition of new ones. Thus, the differentiation will most likely be based on the targeted record atom owner domain knowledge and understanding on the method used. Since the record atom is the state of a simulation execution, the inputs, outputs, dependencies and system fields have to be provided every time because from a run to another any of those may be subject to a change. Sometimes an action as simple as upgrading a library can have terrible and not easy to determine consequences on the outputs of another execution of the same simulation in the same system.

A differentiation request or shortly *diff request* is the *contract* on which the mechanism described before runs. A requesting record owner asks a targeted record atom owner to validate a record atom reproducibility proposal from him. In this mechanism, the requesting party has to define what the assessment is based on: repeated, reproduced, non-reproduced and non-repeated. This party also has to define the base differentiation method on which the assessment has been made: default, visual and custom. A default differentiation method is a Leveinstein distance[1] based differentiation on the text data. A visual one is a nobservation based knowledge assessment. And custom is left to the requester to define and propose to the targeted. It is important to point that the Table 1 is the core scheme of comparison that all differentiation request have to go through upon submission. To be accepted in the platform, the *diff request* assessment has to comply with the content of that Table. As such a *diff request* for two requests that have different inputs contents cannot be assessed as a repeat compared to one another because an input variation should lead to a reproducible assessment as pointed in the Table 1. The targeted record atom owner has to answer to the request by setting after verification on his side, the status of the request to agreed or denied. By default the status value is *proposed*. The table 5 represents the fields that a diff request contains. In fact one may say that in a model level a solved diff request is a relationship of reproducibility assessment between two records.

A project reproducibility property can be assessed from the differentiation requests on its records. All the requests that have a status to *agreed* represent a list of accepted couple of records that have been resolved as: repeated, reproduced, non-repeated and non-reproduced.

Data Driven Cloud Service Platform

To support simulation management tools metadata, we propose a cloud platform that implements the reproducible assessable record described previously. This platform has two sides. As shown in the Figure 1, an API[2] access and a Web Frontend[3] access. These two services are linked to a MongoDB[4] database that contains: the user accounts, the projects, the records, the containers and the differentiation requests. We implemented some restrictions depending on the type of access.

The API service exposes endpoints that are accessible by the Simulation management tool from the executing machine. It is a token based credential access that can be activated and renewed only from the Web Frontend access. The API allows the Simulation Management tools to push, pull and search projects and records. The API documentation will be available publicly and will present the endpoints, HTTP[5] methods and the mandatory fields in a structured JSON[6] format request content.

The Web Frontend service on the other end is controlled by the Cloud service. The Cloud service is accessible only from the Web Frontend. Thus when the user interacts with the Web Frontend, he is actually securely communicating with the Cloud service. This strongly coupled design allows a flexible deployment and upgrades but at the same time harden the security of the platform. The frontend access allows the user to manage his account and handle his API credentials which are used by the Simulation Management tools to communicate with the platform. It also allows the user to visualize his projects, records and requests. It is the only place

1. Levenshtein distance is a string metric for measuring the difference between two sequences.

Output Files	Source Code and Input Files			
	Same and Same	Same and Different	Different and Same	Different and Different
Same	Repeatable	Reproducible	Reproducible	Reproducible
Different	non-repeatable	Unknown	Unknown	Unknown

