SciPy2016

Scientific Computing with Python
Austin, Texas • July 11-17

Proceedings of the 15th Python in Science Conference

July 11 - July 17 • Austin, Texas

Sebastian Benthall
Scott Rostrup

PROCEEDINGS OF THE 15TH PYTHON IN SCIENCE CONFERENCE

Edited by Sebastian Benthall and Scott Rostrup.

SciPy 2016
Austin, Texas
July 11 - July 17, 2016

ISSN:2575-9752
https://doi.org/10.25080/Majora-2e17052b-014

ORGANIZATION

Conference Chairs

ARIC HAGBERG, Los Alamos National Laboratory
PRABHU RAMANCHANDRAN, Enthought, Inc. & IIT Bombay

Program

SERGE REY, Arizona State University
NELLE VAROQUAUX, Mines ParisTech, Institut Curie, INSERM

Communications

JULIE KRUGLER HOLLEK, Twitter

Birds of a Feather

AASHISH CHAUDHARY, Kitware
MIKE HEARNE, USGS

Proceedings

SEBASTIAN BENTHALL, University of Califronia - Berkeley
SCOTT ROSTRUP, AI and Robotics Startup

Financial Aid

SCOTT COLLIS, Argonne National Laboratory
ERIC MA, MIT

Tutorials

BEN ROOT, Atmospheric and Environmental Research, Inc.
JUSTIN VINCENT, Google

Sprints

ANDREW COLLETTE, Enthough Inc.
DHARHAS POTHINA, US Army Corps of Engineers, ERDC

Diversity

JULIE KRUGLER HOLLEK, Twitter

Activities

CHRIS NIEMIRA, AOL
RANDY PAFFENROTH, Worcester Polytechnic Institute

Sponsors

JILL COWAN, Enthought

Financial

BILL COWAN, Enthought
JODI HAVRANEK, Enthought

Logistics

JILL COWAN, Enthought
LEAH JONES, Enthought

Proceedings Reviewers

Tom Aldcroft
Ankur Ankan
Dani Arribas-Bel
Kyle Barbary
Sebastian Benthall
Matthias Bussonier
Chris Calloway
Ondrej Certík
Dav Clark
Matthew Collier
Jean Connelly
Ben Darwin
Juan Duque
Carson Farmer
David Folch
Chris Fonnesbeck
Matt Hall
Katy Huff
Hans Peter Langtangen
Dan Lewis
David Lippa
Matthew McCormick
Kyle Niemeyer
Mike Pacer
Randy Paffenroth
Florian Rathgeber
Thomas Robitaille
Matthew Rocklin
Scott Rostrup
Dan Schult
Skipper Seabold
Bill Spotz
Jordan Suchow
Erik Tollerud
Jake Vanderplas
Bryan W. Weber

SCHOLARSHIP RECIPIENTS

FILIPE FERNANDES, SECOORA
ISURU FERNANDO, University of Moratuwa
GILBERT FORSYTH, George Washington University
KESTREL GORLICK, Northern Arizona University
DEBORAH HANUS, Harvard University
MATAR HALLER, UC Berkeley
NOELLE HELD, MIT-WHOI Joint Program in Oceanography
AMIT KUMAR, SymPy
NELSON LIU, University of Washington
TAYLOR OSHAN, Arizona State University/Python Spatial Analysis Library
THOMAS ROBITAILLE, Freelance
SARTAJ SINGH, SymPy
AMAN SINGH, Scipy
LOIS SMITH, University of Michigan
ELIZABETH WICKES, University of Illinois at Urbana-Champaign
PAMELA WU, NYU Medical Center
V. ZACHARY, Golkhou ASU

CONTENTS

Fitting Human Decision Making Models using Python

Alejandro Weinstein[‡§*], Wael El-Deredy[‡§], Stéren Chabert[‡], Myriam Fuentes[‡]

Abstract—A topic of interest in experimental psychology and cognitive neuroscience is to understand how humans make decisions. A common approach involves using computational models to represent the decision making process, and use the model parameters to analyze brain imaging data. These computational models are based on the Reinforcement Learning (RL) paradigm, where an agent learns to make decisions based on the difference between what it expects and what it gets each time it interacts with the environment. In the typical experimental setup, subjects are presented with a set of options, each one associated to different numerical rewards. The task for each subject is to learn, by taking a series of sequential actions, which option maximizes their total reward. The sequence of actions made by the subject and the obtained rewards are used to fit a parametric RL model. The model is fit by maximizing the likelihood of the parameters given the experiment data. In this work we present a Python implementation of this model fitting procedure. We extend the implementation to fit a model of the experimental setup known as the "contextual bandit", where the probabilities of the outcome change from trial to trial depending on a predictive cue. We also developed an artificial agent that can simulate the behavior of a human making decisions under the RL paradigm. We use this artificial agent to validate the model fitting by comparing the parameters estimated from the data with the known agent parameters. We also present the results of a model fitted with experimental data. We use the standard scientific Python stack (NumPy/SciPy) to compute the likelihood function and to find its maximum. The code organization allows to easily change the RL model. We also use the Seaborn library to create a visualization with the behavior of all the subjects. The simulation results validate the correctness of the implementation. The experimental results shows the usefulness and simplicity of the program when working with experimental data. The source code of the program is available at https://github.com/aweinstein/FHDMM.

Index Terms—decision making modeling, reinforcement learning

Introduction

As stated by the classic work of Rescorla and Wagner [Res72]

> *"... organisms only learn when events violate their expectations. Certain expectations are built up about the events following a stimulus complex; expectations initiated by that complex and its component stimuli are then only modified when consequent events disagree with the composite expectation."*

This paradigm allows to use the framework of Reinforcement Learning (RL) to model the process of human decision making. In the fields of experimental psychology and cognitive neuroscience these models are used to fit experimental data. Once such a model is fitted, one can use the model parameters to draw conclusions about individual difference between the participants. The model parameters can be co-varied with imaging data to make observations about the neural mechanisms underpinning the learning process. For an in-depth discussion about the connections between cognitive neuroscience and RL see chapter 16 of [Wie12].

In this work we present a Python program able to fit experimental data to a RL model. The fitting is based on a maximum likelihood approach [Cas02]. We present simulation and experimental data results.

* *Corresponding author: alejandro.weinstein@uv.cl*
‡ *Universidad de Valparaiso, Chile*
§ *Advanced Center for Electrical and Electronic Engineering*

A Decision Making Model

In this section we present the model used in this work to describe how an agent (either an artificial one or a human) learns to interact with an environment. The setup assumes that at the discrete time t the agent selects action a_t from the set $\mathscr{A} = \{1, \ldots, n\}$. After executing that action the agent gets a reward $r_t \in \mathbb{R}$, according to the properties of the environment. Typically these properties are stochastic and are defined in terms of probabilities conditioned by the action. This sequence is repeated T times. The objective of the agent is to take actions to maximize the total reward

$$R = \sum_{t=1}^{T} r_t.$$

In the RL literature, this setup is known as the "n-armed bandit problem" [Sut98].

According to the Q-learning paradigm [Sut98], the agent keeps track of its perceived value for each action through the so called action-value function $Q(a)$. When the agent selects action a_t at time t, it updates the action-value function according to

$$Q_{t+1}(a_t) = Q_t(a_t) + \alpha(r_t - Q_t(a_t)),$$

where $0 \leq \alpha \leq 1$ is a parameter of the agent known as *learning rate*. To make a decision, the agent selects an action at random from the set $\mathscr{A}$ with probabilities for each action given by the softmax rule

$$P(a_t = a) = \frac{e^{\beta Q_t(a)}}{\sum_{i=1}^{n} e^{\beta Q_t(a_i)}},$$

where $\beta > 0$ is a parameter of the agent known as *inverse temperature*.

In this work we consider the case were the probabilities associated to the reward, in addition to being conditioned by the action, are also conditioned by a context of the environment. This context change at each time step and is observed by the agent. This means that the action-value function, the softmax rule, α,

and β also depend on the current context of the environment. In this scenario, the update action-value and softmax rules become

$$Q_{t+1}(a_t, c_t) = Q_t(a_t, c_t) + \alpha_{c_t}(r_t - Q_t(a_t, c_t)) \quad (1)$$

$$P(a_t = a, c_t) = \frac{e^{\beta_{c_t} Q_t(a, c_t)}}{\sum_{i=1}^{n} e^{\beta_{c_t} Q_t(a_i, c_t)}}, \quad (2)$$

where c_t is the cue observed at time t. In the literature, this setup is known as *associative search* [Sut98] or *contextual bandit* [Lan08].

In summary, each interaction, or trial, between the agent and the environment starts by the agent observing the environment context, or cue. Based on that observed cue and on what the agent has learned so far from previous interactions, the agent makes a decision about what action to execute next. It then gets a reward (or penalty), and based on the value of that reward (or penalty) it updates the action-value function accordingly.

Fitting the Model Using Maximum Likelihood

In cognitive neuroscience and experimental psychology one is interested in fitting a decision making model, as the one described in the previous section, to experimental data [Daw11].

In our case, this means to find, given the sequences of cues, actions and rewards

$$(c_1, a_1, r_1), (c_2, a_2, r_2) \ldots, (c_T, a_T, r_T)$$

the corresponding α_c and β_c. The model is fit by maximizing the likelihood of the parameters α_c and β_c given the experiment data. The likelihood function of the parameters is given by

$$\mathscr{L}(\alpha_c, \beta_c) = \prod_{t=1}^{T} P(a_t, c_t), \quad (3)$$

where the probability $P(a_t, c_t)$ is calculated using equations (1) and (2).

Once one has access to the likelihood function, the parameters are found by determining the α_c and β_c that maximize the function. In practice, this is done by minimizing the negative of the logarithm of the likelihood (NLL) function [Daw11]. In other words, the estimate of the model parameters are given by

$$\widehat{\alpha}_c, \widehat{\beta}_c = \underset{0 \le \alpha \le 1, \beta \ge 0}{\operatorname{argmin}} \; -\log(\mathscr{L}(\alpha_c, \beta_c)). \quad (4)$$

The quality of this estimate can be estimated through the inverse of the Hessian matrix of the NLL function evaluated at the optimum. In particular, the diagonal elements of this matrix correspond to the standard error associated to α_c and β_c [Daw11].

Details about the calculation of the likelihood function and its optimization are given in the *Implementation and Results* section.

Experimental Data

The data used in this work consists on the record of a computerized card game played by 46 participants of the experiment. The game consists of 360 trials. Each trial begins with the presentation of a cue during one second. This cue can be a circle, a square or a triangle. The cue indicates the probability of winning on that trial. These probabilities are 20%, 50% and 80%, and are unknown to the participants. The trial continues with the presentation of four cards with values 23, 14, 8 and 3. The participant select one of these cards and wins or loses the amount of points indicated on the selected card, according to the probabilities defined by the cue. The outcome of the trial is indicated by a stimulus that lasts one second (an arrow pointing up for winning and down for losing). The trial ends with a blank inter-trial stimulus that also last one second. Figure 1 shows a schematic of the stimulus presentation. Participants were instructed to maximize their winnings and minimize their losses. See [Mas12] for more details about the experimental design.