TABLE 1: *Reproducibility assessment based on source code, inputs and outputs*

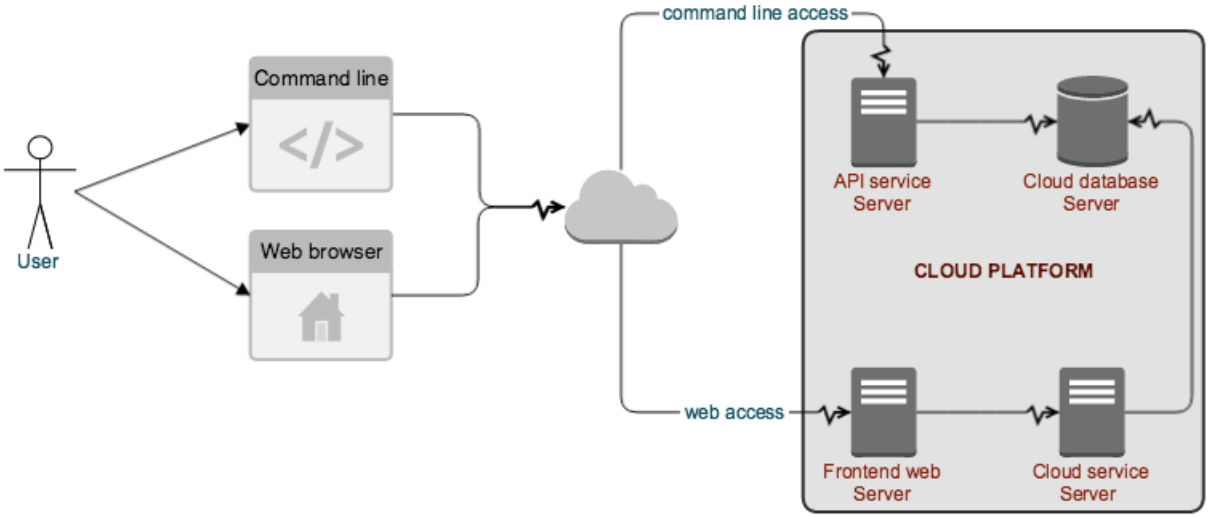


Fig. 1: *Platform Architecture.*

Fields	Descriptions
created	string: simulation creation timestamp.
private	boolean: false if project is public.
name	string: project name.
description	string: full description of the project.
goals	string: project goals.
owner	user: the creator of the project.
history	list: container images list.

TABLE 2: *Simulation metadata Project Model.*

Fields	Descriptions
created	string: simulation creation timestamp.
system	string: docker, rocket, ...
version	dict: version control source code's tag .
image	string: path to the image in the cloud.

TABLE 3: *Simulation metadata Container Model.*

where the user can update some content regarding a project, record or interact with his differentiation requests.

On the platform, the API is the only place where projects and records are automatically created. On the Web side this is still possible but it is a manual process.

A Simulation tool that needs to interact with the platform has to follow the endpoints descriptions in Tables 6 and 7.

Fields	Descriptions
created	string: execution creation timestamp.
updated	string: execution update timestamp.
program	dictionary: command, version control,...
inputs	list: input files.
outputs	list: output files.
dependencies	list: dependencies.
status	string: unknown, started, paused, ...
system	dictionary: machine and os information.
project	project: the simulation project.
image	container: reference to the container.

TABLE 4: *Simulation metadata Record Model.*

Fields	Descriptions
created	string: request creation timestamp.
sender	user: responsible of the request.
toward	record: targeted record.
from	record: requesting record.
diff	dictionary: method of differentiation.
proposition	string: repeated,reproduced,...
status	string: agreed,denied,proposed.

TABLE 5: *Simulation Record Differentiation Request Model.*

Endpoint	Content	
	Method	Envelope
$/api/v1/ < api-token > /project/pull/ < project-name >$	GET	null. Note: pull metadata about the project.
$/api/v1/ < api-token > /project/push/ < project-name >$	POST	name, description, goal... custom. Note: push project metadata.

TABLE 6: *REST Project endpoints*

Endpoint	Content	
	Method	Envelope
$/api/v1/ < api-token > /record/push/ < project-name >$	POST	program, inputs, outputs... Note: push metadata about the record.
$/api/v1/ < api-token > /record/pull/ < project-name >$	GET	null. Note: pull the container.
$/api/v1/ < api-token > /record/display/ < project-name >$	GET	null. Note: metadata of the record.

TABLE 7: *REST Record endpoints*

Integration with Sumatra and Use Case

Sumatra Integration

Sumatra is an open source event based simulation management tool. To integrate the cloud API into Sumatra we briefly investigate how Sumatra stores the metadata about a simulation execution.

To store records about executions, Sumatra implements record stores. It also has data stores that allow the storage of the simulation results. As of today, Sumatra provides three data storage options:

- FileSystemDataStore: It provides methods for accessing files stored on a local file system, under a given root directory.
- ArchivingFileSystemDataStore: It provides methods for accessing files written to a local file system then archived as .tar.gz.
- MirroredFileSystemDataStore: It provides methods for accessing files written to a local file system then mirrored to a web server.

Sumatra also provides three ways of recording the simulation metadata:

- ShelveRecordStore: It provides the Shelve based record storage.
- DjangoRecordStore: It provides the Django based record storage (if Django is installed).
- HttpRecordStore: It provides the HTTP based record storage.

Regarding the visualization of the metadata from a simulation, Sumatra provides a Django[7] tool named *smtweb*. It is a local web app that provides a web view to the project folder from where it has been ran. For a simulation management tool like Sumatra there are many advantages in integrating a cloud platform into its record storage options:

- Cloud Storage capability: When pushed to the cloud, the data is accessible from anywhere.
- Complexity reduction: There is no need for a local record viewer. The scientist can have access to his records anytime and anywhere.
- Discoverability enhancement: Everything about a simulation execution is a click away to being publicly shared.

As presented in the list of record store options, Sumatra already has an HTTP based record store available. Yet it does not suite the requirements of the cloud platform. Firstly because there is no automatic mechanism to push the data in the cloud. The MirroredFileSystemDataStore has to be fully done by the user. Secondly we think there is need for more atomicity. In fact, Sumatra gather the metadata about the execution and store it at the end of the execution, which can have many disadvantages generally when the simulation process dies or the Sumatra instance dies.

To integrate the cloud API and fully comply to the requirement cited before, we had to implement and update some parts of the Sumatra source code:

- DataStore: Currently the collect of newly created data happens at the end of the execution. This creates many issues regarding concurrent runs of the same projects because the same files are going to be manipulated. We are investigating two alternatives. The first is about running the simulation in a labeled working directory. This way, many runs can be done at the same time while having a private labeled space to write to. The second alternative consists of writing directly into the cloud. This will most likely break the already implemented data and record store paradigm in Sumatra.
- RecordStore: We make the point that the simulation management tool is the one that should comply to as many API interfaces as possible to give the user as many interoperability as possible with cloud platforms that support reproducible records. Thus, we intend to provide a total new record store that will fully integrate the API into Sumatra.
- Recording Mechanism: In Sumatra the knowledge of the final result of the execution combined with atomic state monitoring of the process will allow us to have a dynamic state of the execution. We want to make Sumatra record creation a dynamic many points recorder. In addition to an active monitoring, this feature allows the scientist to have basic informations about its runs may they crash or not.

2. Application Programming Interface.
3. Client browser access.
4. An Agile, Scalable NoSQL Database: https://www.mongodb.org/
5. HyperText Transfert Protocol.
6. A Data-Interchange format: http://json.org/
7. Python Web Framework: https://www.djangoproject.com/

Example project with Sumatra

The Sumatra repository[8] provides three test example projects. This example is based on the python one[9]. We propose here an example project as a base line to make the scientist's simulation comply with the principles described here. The platform currently supports docker as a container based system and Sumatra as a simulation management tool.

The example is the encapsulation of the execution of a python simulation code *main.py* that is simply:

```
import numpy
import sys

__version__ = "1.2.3a"

# version numbers are deliberately different
# for testing purposes
def get_version():
    return (1, 2, "3b")

def run():
    parameter_file = sys.argv[1]
    parameters = {}
    # this way of reading parameters
    execfile(parameter_file, parameters)
    # is not necessarily recommended
    numpy.random.seed(parameters["seed"])
    distr = getattr(numpy.random,
                    parameters["distr"])
    data = distr(size=parameters["n"])

    numpy.savetxt("Data/example2.dat",
                  data)


if __name__ == "__main__":
    run()
```

The input file to provide is *default.param* that contains:

```
# seed for random number generator
seed = 65785
# statistical distribution to draw values from
distr = "uniform"
# number of values to draw
n = 100
```

The instrumented project is organized as following:

- Python main: It's the simulation main source code.
- Git ignore: It contains the files that will not be versioned by git.
- Requirements: It contains all the python requirements needed by the simulation.
- Dockerfile: It contains the simulation docker container setup.
- Manage files: It's a script that allows the scientist to manage the container builds and the simulation executions.
- Sumatra integrate: It is a modified copy of Sumatra that integrates the API.