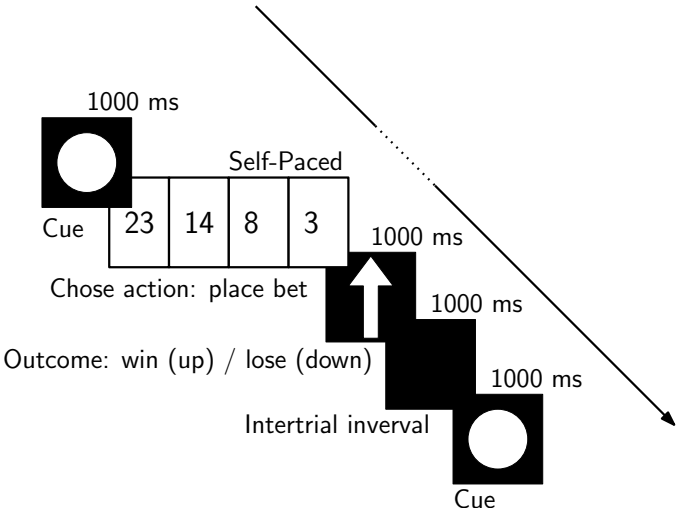

Fig. 1: *Schematic of the stimulus presentation. A trial begins with the presentation of a cue. This cue can be a circle, a square or a triangle and is associated with the probability of winning in that trial. These probabilities are 20%, 50% and 80%, and are unknown to the participants. The trial continues with the presentation of four cards with values 23, 14, 8 and 3. After selecting a card, the participant wins or lose the amount of points indicated on the card, according to the probabilities associated with the cue. The outcome of the trial is indicated by a stimulus, where the win or lose outcome is indicated by an arrow up or down, respectively [Mas12].*

Note that in the context with probability of winning 50% any strategy followed by the subject will produce an expected reward of 0. Thus, there is nothing to learn for this context. For this reason, we do not consider this context in the following analysis.[1]

The study was approved by the University of Manchester research ethics committee. Informed written consent was obtained from all participants.

Implementation and Results

Before testing the experimental data, we present an implementation of an artificial agent that makes decisions according to the decision model presented above. This artificial agent allows us to generate simulated data for different parameters, and then use the data to evaluate the estimation algorithm.

The code for the artificial agent is organized around two classes. The class `ContextualBandit` provides a simulation of the environment. The key two methods of the class are `get_context` and `reward`. The `get_context` method sets the context, or cue, for the trial uniformly at random and returns its value. The `reward` method returns the reward, given the selected action. The value of the reward is selected at random with the probability of winning determined by the current context. The following code snippet shows the class implementation.

1. This condition was included in the original work to do a behavioral study not related to decision making.

```
class ContextualBandit(object):
    def __init__(self):
        # Contexts and their probabilities of
        # winning
        self.contexts = {'punishment': 0.2,
                         'neutral': 0.5,
                         'reward': 0.8}
        self.actions = (23, 14, 8, 3)
        self.n = len(self.actions)
        self.get_context()

    def get_context_list(self):
        return list(self.contexts.keys())

    def get_context(self):
        k = list(self.contexts.keys())
        self.context = np.random.choice(k)
        return self.context

    def reward(self, action):
        p = self.contexts[self.context]
        if np.random.rand() < p:
            r = action
        else:
            r = -action
        return r
```

The behavior of the artificial agent is implemented in the `ContextualAgent` class. The class is initialized with parameters learning rate `alpha` and inverse temperature `beta`. Then, the `run` method is called for each trial, which in turn calls the `choose_action` and `update_action_value` methods. These methods implement equations (2) and (1), respectively. The action-value function is stored in a dictionary of NumPy arrays, where the key is the context of the environment. The following code snippet shows the class implementation.

```
class ContextualAgent(object):
    def __init__(self, bandit, beta, alpha):
        # ...

    def run(self):
        context = self.bandit.get_context()
        action = self.choose_action(context)
        action_i = self.actions[action]
        reward = self.bandit.reward(action_i)
        # Update action-value
        self.update_action_value(context, action,
                                 reward)

    def choose_action(self, context):
        p = softmax(self.Q[context], self.beta)
        actions = range(self.n)
        action = np.random.choice(actions, p=p)
        return action

    def update_action_value(self, context, action,
                            reward):
        error = reward - self.Q[context][action]
        self.Q[context][action] += self.alpha * error
```

The function `run_single_softmax_experiment` shows how these two classes interact:

```
def run_single_softmax_experiment(beta, alpha):
    cb = ContextualBandit()
    ca = ContextualAgent(cb, beta=beta, alpha=alpha)
    trials = 360
    for _ in range(steps):
        ca.run()
```

In this function, after the classes are initialized, the `run` method is run once per trial. The results of the simulation are stored in a pandas dataframe (code not shown). Figure 2 shows an example of

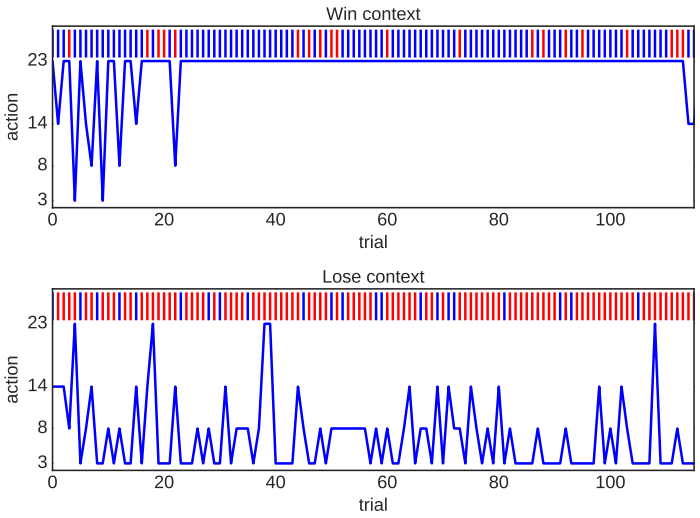

Fig. 2: *Simulation results for an experiment with $\alpha = 0.1$ and $\beta = 0.5$. Actions made by the agent when the context has a probability of winning of 80% (top) and 20% (bottom). The plots also show a vertical bar for each trial indicating if the agent won (blue) or lose (red).*

a simulation for $\alpha = 0.1$ and $\beta = 0.5$ (same value for all contexts). The top and bottom plots show the actions made by the agent when it observes the context with a probability of winning of 80% and 20%, respectively. The plots also show a blue and red vertical bar for each trial where the agent won or lost, respectively. We observe that the agent learned to made actions close to the optimal ones.

The key step in the estimation of the parameters is the computation of the likelihood function described by equation (3). As explained before, for numerical reasons one works with the negative of the likelihood function of the parameters $-\log(\mathscr{L}(\alpha_c, \beta_c))$. The following code snippet describes the steps used to compute the negative log likelihood function.

```
prob_log = 0
Q = dict([[cue, np.zeros(self.n_actions)]
          for cue in self.cues])
for action, reward, cue in zip(actions, rewards, cues):
    Q[cue][action] += alpha * (reward - Q[cue][action])
    prob_log += np.log(softmax(Q[cue], beta)[action])
prob_log *= -1
```

After applying the logarithmic function to the likelihood function, the product of probabilities becomes a sum of probabilities. We initialize the variable `prob_log` to zero, and then we iterate over the sequence (c_t, a_t, r_t) of cues, actions, and rewards. These values are stored as lists in the variables `actions`, `rewards`, and `cues`, respectively. The action value function $Q(a_t, c_t)$ is represented as a dictionary of NumPy arrays, where the cues are the keys of the dictionary. The arrays in this dictionary are initialized to zero. To compute each term of the sum of logarithms, we first compute the corresponding value of the action-value function according to equation (1). After updating the action-value function, we can compute the probability of choosing the action according to equation (2). Finally we multiply the sum of probabilities by negative one.

Once we are able to compute the negative log-likelihood function, to find the model parameter we just need to minimize this function, according to equation (3). Since this is a con-

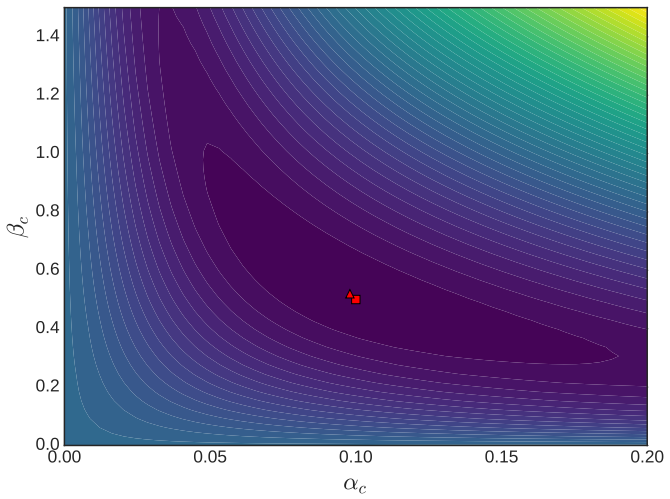

Fig. 3: *Likelihood function of the parameters given the data of the artificial agent for the win context. The data correspond to an agent operating with $\alpha_c = 0.1$ and $\beta_c = 0.5$ (red square). The model parameters estimated using the maximum likelihood are $\widehat{\alpha}_c = 0.098$ and $\widehat{\beta}_c = 0.508$ (red triangle).*

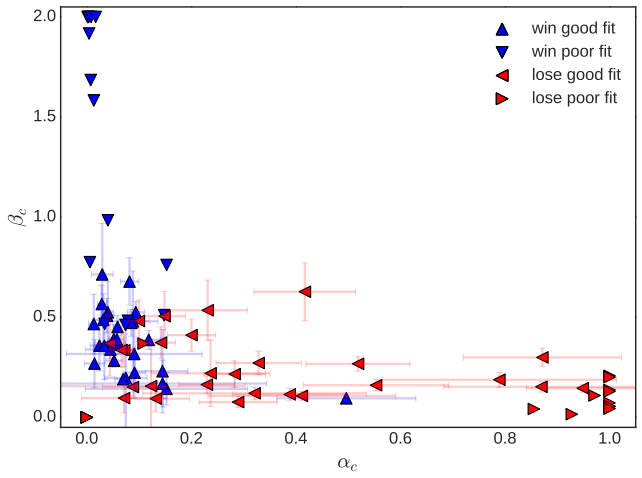

Fig. 4: *Estimated model parameters. Each point shows the estimated $\widehat{\alpha}_c$ and $\widehat{\beta}_c$ for each subject and and context. Blue upside/down triangles are the estimates for the "win context" (probability of winning 80%). Red left/right triangles are the estimates for the "lose context" (probability of winning 20%). We show the standard error for the estimates that are a good fit.*

strained minimization problem, we use the L-BFGS-B algorithm [Byr95], available as an option of the `minimize` function of the `scipy.optimize` module. The following code snippet shows the details.

```
r = minimize(self.neg_log_likelihood, [0.1,0.1],
             method='L-BFGS-B',
             bounds=(0,1), (0,2))
```

This function also computes an approximation of the inverse Hessian matrix evaluated at the optimum. We use this matrix to compute the standard error associated to the estimated parameter.

Before using our implementation of the model estimation method with real data, it is important, as a sanity check, to test the code with the data generated by the artificial agent. Since in this case we know the actual values of the parameters, we can compare the estimated values with the real ones. To run this test we generate 360 trials (same number of trials as in the experimental data) with an agent using parameters $\alpha_c = 0.1$ and $\beta_c = 0.5$. Figure 3 shows the likelihood function of the parameters. Using the maximum likelihood criteria we find the estimated parameters $\widehat{\alpha}_c = 0.098$ and $\widehat{\beta}_c = 0.508$. The actual values of the agent parameters are shown with a red square and the estimated parameters with a red plus sign. This result shows that our implementation is calculating the parameter estimation as expected. The NLL function and the quality of the estimation is similar for other parameter settings.

It is good practice to visualize the raw experimental data before doing any further analysis. In this case, this means showing the actions taken by each subject for each trial. Ideally, we wish to show the behaviors of all the subject for a given context in a single figure, to get an overview of the whole experiment. Fortunately, the Seaborn library [Was16] allows us to do this with little effort. Figure 5 shows the result for the context with a probability of winning of 80%. We also add vertical lines (blue for winning and red for losing) for each trial.