This demo example is currently working in linux and OsX systems and to run it, the scientist has to proceed as following:

- Get the source from github.
- To have an API key: Create an account on the platform and login.
- Access the user profile: In the home page, the round user floating image display two buttons that are the user profile access. Click the first one to view and the second one to edit the profile.
- Get the API key: Go to view the user profile and copy the string near the key image.
- Open the manage.sh file and replace the API key 3a8d4cc793bd3e5b85c733b523584... by this string. Update data path to be where the default.param file is located and the container path to be where the container image will be placed. By default the container image is generated in the demo-sumatra directory.
- Git global settings: Replace the git global username and email by the scientist's.
- Build the container image.
- Run the simulation: It will run main.py in the container and push the record along with the container image to the cloud space in the platform.
- Outcome: In the online dashboard, there will be a new project named demo-sumatra with a record that can be downloaded and executed with an input file like the default.param.

The following bash code, is the set of commands that will be ran by the scientist. Note that the first echo is the step described previously about replacing the API key in *manage.sh* by the scientist's one.

```
git clone github.com/faical-yannick-congo/ddsm-demo
cd ddsm-demo
git checkout setup
echo "Update the api key."
echo "To build the container image: "
./manage.sh --build --simulation demo-sumatra
echo "To run the simulation: "
./manage.sh --run-core --simulation demo-sumata
```

For a new simulation project we suggest that the scientist follow the same source structure as done in the demo example. Then to instrument his simulation, the scientist has to go through some few steps:

- Source code: The scientist may remove the script main.py and include his source code.
- Requirements: The scientist may provide the python libraries used by the simulation there.
- Dockerfile: Uncomment line 54 by removing the first character. Also the installation of non python libraries should be added here.
- Management: Here, the scientist has to update the API key and the git global settings (username and email).
- Running command: The scientist has to determine the full command that will be ran with the simulation and the input data to provide. The -v argument for docker allows file mapping from the local file system to the docker container. The -c argument allows the user to run a string command in the docker's /bin/bash terminal. More information can be found about those arguments. The scientist should update the run string to fit the simulation execution.

After performing this instrumentation on his simulation source code, the scientist has to build and run the simulation as done previously for the demo example. In addition, it is important that the scientist builds the container every time that the source modifications are ready to be tested as justified before when presenting the record model. In this case a newly exported image will be available to be ran with Sumatra. After a build, a run will

8. https://github.com/open-research/sumatra.git
9. https://github.com/faical-yannick-congo/ddsm-demo/tree/setup

execute the simulation and create the associated record that will be pushed to the cloud API. The interesting part of such a design is that the record image can be ran by any other scientist with the possibility to change the input data. This allows reproducibility at an input data level. For source code level modifications, the other scientist has to recreate an instrumented project. In the manage script, an API token is required to be able to access the cloud API. The scientist will have to put his own. A further detailed documentation will be provided. The source code of the demo can be found here[10]. It has been tested on an Ubuntu 15.04 machine and will work on any Linux or OsX machine that has docker installed.

The instrumented example presented here, has been done from a local development instance of the platform. AWS[11] server instances are being setup to host a public access to a production version of this platform. To reproduce this example demo, the url inside the *manage.sh* will have to be update accordingly to the location of the API endpoint. Further information will be delivered.

Conclusion and Perspective

Scientific computational experiments through simulation is getting more support to enhance the reproducibility of research results. Execution metadata recording systems through event control, workflows and libraries are the approaches that are investigated and quite a good number of softwares and tools implement them. Yet the aspect of having these records discoverable in a reproducible manner is still an unfulfilled need. This paper proposes a container based reproducible record atom and the cloud platform to support it. The cloud platform provides an API that can easily be integrated to the existing Data Driven Simulation Management tools and allow: reproducibility assessments, world wide web exposure and sharing. We described an integration use case with Sumatra and explained how beneficial and useful it is for Sumatra users to link the cloud API to their Sumatra tool. This platform main focus is to provide standard and generic ways for scientists to collaborate through reproducible record atoms and interact by the mean of differentiation procedures that will allow them to assess if a simulation is repeatable, reproducible, non-repeatable, non-reproducible or if its an ongoing research. A differentiation request description has been provided and can be presented as a hand shake between scientists regarding the result of simulation runs. One can request a reproducibility assessment property validation from a record against another.