Finally, we can fit a model for each subject. To do this we perform the maximum likelihood estimation of the parameters using the experimental data. Figure 4 shows the estimated $\widehat{\alpha}_c$ and $\widehat{\beta}_c$ for each subject and context. Blue upside/down triangles are the estimates for the "win context" (probability of winning 80%). Red left/right triangles are the estimates for the "lose context" (probability of winning 20%). We show the standard error for the estimates that are a good fit, declared when the standard error is below 0.3 for both $\widehat{\alpha}_c$ and $\widehat{\beta}_c$.

We notice from this result that not all behaviors can be properly fitted with the RL model. This is a known limitation of this model [Daw11]. We also observe that in general the parameters associated with the "lose context" exhibit larger values of learning rate α and smaller values of inverse temperature β. Although at this point of our research it is not clear the reason for this difference, we conjecture that this phenomenon can be explained by two factors. First, in the lose context people bet smaller amounts after learning that the probability of wining is low in this context. This means that the term $(r_t - Q_t(a_t, c_t))$ in equation (1) is smaller compared to the win context. Thus, a larger learning rate is needed to get an update on the action value function of a magnitude similar to the win context.[2] Secondly, it is known that humans commonly exhibit a loss aversion behavior [Kah84]. This can explain, at least in part, the larger learning rates observed for the lose context, since it could be argued that people penalized more their violation of their expectations, as reflected by the term $(r_t - Q_t(a_t, c_t))$ of equation (1), when they were experiencing the losing situation.

In terms of execution time, running a simulation of the artificial agent consisting of 360 steps takes 34 milliseconds; minimizing the NLL function for a single subject takes 21 milliseconds; and fitting the model for all 43 subjects, including loading the experimental data from the hard disk, takes 14 seconds. All these measurements were made using the IPython `%timeit` magic function in a standard laptop (Intel Core i5 processor with 8 gigabytes of RAM).

2. This difference suggests that the experimental design should be modified to equalize this effect between the contexts.

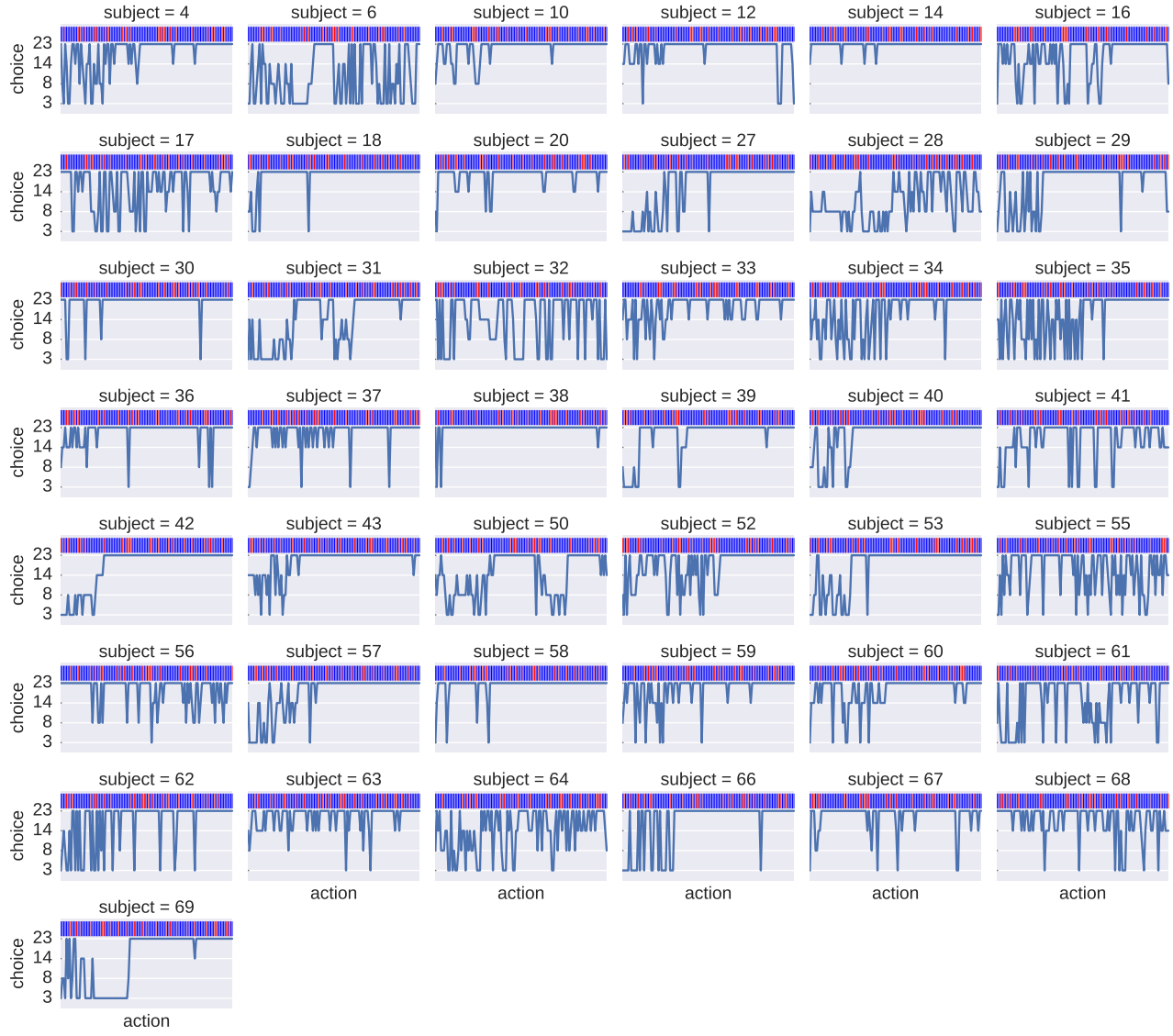

Fig. 5: *Actions taken by all the subjects for trials with context associated to the 80% probability of winning. The vertical bars show if the subject won (blue) or lost (red) in that particular trial.*

Discussion

We have shown a Python program able to fit a decision making model from experimental data, using the maximum likelihood principle. Thanks to Python and the SciPy stack, it was possible to implement this program in a way that we believe is easy to understand and that has a clear correspondence to the theoretical development of the model. We think that the structure of the code allows to easily extend the implementation to test variations in the decision making model presented in this work.

Acknowledgments

We thanks Liam Mason for sharing the experimental data used in this work. This work was supported by the Advanced Center for Electrical and Electronic Engineering, AC3E, Basal Project FB0008, CONICYT.

References

[Byr95] R.Byrd, P. Lu and J. Nocedal. *A Limited Memory Algorithm for Bound Constrained Optimization*, SIAM Journal on Scientific and Statistical Computing 16 (5): 1190-1208, 1995.

[Cas02] G. Casella and R. L. Berger, Statistical Inference. Thomson Learning, 2002.

[Daw11] N. D. Daw, *Trial-by-trial data analysis using computational models*, Decision making, affect, and learning: Attention and performance XXIII, vol. 23, p. 1, 2011.

[Kah84] D. Kahneman and A. Tversky. *Choices, values, and frames.*, American psychologist 39.4, 1984.

[Lan08] J. Langford, and T. Zhang, *The epoch-greedy algorithm for multi-armed bandits with side information*, Advances in neural information processing systems, 2008.

[Mas12] L. Mason, N. O'Sullivan, R. P. Bentall, and W. El-Deredy, *Better Than I Thought: Positive Evaluation Bias in Hypomania*, PLoS ONE, vol. 7, no. 10, p. e47754, Oct. 2012.

[Res72] R. A. Rescorla and A. R. Wagner, *A theory of Pavlovian conditioning: Variations in the effectiveness of reinforcement and nonreinforcement*, Classical conditioning II: Current research and theory, vol. 2, pp. 64–99, 1972.

[Sut98] R. Sutton and A. Barto, Reinforcement Learning. Cambridge, Massachusetts: The MIT press, 1998.

[Wie12] M. Wiering and M. van Otterlo, Eds., Reinforcement Learning, vol. 12. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012.

[Was16] M. Waskom et al. seaborn: v0.7.0 (January 2016). ; DOI: 10.5281/zenodo.45133. Available at: http://dx.doi.org/10.5281/zenodo.45133.

Functional Uncertainty Constrained by Law and Experiment

Andrew M. Fraser‡*, Stephen A. Andrews‡

Abstract—Many physical processes are modeled by unspecified functions. Here, we introduce the *F_UNCLE* project which uses the Python ecosystem of scientific software to develop and explore techniques for estimating such unknown functions and our uncertainty about them. The work provides ideas for quantifying uncertainty about functions given the constraints of both laws governing the function's behavior and experimental data. We present an analysis of pressure as a function of volume for the gases produced by detonating an imaginary explosive, estimating a *best* pressure function and using estimates of *Fisher information* to quantify how well a collection of experiments constrains uncertainty about the function. A need to model particular physical processes has driven our work on the project, and we conclude with a plot from such a process.

Index Terms—python, uncertainty quantification, Bayesian inference, convex optimization, reproducible research, function estimation, equation of state, inverse problems

Introduction

Some tasks require one to quantitatively characterize the accuracy of models of physical material properties which are based on existing theory and experiments. If the accuracy is inadequate, one must then evaluate whether or not proposed experiments or theoretical work will provide the necessary information. Faced with several such tasks, we have chosen to first work on the equation of state (EOS) of the gas produced by detonating an explosive called PBX-9501 because it is relatively simple. In particular Hixson et al. [hixson2000] describe a model form that roughly defines its properties in terms of an unknown one dimensional function (pressure as a function of volume on a special path) and simple constraints. This EOS estimation problem shares the following challenges with many of the other material models that we must analyze:

1) The uncertain object is a *function*. In principal it has an infinite number of degrees of freedom. In order to implement a Bayesian analysis one must define and manipulate probability measures on sets in function space. We do not know how to define a probability measure on sets in function space, and we do not know how to compare the utility of different families of parametric approximations.
2) Understanding the constraints on the unknown function and the connection between it and experimental measurements requires understanding some detailed physics.
3) Simulations of some of the experiments run for more than a few minutes on high performance computers. The job control is unwieldy as are the mechanisms for expressing trial instances of the unknown functions and connecting them to the simulations.

We are organizing our efforts to address those challenges under the title *F_UNCLE* (Functional UNcertainty Constrained by Law and Experiment). We work in two parallel modes as we develop ideas and algorithms. We write code for a surrogate problem that runs in a fraction of a minute on a PC, and we write code for fitting a model to PBX-9501 in a high performance computing environment. Our focus shifts back and forth as we find and resolve problems. As we have progressed, we have found that improving our software practices makes it easier to express ideas, test them on PCs and implement them on the HPCs. In this paper, we introduce the [F_UNCLE] code, the surrogate problem we have developed for the EOS and our analysis of that problem.

We are also using the project to learn and demonstrate *Best Practices for Scientific Computing* (eg, [wilson2014]) and *Reproducible Research* (eg, [fomel2009]). The work is designed to be modular, allowing a wide range of experiments and simulations to be used in an analysis. The code is self documenting, with full docstring coverage, and is converted into online documentation using [sphinx]. Each class has a test suite to allow unit testing. Tests are collected and run using [nose]. Each file is also tested using [pylint] with all default checks enabled to ensure it adheres to Python coding standards, including PEP8. Graphics in this paper were generated using [matplotlib] and the code made use of the [numpy] and [scipy] packages. Among the reasons we chose the Python/SciPy ecosystem, the most important are:

Readable
: Writing in Python helps us implement the most important point in [wilson2014] : "Write programs for people, not computers."

Versatile
: The Python Standard Library lets us easily connect our scripts to other code, eg, submitting HPC jobs and wrapping libraries written in other languages.

Community support
: Because of the large number of other users, it is easy to get answers to questions.

Numerical packages

* *Corresponding author: afraser@lanl.gov*
‡ *XCP-8, Los Alamos National Laboratory*

1. LA-UR-16-23717

We use a host of modules from Numpy, SciPy and other sources.

Portable

With the Python/SciPy ecosystem, it is easy to write code that runs on our desktops and also runs in our HPC environment.

The task of mapping measurements to estimates of the characteristics of models for the physical processes that generated them is called an *inverse problem*. Classic examples include RADAR, tomography and image estimation. Our problems differ from those in the diverse and indirect nature of the measurements, the absence of translation invariance and in the kinds of constraints. [F_UNCLE] uses constrained optimization and physical models with many degrees of freedom to span a large portion of the allowable function space while strictly enforcing constraints. The analysis determines the function that maximizes the *a posteriori* probability (the MAP estimate) using simulations to match K data-sets. We characterize how each experiment constrains our uncertainty about the function in terms of its *Fisher information*.

As a surrogate problem, we have chosen to investigate the equation of state (EOS) for the products-of-detonation of a hypothetical High Explosive (HE). The EOS relates the pressure to the specific volume of the products-of-detonation mixture. We follow traditional practice (eg, [ficket2000]) and constrain the function to be positive, monotonically decreasing and convex. To date we have incorporated two examples of experiments: the detonation velocity of a *rate stick* of HE, and the velocity of a projectile driven by HE. The behavior of both these experiments depend sensitively on the EOS function.

The following sections describe the choices made in modeling the EOS function, the algorithm used for estimating the function and the use of the Fisher information to characterize the uncertainty about the function. Results so far indicate optimization can find good approximations to the unknown functions and that analysis of Fisher information can quantify how various experiments constrain uncertainty about their different aspects. While these preliminary results are limited to an illustration of the ideas applied to synthetic data and simple models, the approach can be applied to real data and complex simulations. A plot from work on estimating the EOS of the high explosive PBX-9501 appear in the concluding section.

Fisher Information and a Sequence of Quadratic Programs

Our analysis is approximately Bayesian and Gaussian. We suppose that:

1) Experiments provide data $x = [x_0, \ldots, x_n]$, where x_k is the data from the k^{th} experiment
2) We have a likelihood function $p_l(x|\theta) = \prod_k p_k(x_k|\theta)$ in which the data from different experiments are conditionally independent given the parameters θ
3) We have a prior on the parameters $p_p(\theta)$

From those assumptions, one can write the *a posteriori* distribution of the parameters as

$$p(\theta|x) = \frac{p_l(x|\theta)p_p(\theta)}{\int p_l(x|\phi)p_p(\phi)d\phi}. \tag{1}$$

Rather than implement Equation (1) exactly, we use a Gaussian approximation calculated at

$$\hat{\theta} \equiv \text{argmax}_\theta p(\theta|x). \tag{2}$$

Since θ does not appear in the denominator on the right hand side of Equation (1), in a Taylor series expansion of the log of the *a posteriori* distribution about $\hat{\theta}$ the denominator only contributes a constant added to expansions of the log of the likelihood and the log of the prior, and

$$\begin{aligned}\log(p(\theta|x)) &= \log\left(\frac{p_l(x|\hat{\theta})p_p(\hat{\theta})}{\int p_l(x|\phi)p_p(\phi)d\phi}\right) \\ &+ \frac{1}{2}(\theta-\hat{\theta})^T\left(\frac{d^2\log(p_l(x|\phi))}{d\phi^2} + \frac{d^2\log(p_p(\phi))}{d\phi^2}\right)_{\phi=\hat{\theta}}(\theta-\hat{\theta}) \\ &+ R \\ &\equiv C + \frac{1}{2}(\theta-\hat{\theta})^T H(\theta-\hat{\theta}) + R.\end{aligned}$$

Dropping the higher order terms in the remainder R in leaves the normal or Gaussian approximation

$$\begin{aligned}\theta|x &\sim \mathcal{N}(\hat{\theta}, \Sigma = H^{-1}) \\ p(\theta|x) &= \frac{1}{\sqrt{(2\pi)^{\dim|\Sigma|}}}\exp\left(-\frac{1}{2}(\theta-\hat{\theta})^{\mathrm{T}}\Sigma^{-1}(\theta-\hat{\theta})\right).\end{aligned}$$

With this approximation, experiments constrain the a posteriori distribution by the second derivative of their log likelihoods. Quoting Wikipedia: "If $p(x|\theta)$ is twice differentiable with respect to θ, and under certain regularity conditions, then the Fisher information may also be written as

$$\mathcal{I}(\theta) = -\mathbb{E}_X\left[\frac{\partial^2}{\partial\theta^2}\log p(X;\theta)\middle|\theta\right]. \tag{3}$$

[...] The Cramér–Rao bound states that the inverse of the Fisher information is a lower bound on the variance of any unbiased estimator"

Our simulated measurements have Gaussian likelihood functions in which the unknown function only influences the mean. Thus we calculate the second derivative of the log likelihood as follows:

$$\begin{aligned}L &\equiv -\frac{1}{2}(x-\mu(\theta))^T\Sigma^{-1}(x-\mu(\theta)) + C \\ \frac{\partial L}{\partial\theta} &= (x-\mu(\theta))^T\Sigma^{-1}\frac{\partial\mu}{\partial\theta} \\ \frac{\partial^2}{\partial\theta^2}L &= -\left(\frac{\partial\mu}{\partial\theta}\right)^T\Sigma^{-1}\left(\frac{\partial\mu}{\partial\theta}\right) + (x-\mu(\theta))^T\Sigma^{-1}\frac{\partial^2\mu}{\partial\theta^2}\end{aligned}$$

and

$$\mathbb{E}_X\frac{\partial^2}{\partial\theta^2}L = -\left(\frac{\partial\mu}{\partial\theta}\right)^T\Sigma^{-1}\left(\frac{\partial\mu}{\partial\theta}\right),$$

because $\Sigma^{-1}\frac{\partial^2\mu}{\partial\theta^2}$ is independent of X and $\mathbb{E}_X(x-\mu(\theta)) = 0$.

Iterative Optimization

We maximize the *log* of the *a posteriori* probability as the objective function which is equivalent to 2. Dropping terms that do not depend on θ, we write the cost function as follows:

$$\begin{aligned}C(\theta) &\equiv -\log(p(\theta)) - \sum_k \log(p(x_k|\theta)) \\ &\equiv \frac{1}{2}(\theta-\mu)^T\Sigma^{-1}(\theta-\mu) - \sum_k \log(p(x_k|\theta)),\end{aligned}$$

where k is an index over a set of independent experiments. We use the following iterative procedure to find $\hat{\theta}$, the *Maximum A posteriori Probability* (MAP) estimate of the parameters:

1) Set $i = 0$ and $\theta_i[j] = \mu[j]$, where i is the index of the iteration and j is index of the components of θ.
2) Increment i
3) Estimate P_i and q_i defined as

$$q_i^T \equiv \frac{d}{d\theta} C(\theta)\Big|_{\theta=\theta_{i-1}}$$
$$P_i \equiv \frac{d^2}{d\theta^2} C(\theta)\Big|_{\theta=\theta_{i-1}}$$

Since the experiments are independent, the joint likelihood is the product of the individual likelihoods and the log of the joint likelihood is the sum of the logs of the individual likelihoods, ie,

$$\begin{aligned} q_i^T &\equiv (\theta_{i-1} - \mu)\Sigma^{-1} + \sum_k \frac{d}{d\theta} \log(p(x_k|\theta)\Big|_{\theta=\theta_{i-1}} \\ &\equiv (\theta_{i-1} - \mu)\Sigma^{-1} + \sum_k q_{i,k}^T \\ P_i &\equiv \Sigma^{-1} + \sum_k \frac{d^2}{d\theta^2} \log(p(x_k|\theta)\Big|_{\theta=\theta_{i-1}} \\ &\equiv \Sigma^{-1} + \sum_k P_{i,k} \end{aligned}$$

where in $P_{i,k}$ and $q_{i,k}$, i is the iteration number and k is the experiment number.
4) Calculate the matrix G_i and the vector h_i to express the appropriate constraints[2].
5) Calculate $\theta_i = \theta_{i-1} + d$ by solving the quadratic program

$$\begin{aligned} &\text{Minimize } \frac{1}{2} d^T P_i d + q^T d \\ &\text{Subject to } G_i d \preceq h_i \end{aligned}$$

where $\preceq$ means that for each component the left hand side is less than or equal to the right hand side.
6) If not converged go back to step 2.

This algorithm differs from modern SQP methods as each QP sub-problem is has no knowledge of the previous iteration. This choice is justified as the algorithm converges in less than 5 outer loop iterations. This unconventional formulation helps accelerate convergence as the algorithm does not need multiple outer loop iterations to obtain a good estimate of the Hessian, as in modern SQP methods.

The assumption that the experiments are statistically independent enables the calculations for each experiment k in to be done independently. In the next few sections, we describe both the data from each experiment and the procedure for calculating $P_{i,k}$ and $q_{i,k}$.

Equation of State

For our surrogate problem, we say that the thing we want to estimate, θ, represents the equation of state (EOS) of a gas. We also say that the state of the gas in experiments always lies on an isentrope[3] and consequently the only relevant data is the pressure as a function of specific volume (cm^3/gram) of the gas.

2. For our surrogate problem, we constrain the function at the last knot to be positive and have negative slope. We also constrain the second derivative to be positive at every knot. See the [F_UNCLE] code and documentation for more details.

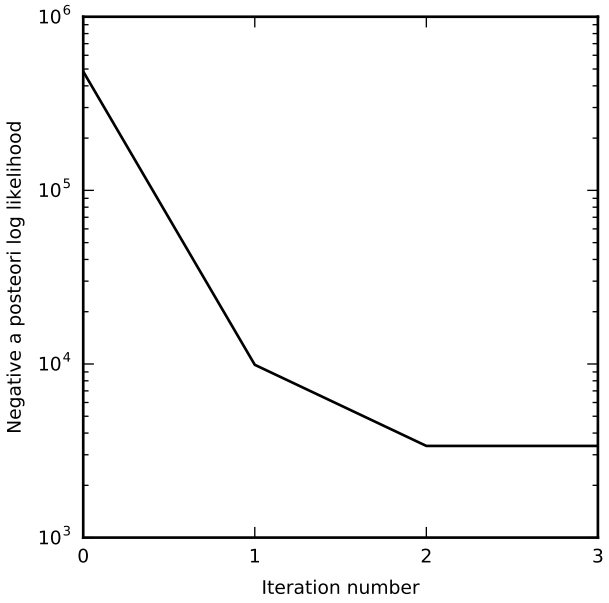

Fig. 1: *Convergence history of a typical solution to the MAP optimization problem*

For physical plausibility, we constrain the function to have the following properties:

- Positive
- Monotonic
- Convex

Here, let us introduce the following notation:

- v Specific volume
- p Pressure
- f An EOS that maps specific volume to pressure, $f : v \mapsto p$.
- v_0 The minimum relevant volume.
- v_1 The maximum relevant volume.
- $\mathscr{F}$ The set of possible EOS functions, $p(v), v_0 \le v \le v_1$

Cubic Splines

While no finite dimensional coordinate scheme can represent every element of $\mathscr{F}$, the flexibility of cubic splines lets us get close to any element of $\mathscr{F}$ using a finite number of parameters. (An analysis of the efficiency of various representations is beyond the scope of this paper.)

Constraining f to be positive and to be a convex function of v is sufficient to ensure that it is also monotonic. Although we are working on a definition of a probability measure on a sets of functions that obeys those constraints and is further constrained by $\frac{|f(v)-\mu_f(v)|}{\mu_f(v)} \le \Delta$, for now, we characterize the prior as Gaussian. As we search for the mean of the a posteriori distribution, we enforce the constraints, and the result is definitely not Gaussian. For the remainder of the present work we ignore that inconsistency and use a prior defined in terms of spline coefficients. We start with a nominal EOS

$$\tilde{f}(v) = \frac{F}{v^3}, \text{ where } F \leftrightarrow 2.56 \times 10^9 \text{Pa at one cm}^3\text{g}^{-1} \qquad (4)$$

3. In an *isentropic* expansion or compression there is no heat conduction. Our isentropic approximation relies on the expansion being so rapid that there is not enough time for heat conduction.

and over a finite domain we approximate it by a cubic spline with coefficients $\{\tilde{c_f}[j]\}$. Thus c, the vector of spline coefficients, is the set of unknown parameters that we have previously let θ denote. Then we assign a variance to each coefficient:

$$\sigma^2[j] = \left(c_f[j]\Delta\right)^2. \tag{5}$$

We set $\Delta = 0.05$. These choices yield:

$$\mu_f \leftrightarrow \{\tilde{c}[j]\}$$
$$\Sigma_f[i,j] = \tilde{\sigma}^2[j]\delta_{i,j}$$

Thus we have the following notation for splines and a prior distribution over $\mathscr{F}$.