We are under integration investigation for other simulation management tools used in the community. In the short term this platform will hopefully be a space where scientists could clone the entire execution environment that another scientist did. And from there be able to verify the claims of the project and investigate other execution on different input data. The container based record described here, we hope, will allow a better standard environment control across repeats and reproductions, which is a very hard battle currently for all simulation management tools. Operating systems, compilers and dependencies variations are the nightmare of reproducibility tools because the information is usually not fully accessible and recreating the appropriate environment is not an easy straight forward task.

10. https://github.com/faical-yannick-congo/ddsm-demo
11. Amazon Web Services: http://aws.amazon.com/

Finally it is important to point out that in some cases the five components (source code, inputs, hosting system, dependencies and outputs) cited before are not sufficient because the design of the simulation itself has to follow a rigorous method to better enforce reproducibility. Parallel stochastic simulations presents this requirement of determining the right techniques for generating parallel pseudorandom numbers [Hill2015].

Acknowledgments

This research paper is made possible through the help of my thesis supervisors and colleagues.

First and foremost, I would like to thank Dr. David Hill and Dr. Jonathan Guyer for their most support, encouragements and critics.

Second, I would also like to thank Dr. Daniel Wheeler for his ideas and brainstorms at the early stage of this investigation and his continuous research for better technologies for computational science.

Finally, I would like to thank Dr. Andrew Reid and Dr. Stephen Langer for their exceptional willingness to help me reshape and bring more lights in this paper. They kindly read my paper and offered invaluable detailed advices on grammar and organization of the paper.

References

[Slezak2011] P. Slezák and I. Waczulíková, *Reproducibility and Repeatability*. Physiological Research, Volume 60, Issue 1, pp. 203-205, 2011.

[Oinn2006] Tom Oinn et al, *Taverna: lessons in creating a workflow environment for the life sciences*. Concurrency and Computation: Practice and Experience, Special Issue: Workflow in Grid Systems, Volume 18, Issue 10, pages 1067–1100, 25 August 2006.

[Langer2014] Stephen Langer et al, *gtklogger: A Tool For Systematically Testing Graphical User Interfaces*. NIST Internal Publication, pp. 2-3, October 2014.

[Guo2012] Philip Guo, *CDE: A Tool for Creating Portable Experimental Software Packages*. Reproducible Research For Scientific Computing, pp. 2-3, October 2012.

[MacDonnell2012] John MacDonnell, *Git for Scientists: A Tutorial*. http://nyuccl.org/pages/gittutorial/, July 2012.

[Oliver2013] Marc Oliver, *Introduction to the Scipy Stack - Scientific Computing Tools for Python*. Jacobs University, http://math.jacobs-university.de/oliver/teaching/scipy-intro/scipy-intro.pdf, November 2013.

[Davidson2010] Andrew Davidson, *Automated tracking of computational experiments using Sumatra*. EuroSciPy 2010, http://www.andrewdavison.info/media/slides/sumatra_euroscipy2010.pdf, 2010.

[Sandve2013] Geir Kjetil Sandve et al, *Ten Simple Rules for Reproducible Computational Research.*. PLoS Comput Biol 9(10): e1003285. doi:10.1371/journal.pcbi.1003285, October 2013.

[Heroux2011] Michael A. Heroux, *Improving CSE Software through Reproducibility Requirements*. SECSE '11 Proceedings of the 4th International Workshop on Software Engineering for Computational Science and Engineering, pp. 28-31, ISBN: 978-1-4503-0598-3 do:10.1145/1985782.1985787, May 2011.

[Bottomley2014] James Bottomley, *What is All the Container Hype?*. Linux Foundation, p. 2, http://www.odin.com/fileadmin/media/hcap/pcs/documents/ParCloudStorage_Mini_WP_EN_042014.pdf, April 2014.

[Melia2014] Ivan Melia et al, *Linux Containers: Why They are in Your Future and What Has to Happen First*. Cisco and RedHat, p.7, https://www.cisco.com/c/dam/en/us/solutions/collateral/data-center-virtualization/openstack-at-cisco/linux-containers-white-paper-cisco-red-hat.pdf, September 2014.

[Hill2015] David Hill, *Parallel Random Numbers, Simulation, Science and reproducibility*. IEEE/AIP - Computing in Science and Engineering, Volume:17, Issue: 4, pp. 66-71.