- c_f, b_f Vector of coefficients and cubic spline basis functions that define an EOS. We will use $c_f[j]$ and $b_f[j]$ to denote components.
- μ_f, Σ_f Mean and covariance of prior distribution of EOS. In a context that requires coordinates, we let $\mu_f = \left(c_f[0], c_f[1], \ldots, c_f[n]\right)^T$.

The Nominal and True *EOS*

For each experiment, data comes from a simulation using a *true* function and each optimization starts from the nominal EOS which is the mean of the prior given in 4. We've made the *true* EOS differ from the nominal EOS by a sum of Gaussian bumps. Each bump is characterized by a center volume v_k, a width w_k and a scale s_k, with:

$$b_k(v) = \frac{s_k F}{v_k^3} e^{-\frac{(v-v_k)^2}{2w_k^2}}$$

Throughout the remainder of this paper, the *true* EOS that we have used to generate pseudo-experimental data is:

$$f(v) = \frac{F}{v^3} + b_0(v) + b_1(v) \tag{6}$$

where: $v_0 = .4\,\mathrm{cm^3 g^{-1}}$, $w_0 = .1\,\mathrm{cm^3 g^{-1}}$, $s_0 = .25$, $v_1 = .5\,\mathrm{cm^3 g^{-1}}$, $w_1 = .1\,\mathrm{cm^3 g^{-1}}$, and $s_1 = -.3$.

A Rate Stick

The data from this experiment represent a sequence of times that a detonation shock is measured arriving at locations along a stick of HE that is so thick that the detonation velocity is not reduced by curvature. The code for the pseudo data uses the average density and sensor positions given by Pemberton et al. [pemberton2011] for their *Shot 1*.

Implementation

The only property that influences the ideal measurements of rate stick data is the HE detonation velocity. Code in *F_UNCLE.Experiments.Stick* calculates that velocity following Section 2A of Fickett and Davis [ficket2000] (entitled *The Simplest Theory*). The calculation solves for conditions at what is called the *Chapman Jouguet* (CJ) state. The CJ state is defined implicitly by a line (called the *Rayleigh line*) in the (p,v) plane that goes through (p_0, v_0), the pressure and volume before detonation, and (p_{CJ}, v_{CJ}). The essential requirement is that the Rayleigh line be tangent to the isentrope or EOS curve in the (p,v) plane. The slope of the Rayleigh line that satisfies those conditions defines the CJ velocity, V in terms of the following equation:

$$\frac{V^2}{v_0^2} = \frac{p_{CJ} - p_0}{v_0 - v_{CJ}}.$$

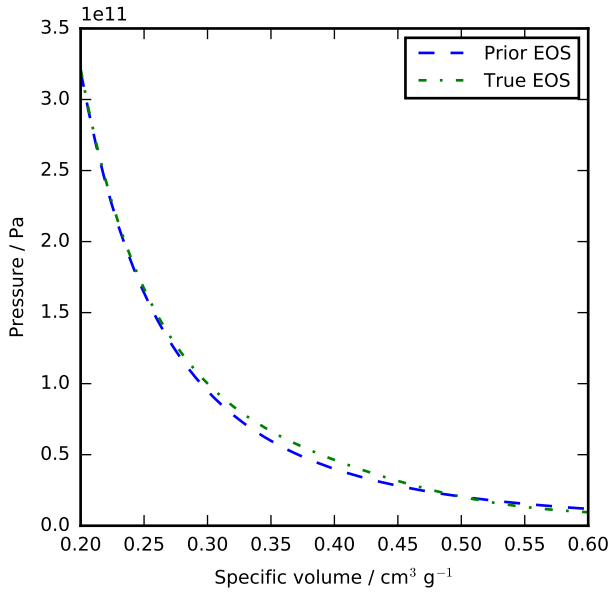

Fig. 2: *The prior and nominal* true *equation of state function. The two models differ most at a specific volume of 0.4 $cm^3 g^{-1}$*

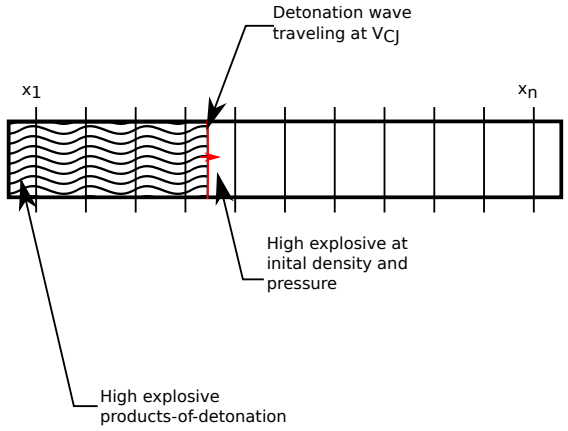

Fig. 3: *The rate stick experiment showing the detonation wave propagating through the rate stick at the CJ velocity. Detonation velocity is measured by the arrival time of the shock at the sensors placed along the stick.*

For each trial EOS, the *F_UNCLE* code uses the *scipy.optimize.brentq* method in a nested loop to solve for (p_{CJ}, v_{CJ}). Figure 4 shows the EOS and both the Rayleigh line and the CJ point that the procedure yields.

Comparison to Pseudo Experimental Data

The previous section explained how to calculate the detonation velocity, $V_{CJ}(f)$, but the *experimental* data are a series of times when the shock reached specified positions on the rate-stick. The simulated detonation velocity is related to these arrival times by:

$$t[i] = \frac{x[i]}{V_{CJ}(f)}.$$

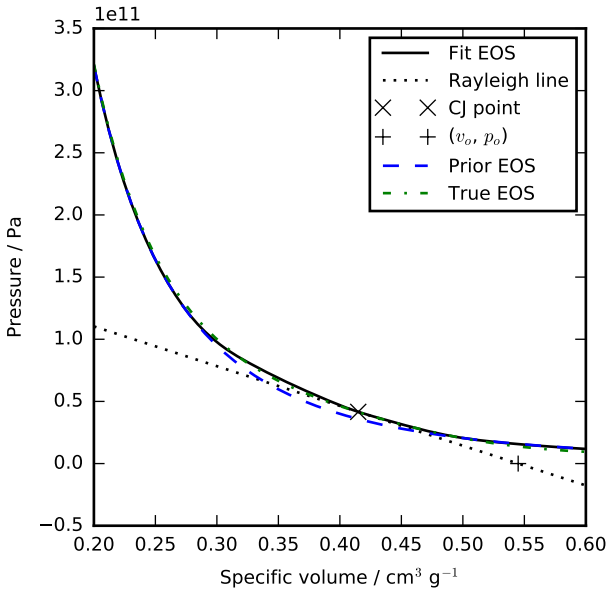

Fig. 4: *Isentropes, a Rayleigh line and the CJ conditions. Starting from the isentrope labeled* Prior EOS *and using data from simulated experiments based on the isentrope labeled* True EOS*, the optimization algorithm described in the Algorithm section produced the estimate labeled* Fit EOS*. Solving for the CJ state of* Fit EOS *isentropes yields a Rayleigh line. The data constrains the isentrope only at* v_{CJ}*.*

where $x[i]$ are the locations of each sensor measuring arrival time.

We let D denote the sensitivity of the set of simulated arrival times to the spline coefficients governing the equation of state, and write:

$$D[i,j] \equiv \frac{\partial t[i]}{\partial c[j]}.$$

We use finite differences to estimate D.

The Gun

The data from this experiment are a time series of measurements of a projectile's velocity as it accelerates along a gun barrel driven by the expanding products-of-detonation of HE. Newton's equation

$$F = ma$$

determines the velocity time series. The product of the pressure from the EOS and the area of the barrel cross section is the force.

Implementation

The position and velocity history of the projectile is generated by the *scipy.integrate.odeint* algorithm. This method solves the differential equation for the projectile position and velocity as it is accelerated along the barrel.

$$\frac{\mathrm{d}x(t)}{\mathrm{d}t} = v(t) \tag{7}$$

$$\frac{\mathrm{d}v(t)}{\mathrm{d}t} = \frac{A}{m_{proj}} f\left(\frac{x(t)A}{m_{HE}}\right) \tag{8}$$

where:

- t is time from detonation (assuming the HE burns instantly)

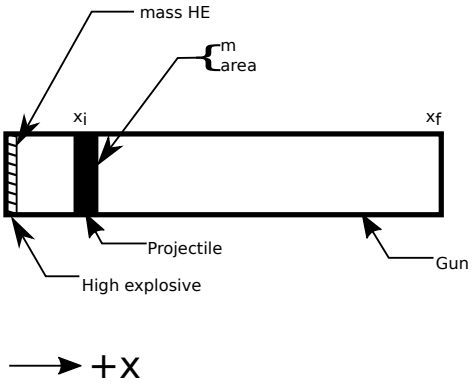

Fig. 5: *The gun experiment. The projectile of a given mass and cross-sectional area is accelerated along the barrel by the expanding products of combustion from the high explosives in the barrel.*

- $x(t)$ is the position of the projectile along the barrel
- $v(t)$ is the velocity of the projectile
- A is the cross-sectional area of the barrel
- m_{HE} is the mass of high explosives
- m_{proj} is the mass of the projectile
- f is the equation of state which relates the pressure to the specific volume of the HE products-of-detonation

The acceleration is computed based the projectile's mass and the force resulting from the uniform pressure acting on the projectile. This pressure is related to the projectile's position by the EOS, assuming that the projectile perfectly seals the barrel so the mass of products-of-detonation behind the projectile remains constant.

Comparison to Pseudo Experimental Data

We generated *experimental data* using our simulation code with the nominal *true* EOS described previously. These experimental data were a series of times and corresponding velocities. To compare the experiments to simulations, which may use a different time discretization, the simulated response was represented by a spline, and was compared to the experiments at each experimental time stamp.

$$D[i,j] = \frac{\partial \hat{v}(t_{exp}[i])}{\partial c_f[j]} \tag{9}$$

where:

- $\hat{v}$ is the velocity given from the spline fit to simulated $v(t)$ data
- t_{exp} is the times where experimental data were available

Numerical Results

The algorithm was applied to the sets of simulation results and pseudo experimental data for both the rate-stick and gun models. Figure 6 shows the improved agreement between the simulated and *experimental* arrival times for the rate-stick after the algorithm adjusts the equation of state. Similar results are shown in Figure 7 for the gun experiment, where the significant error in velocity history at early times is reduced by an order of magnitude with the optimized EOS model.

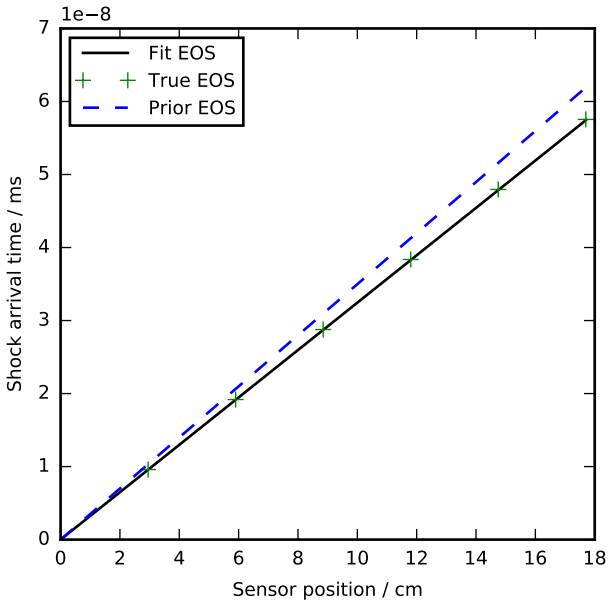

Fig. 6: *Fitting an isentrope to rate stick data. Green +'s denote measured shock arrival time at various positions. The blue line represents the shock velocity calculated from the prior EOS, and the black line is the result of the optimization algorithm described in the text.*

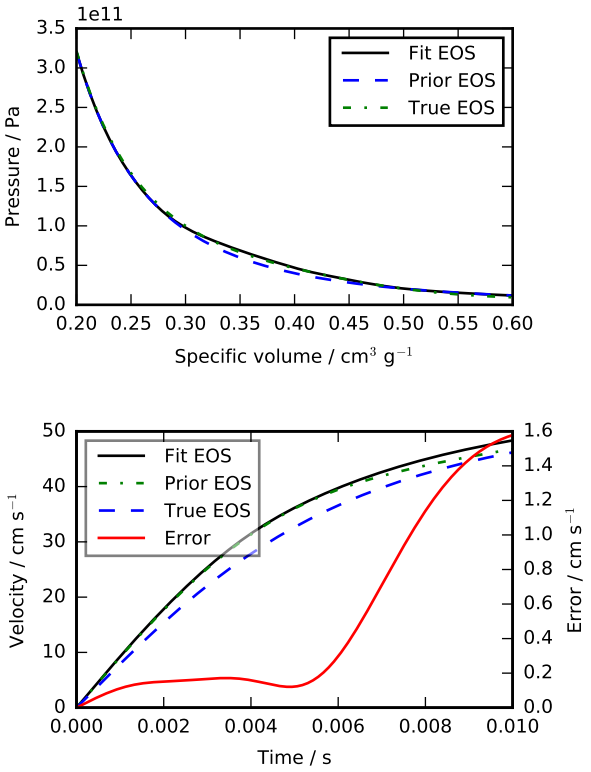

Fig. 7: *Estimation of the maximum* a posteriori *probability parameters of the gun experiment. The* True EOS *appears in the upper plot, and the optimization starts with the* Prior EOS *and ends with* Fit EOS. *The corresponding velocity for the gun as a function of position appears in the lower plot. The estimation also used experimental data from the rate stick.*

Fisher Information Matrix

The Fisher information matrix characterizes how tightly the experimental data constrain the spline coefficients. This matrix can be better understood through a spectral decomposition to show the magnitude of the eigenvalues and the eigenvector behavior.

Figure 8 illustrates the spectral decomposition of the Fisher information matrix for the rate-stick experiment. To machine precision, there is only one nonzero eigenvalue. We expect that because only the CJ point on the EOS influences the forecast data, $\mu(c)$. The eigenvector corresponding to this eigenvalue is most influential about the specific volume corresponding to the CJ state.

The Fisher information matrix of the gun experiment is more complex as changes to the EOS affect the entire time history of the projectile velocity. In Figure 9 There is no clear *dominating* eigenvalue, the largest eigenvalue corresponds to an eigenvector which is more influential at smaller projectile displacements while the next three largest eigenvalues correspond to eigenvectors which are more influential across the range of displacements.

These preliminary investigations of the Fisher information matrix show how this matrix can be informative in describing the uncertainty associated with the optimal EOS function determined by the [F_UNCLE] algorithm. Notice that the eigenvectors of the matrix describe functions that are zero for states not visited by the gun experiment.

Conclusion, Caveats and Future Work

We have described an iterative procedure for estimating functions based on experimental data in a manner that enforces chosen constraints. The [F_UNCLE] code implements the procedure, and we used it to make the figures in the previous sections. The code runs on a modest desktop computer and makes the figures in a fraction of a minute. That speed and simplicity allows one to easily try out new ideas and code. We have relied on the [F_UNCLE] code to guide work with real experimental data and simulations on high performance computers that use proprietary software. Figure 10 is the result of applying the ideas presented here to the physical experiments described in [pemberton2011].

The [F_UNCLE] code has been useful for us, and while we believe it could be useful for others, we emphasize that it is a work in progress. In particular:

- The prior is inconsistent. We hope to analyze and perhaps mitigate the effects of that inconsistency in future work.
- The choice of splines is not justified. We plan to compare the performance of coordinate system options in terms of quantities such as bias and variance in future work.
- The optimization procedure is ad hoc and we have not considered convergence or stability. We have already begun to consider other optimization algorithms.

We have designed the [F_UNCLE] code so that one can easily use it to model any process where there is a simulation which depends on a model with an unknown functional form. The self documenting capabilities of the code and the test suites included with the source code will help others integrate other existing models and simulations into this framework to allow it to be applied to many other physical problems.

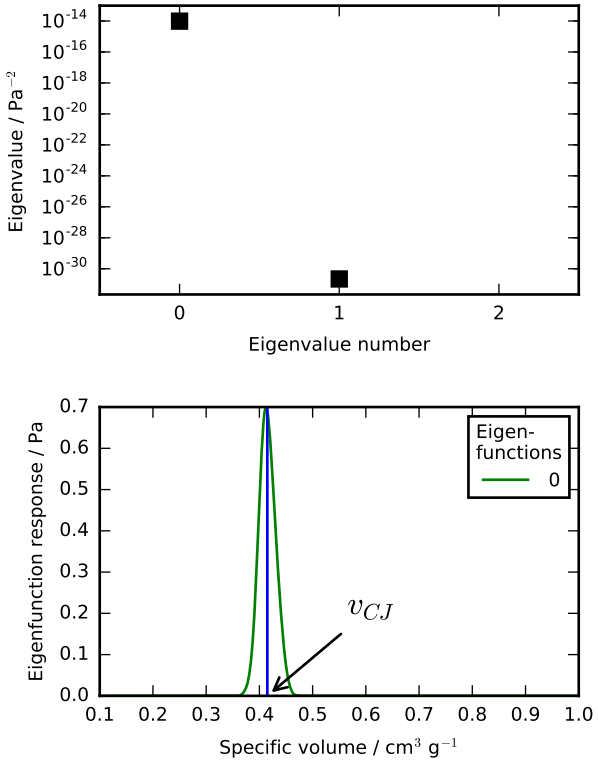

Fig. 8: *Fisher Information of the Rate Stick Experiment. The largest two eigenvalues of $\mathscr{I}(\hat{c})$ appear in the upper plot, and the eigenfunction corresponding to the largest eigenvalue appears in the lower plot.*

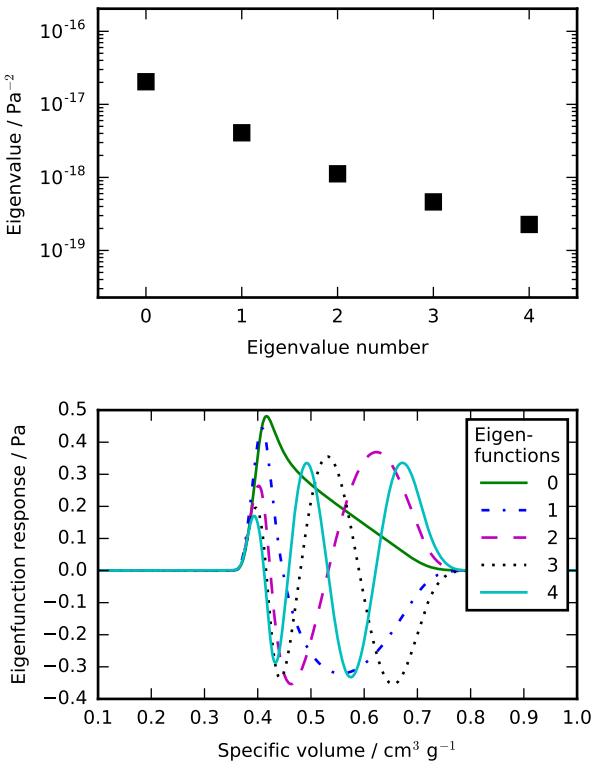

Fig. 9: *Fisher Information of the Gun Experiment. The largest five eigenvalues of $\mathscr{I}(\hat{c})$ appear in the upper plot and the eigenfunctions corresponding to the largest five eigenvalues appear in the lower plot.*

REFERENCES

[cvxopt] Andersen, M. and Vandenberghe, L.. "cvxopt: Convex Optimization Package" http://cvxopt.org/ [Online; accessed 2016-05-27].

[ficket2000] Ficket, W. and Davis, W. C., 2000. "Detonation". University of California Press: Berkeley, CA.

[fomel2009] Fomel, Sergey, and Jon F. Claerbout. "Reproducible research." Computing in Science & Engineering 11.1 (2009): 5-7.

[F_UNCLE] "F_UNCLE: Functional Uncertainty Constrained by Law and Experiment" https://github.com/fraserphysics/F_UNCLE [Online; accessed 2016-05-27].

[hill1997] Hill, L. G., 1997. "Detonation Product Equation-of-State Directly From the Cylinder Test". Proc. 21st Int. Symp. on Shock Waves, Great Keppel Insland, Australia.

[hixson2000] Hixson, R. S. et al., 2000. "Release isentropes of overdriven plastic-bonded explosive PBX-9501." *J. Applied Physics* **88** (11) pp. 6287-6293

[matplotlib] Hunter, J. D.. "Matplotlib: A 2D Graphics Environment", Computing in Science & Engineering, **9**, 90-95 (2007), DOI:10.1109/MCSE.2007.55

[nose] "nose: Nose Extends Unittest to Make Testing Easier" https://pypi.python.org/pypi/nose/1.3.7 [Online; accessed 2016-05-27].

[numpy] van der Walt, S. , Colbert, C. S. and Varoquaux, G.. "The NumPy Array: A Structure for Efficient Numerical Computation", Computing in Science & Engineering, **13**, 22-30 (2011), DOI:10.1109/MCSE.2011.37

[pemberton2011] Pemberton et al. "Test Report for Equation of State Measurements of PBX-9501". LA-UR-11-04999, Los Alamos National Laboratory, Los Alamos, NM.

[pylint] "pylint: Python Code Static Checker" https://www.pylint.org/ [Online; accessed 2016-05-27].

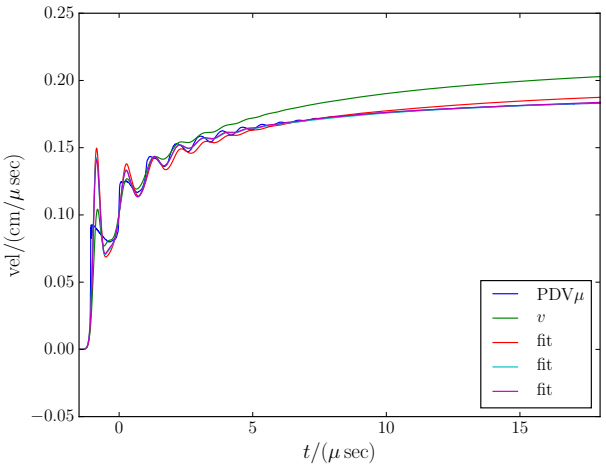

Fig. 10: *Improvement of match between true experiments on PBX-9501 and simulations on a high performance computer. The mean of the experimental data is labeled μ, and the optimization scheme yields the EOSs that produce the traces labeled fit_n.*

[scipy] Jones, E., Oliphant, E., Peterson, P., et al. "SciPy: Open Source Scientific Tools for Python", 2001-, http://www.scipy.org/ [Online; accessed 2016-05-27].

[sphinx] "sphinx: Python Documentation Generator" http://www.sphinx-doc.org/ [Online; accessed 2016-05-27].

[wilson2014] Wilson, Greg, et al. "Best practices for scientific computing." PLoS Biol 12.1 (2014): e1001745.

Composable Multi-Threading for Python Libraries

Anton Malakhov‡*

https://youtu.be/kfQcWez2URE

Abstract—Python is popular among numeric communities that value it for easy to use number crunching modules like [NumPy], [SciPy], [Dask], [Numba], and many others. These modules often use multi-threading for efficient multi-core parallelism in order to utilize all the available CPU cores. Nevertheless, their threads can interfere with each other leading to overhead and inefficiency if used together in one application. The loss of performance can be prevented if all the multi-threaded parties are coordinated. This paper describes usage of Intel® Threading Building Blocks (Intel® TBB), an open-source cross-platform library for multi-core parallelism [TBB], as the composability layer for Python modules. It helps to unlock additional performance for numeric applications on multi-core systems.

Index Terms—Multi-threading, Over-subscription, Parallel Computations, Nested Parallelism, Multi-core, Python, GIL, Dask, Joblib, NumPy, SciPy, Numba, TBB

Motivation

The fundamental shift toward parallelism was loudly declared more than 11 years ago [HSutter] and multi-core processors have become ubiquitous nowadays [WTichy]. However, the adoption of multi-core parallelism in the software world has been slow and Python along with its computing ecosystem is not an exception. Python suffers from several issues which make it suboptimal for parallel processing.

The parallelism with multiple isolated processes is popular in Python but it is prone to inefficiency due to memory-related overhead. On the other hand, multi-threaded parallelism is known to be more efficient but with Python, though it suffers from the limitations of the global interpreter lock [GIL], which prevents scaling of Python programs effectively serializing them. However, when it comes to numeric computations, most of the time is spent in native code where the GIL can easily be released and programs can scale.

Scaling parallel programs is not an easy thing. There are two fundamental laws which mathematically describe and predict scalability of a program: Amdahl's Law and Gustafson-Barsis' Law [AGlaws]. According to Amdahl's Law, speedup is limited by the serial portion of the work, which effectively puts a limit on scalability of parallel processing for a fixed-size job. Python is especially vulnerable to this because it makes the serial part of the same code much slower compared to implementations in some other languages due to its deeply dynamic and interpretative nature. Moreover, the GIL makes things serial often where they potentially can be parallel, further adding to the serial portion of a program.

Gustafson-Barsis' law offers some hope stating that if the problem-size grows along with the number of parallel processors, while the serial portion grows slowly or remains fixed, speedup grows as processors are added. This might relax the concerns regarding Python as a language for parallel computing since the serial portion is mostly fixed in Python when all the data-processing is hidden behind libraries like NumPy and SciPy which are written in other languages. Nevertheless, a larger problem size demands more operational memory to be used for processing it, but memory is a limited resource. Thus, even working with "Big Data", it must be processed by chunks that fit into memory, which puts a limit for the growth of the problem-size. As result, the best strategy to efficiently load a multi-core system is still to fight against serial regions and synchronization.

Nested Parallelism

One way to do that is to expose parallelism on all the possible levels of an application, for example, by making outermost loops parallel or exploring functional or pipeline types of parallelism on the application level. Python libraries that help to achieve this are Dask [Dask], Joblib [Joblib], and even the built-in `multiprocessing` module [mproc] (including its `ThreadPool` class). On the innermost level, data-parallelism can be delivered by Python modules like NumPy [**?**] and SciPy [SciPy]. These modules can be accelerated with an optimized math library like Intel® Math Kernel Library (Intel® MKL) [MKL], which is multi-threaded internally using OpenMP [OpenMP] (with default settings).

When everything is combined together, it results in a construction where code from one parallel region calls a function with another parallel region inside. This is called *nested parallelism*. It is an efficient way for hiding latencies of synchronization and serial regions which are an inevitable part of regular NumPy/SciPy programs.

Issues of Over-subscription

Nevertheless, the libraries named above do not coordinate the creation or pooling of threads, which may lead to *over-subscription*, where there are more active software threads than available hardware resources. It can lead to sub-optimal execution due to frequent context switches, thread migration, broken cache-locality,

* *Corresponding author: Anton.Malakhov@intel.com*
‡ *Intel Corporation*

and finally to a load imbalance when some threads have finished their work but others are stuck, thus halting the overall progress.

For example, OpenMP (used by NumPy/SciPy) may keep its threads active for some time to start subsequent parallel regions quickly. Usually, this is a useful approach to reduce work distribution overhead. Yet with another active thread pool in the application, it impairs better performance because while OpenMP worker threads keep consuming CPU time in busy-waiting loops, the other parallel work cannot start until OpenMP threads stop spinning or are preempted by the OS.

Because overhead from linear over-subscription (e.g. 2x) is not always visible on the application level (especially for small systems), it can be tolerated in many cases when the work for parallel regions is big enough. However, in the worst case a program starts multiple parallel tasks and each of these tasks ends up executing an OpenMP parallel region. This results in quadratic over-subscription (with default settings) which ruins multi-threaded performance on systems with a significant number of threads (roughly more than ten). In some big systems, it may not even be possible to create as many software threads as the number of hardware threads multiplied by itself due to insufficient resources.

Threading Composability

Altogether, the co-existing issues of multi-threaded components define *threading composability* of a program module or a component. A perfectly composable component should be able to function efficiently among other such components without affecting their efficiency. The first aspect of building a composable threading system is to avoid creation of an excessive number of software threads, preventing over-subscription. That effectively means that a component and especially a parallel region cannot dictate how many threads it needs for execution (*mandatory parallelism*). Instead, it should expose available parallelism to a work scheduler (*optional parallelism*), which is often implemented as a user-level work stealing task scheduler that coordinates tasks between components and parallel regions and map them onto software threads. Since such a task scheduler shares a single thread pool among all the program modules and native libraries, it has to be efficient enough to be attractive for high-performance libraries. Otherwise, these libraries will not be able or willing to switch their own threading model to the new scheme.

Intel Solution

Intel's approach to achieve threading composability is to use Intel® Threading Building Blocks (Intel® TBB) library as the common work scheduler, see Figure 1. Intel® TBB is an open-source, cross-platform, mature and recognized C++ library for enabling multi-core parallelism. It was designed for composability, as well as optional and nested parallelism support.

In the Intel® Distribution for Python 2017 Beta and later, as part of Intel® TBB release 4.4 Update 5, I introduce an experimental module which unlocks the potential for additional performance for multi-threaded Python programs by enabling threading composability between two or more thread-enabled libraries. Thanks to threading composability, it can accelerate programs by avoiding inefficient thread allocation as discussed above.

The TBB module implements a `Pool` class with the standard Python interface using Intel® TBB which can be used to replace

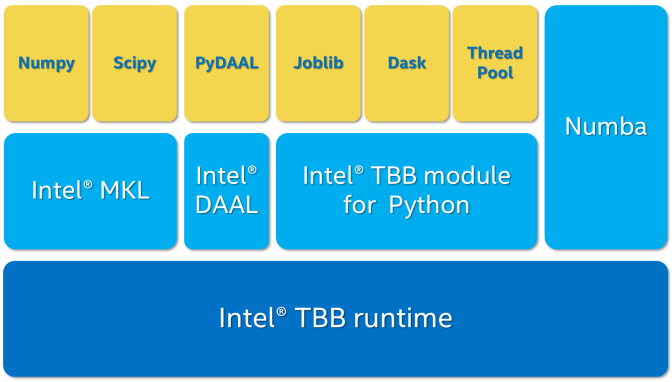

Fig. 1: *Intel® Threading Building Blocks is used as a common runtime for different Python modules.*

Python's `ThreadPool`. Python allows users to dynamically replace any object (e.g. class or function) at runtime (*monkey patching*). Thanks to this technique implemented in class `Monkey`, no source code change is needed in order to enable single thread pool across different Python modules. The TBB module also switches Intel® MKL to use TBB-based threading layer, which automatically enables composable parallelism [ParUniv] for NumPy and SciPy calls.

Usage example

For our first experiment, we need Intel® Distribution for Python [IntelPy] to be installed along with the Dask [Dask] library which simplifies parallelism with Python.

```
# install Intel(R) Distribution for Python
<path to installer of the Distribution>/install.sh
# setup environment
source <path to the Distribution>/bin/pythonvars.sh
# install Dask
conda install dask
```

The code below is a simple program using NumPy that validates QR decomposition by multiplying computed components and comparing the result against the original input:

```
import time, numpy as np
x = np.random.random((100000, 2000))
t0 = time.time()
q, r = np.linalg.qr(x)
test = np.allclose(x, q.dot(r))
assert(test)
print(time.time() - t0)
```

And here is the same program using Dask:

```
import time, dask, dask.array as da
x = da.random.random((100000, 2000),
                     chunks=(10000, 2000))
t0 = time.time()
q, r = da.linalg.qr(x)
test = da.all(da.isclose(x, q.dot(r)))
assert(test.compute()) # threaded
print(time.time() - t0)
```

Here, Dask splits the array into 10 chunks and processes them in parallel using multiple threads. However, each Dask task executes the same NumPy matrix operations which are accelerated using Intel® MKL under the hood and thus multi-threaded by default. This combination results in nested parallelism, i.e. when one parallel component calls another component, which is also threaded.

The reason why the Dask version was set to have only 10 tasks is to model real-life applications with limited parallelism

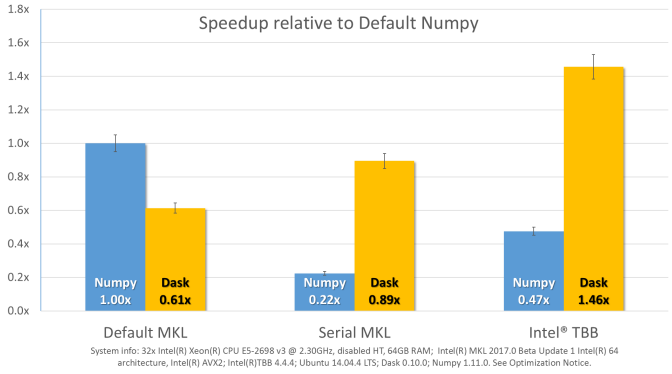

Fig. 2: *Execution times for QR validation example.*

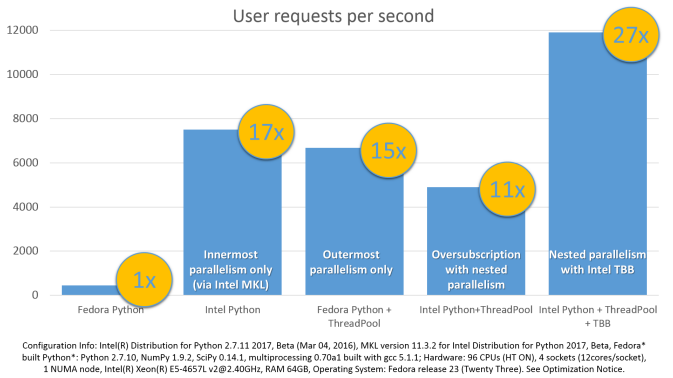

Fig. 3: *Case study results: Generation of User Recommendations.*

on the outermost level, which is quite typical for functional and pipeline types of parallelism. Such cases might benefit the most from enabling parallelism at inner levels of the code. In the case when the top-level parallelism can load all the available cores and is well-balanced, nested parallelism is not that likely to improve performance (but can make it much worse without a composable threading solution).

Here is an example of running the benchmark program in three different modes:

```
1 python bench.py                    # Default MKL
2 OMP_NUM_THREADS=1 python bench.py  # Serial MKL
3 python -m TBB bench.py             # Intel TBB mode
```

Figure 2 shows performance results acquired on a 32-core (no hyper-threading) machine with 64GB memory. The results presented here were acquired with cpython v3.5.1; however, there is no performance difference with cpython v2.7.1. The Dask version runs slower than the NumPy version with the default setting because 10 outermost tasks end up calling 10 OpenMP-based parallel regions that create 10 times more threads than available hardware resources.

The second command runs this benchmark with innermost OpenMP parallelism disabled. It results in the worst performance for the NumPy version since everything is now serialized. Moreover, the Dask version is not able to close the gap completely since it has only 10 tasks, which can run in parallel, while NumPy with parallel MKL is able to utilize the whole machine with 32 threads.

The last command demonstrates how Intel® TBB can be enabled as the orchestrator of multi-threaded modules. The TBB module runs the benchmark in the context of `with TBB.Monkey():` which replaces the standard Python *ThreadPool* class used by Dask and also switches MKL into TBB mode. In this mode, NumPy executes in more than twice the time compared to the default NumPy run. This happens because TBB-based threading in MKL is new and not as optimized as the OpenMP-based MKL threading implementation. However despite that fact, Dask in TBB mode shows the best performance for this benchmark, 46% improvement compared to default NumPy. This happens because the Dask version exposes more parallelism to the system without over-subscription overhead, hiding latencies of serial regions and fork-join synchronization in MKL functions.

1. For more complete information about compiler optimizations, see our Optimization Notice [OptNote]

Case study

The previous example was intentionally selected to be small enough to fit into this paper with all the sources. Here is another case study [FedLitC] that is closer to real-world applications. It implements a recommendation system similar to the ones used on popular web-sites for generating suggestions for the next application to download or the next movie to watch. However, the core of the algorithm is still quite simple and spends most of the time in matrix multiplication. Figure 3 shows results collected on an older machine with a bigger number of cores.

The leftmost result in Figure 3 was acquired on pure, non-accelerated Python that comes by default on Fedora 23. It is used as the base of comparison. Running the same application without modifications with Intel® Distribution for Python results in a 17 times speedup. One reason for this performance increase is that Intel® MKL runs computations in parallel. Thus, for the sake of experiment, outermost parallelism was implemented on the application level processing different user requests in parallel. For the same system-default python, the new version helped to close the gap with the MKL-based version though not completely: executing 15 times faster than the base. However, running the same parallel application with the Intel Distribution resulted in worse performance (11x). This is explained by overhead induced by over-subscription.

In order to remove overhead, the previous experiment was executed with the TBB module on the command line. It results in the best performance for the application - 27 times speedup over the base.

Numba

NumPy and SciPy provide a rich but fixed set of mathematical instruments accelerated with C extensions. However, sometimes one might need non-standard math to be as fast as C extensions. That's where Numba [Numba] can be efficiently used. Numba is a Just-In-Time compiler (JIT) based on LLVM [LLVM]. It aims to close the gap in performance between Python and statically typed, compiled languages like C/C++, which also have popular implementation based on LLVM.

Numba implements the notion of universal functions (ufunc, a scalar function which can be used for processing arrays as well) defined in SciPy [ufunc] and extends it to a computation kernel that can be not only mapped onto arrays but can also spread the work across multiple cores. The original Numba version implements it using a pool of native threads and a simple work-sharing scheduler, which coordinates work distribution between

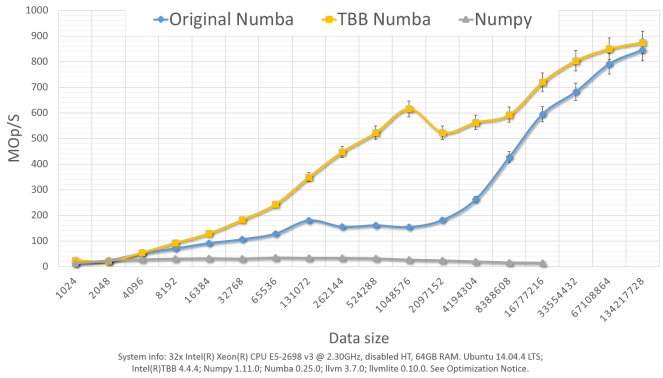

Fig. 4: *Black Scholes benchmark running with Numba on 32 threads.*

them. If used in a parallel numeric Python application, it adds a third thread pool to the existing threading mess described in previous sections. Thus, our strategy was to put it on top of the common Intel® TBB runtime as well.

The original version of Numba's multi-threading runtime was replaced with a very basic and naive implementation based on TBB tasks. Nevertheless, even without nested parallelism and advanced features of Intel® TBB such as work partitioning algorithms, it resulted in improved performance.

Figure 4 shows how original Numba and TBB-based versions perform with the Black Scholes [BSform] benchmark implemented with Numba. Whether the problem size is small or big, they work at almost the same speed. However, TBB-based Numba performs up to 3 or 4 times faster for the problem sizes in between.

The following code is a simplified version of this benchmark that gives an idea how to write parallel code using Numba:

```
import numba as nb, numpy.random as rng
from math import sqrt, log, erf, exp

@nb.vectorize('(f8,f8,f8,f8,f8)',target='parallel')
def BlackScholes(S, X, T, R, V):
    VqT = V * sqrt(T)
    d1 = (log(S / X) + (R + .5*V*V) * T) / VqT
    d2 = d1 - VqT
    n1 = .5 + .5 * erf(d1 * 1./sqrt(2.))
    n2 = .5 + .5 * erf(d2 * 1./sqrt(2.))
    eRT  = exp(-R * T)
    return S * n1 - X * eRT * n2 # Call price
    # Put price = (X * eRT * (1.-n2) - S * (1.-n1))

price  = rng.uniform(10., 50., 10**6) # array
strike = rng.uniform(10., 50., 10**6) # array
time   = rng.uniform(1.0, 2.0, 10**6) # array
BlackScholes(price, strike, time, .1, .2)
```

Here is the scalar function `BlackScholes`, consisting of many elementary and transcendental operations, which is applied (*broadcasted*) by Numba to every element of the input arrays. Additionally, `target='parallel'` specifies to run the computation using multiple threads. The real benchmark also computes the put price using `numba.guvectorize`, uses an approximated function instead of `erf()` for better SIMD optimization, optimizes the sequence of math operations for speed, and repeats the calculation multiple times.

Limitations and Future Work

Intel® TBB does not work well for blocking I/O operations because it limits the number of active threads. It is applicable only for tasks, which do not block in the operating system. If your program uses blocking I/O, please consider using asynchronous I/O that blocks only one thread for the event loop and so prevents other threads from being blocked.

The Python module for Intel® TBB is in an experimental stage and might be not sufficiently optimized and verified with different use-cases. In particular, it does not yet use the master thread efficiently as a regular TBB program is supposed to do. This reduces performance for small workloads and on systems with small numbers of hardware threads.

As was discussed above, the TBB-based implementation of Intel® MKL threading layer is yet in its infancy and is therefore suboptimal. However, all these problems can be eliminated as more users will become interested in solving their composability issues and Intel® MKL and the TBB module are further developed.

Another limitation is that Intel® TBB only coordinates threads inside a single process while the most popular approach to parallelism in Python is multi-processing. Intel® TBB survives in an oversubscribed environment better than OpenMP because it does not rely on the particular number of threads participating in a parallel computation at any given moment, thus the threads preempted by the OS do not prevent the computation from making an overall progress. Nevertheless, it is possible to implement a cross-process mechanism to coordinate resources utilization and avoid over-subscription.

A different approach is suggested by the observation that a moderate over-subscription, such as from two fully subscribed thread pools, does not significantly affect performance for most use cases. In this case, preventing quadratic over-subscription from the nested parallelism (in particular, with OpenMP) can be a practical alternative. Therefore, the solution for that can be as simple as "Global OpenMP Lock" (GOL) or a more elaborate inter-process semaphore that coordinates OpenMP parallel regions.

Conclusion

This paper starts with substantiating the necessity of broader usage of nested parallelism for multi-core systems. Then, it defines threading composability and discusses the issues of Python programs and libraries which use nested parallelism with multi-core systems, such as GIL and over-subscription. These issues affect performance of Python programs that use libraries like NumPy, SciPy, Dask, and Numba.

The suggested solution is to use a common threading runtime library such as Intel® TBB which limits the number of threads in order to prevent over-subscription and coordinates parallel execution of independent program modules. A Python module for Intel® TBB was introduced to substitute Python's ThreadPool implementation and switch Intel® MKL into TBB-based threading mode, which enables threading composability for mentioned Python libraries.

The examples referred in the paper show promising results, where, thanks to nested parallelism and threading composability, the best performance was achieved. In particular, QR decomposition example is faster by 46% comparing to the baseline implementation that uses parallelism only on the innermost level. This result was confirmed by the case study of a recommendation system where 59% increase was achieved for the similar base.

2. For more complete information about compiler optimizations, see our Optimization Notice [OptNote]

And finally, Intel® TBB was proved as a mature multi-threading system by replacing threading runtime implemented in Numba and achieving more than 3 times speedup on several problem sizes.

Intel® TBB along with the Python module are available in open-source [TBB] for different platforms and architectures while Intel® Distribution for Python accelerated with Intel® MKL is available for free as a stand-alone package [IntelPy] and on anaconda.org/intel channel.

REFERENCES

[NumPy] NumPy, http://www.numpy.org/

[SciPy] SciPy, https://www.scipy.org/

[Dask] Dask, http://dask.pydata.org/

[Numba] Numba, http://numba.pydata.org/

[TBB] Intel(R) TBB open-source site, https://www.threadingbuildingblocks.org/

[HSutter] Herb Sutter, "The Free Lunch Is Over", Dr. Dobb's Journal, 30(3), March 2005. http://www.gotw.ca/publications/concurrency-ddj.htm

[WTichy] Walter Tichy, "The Multicore Transformation", Ubiquity, Volume 2014 Issue May, May 2014. DOI: 10.1145/2618393. http://ubiquity.acm.org/article.cfm?id=2618393

[GIL] David Beazley, "Understanding the Python GIL", PyCON Python Conference, Atlanta, Georgia, 2010. http://www.dabeaz.com/python/UnderstandingGIL.pdf

[AGlaws] Michael McCool, Arch Robison, James Reinders, "Amdahl's Law vs. Gustafson-Barsis' Law", Dr. Dobb's Parallel, October 22, 2013. http://www.drdobbs.com/parallel/amdahls-law-vs-gustafson-barsis-law/240162980

[mproc] Python documentation on *multiprocessing*, https://docs.python.org/library/multiprocessing.html

[Joblib] Joblib, http://pythonhosted.org/joblib/

[OpenMP] The OpenMP(R) API specification for parallel programming, http://openmp.org/

[MKL] Intel(R) MKL, https://software.intel.com/intel-mkl

[ParUniv] Vipin Kumar E.K. *A Tale of Two High-Performance Libraries*, The Parallel Universe Magazine, Special Edition, 2016. https://software.intel.com/intel-parallel-universe-magazine

[IntelPy] Intel(R) Distribution for Python, https://software.intel.com/python-distribution

[FedLitC] Alexey Fedotov, Vasilij Litvinov, "Faster, Python!" (in Russian), CodeFest, Novosibirsk, 2016 http://2016.codefest.ru/lecture/1117

[LLVM] The LLVM Compiler Infrastructure, http://llvm.org/

[ufunc] Universal functions (ufunc), SciPy documentation http://docs.scipy.org/doc/numpy/reference/ufuncs.html

[BSform] Fischer Black, Myron Scholes, "The Pricing of Options and Corporate Liabilities", Journal of Political Economy 81 (3) 1973: 637-654. doi:10.1086/260062

[OptNote] https://software.intel.com/en-us/articles/optimization-notice

Generalized earthquake classification

Ben Lasscock[‡*]

https://youtu.be/uT0Nkf0BA7o

Abstract—We characterize the source of an earthquake based on identifying the nodal lines of the radiation pattern it produces. These characteristics are the mode of failure of the rock (shear or tensile), the orientation of the fault plane and direction of slip. We will also derive a correlation coefficient comparing the source mechanisms of different earthquakes. The problem is formulated in terms of a simple binary classification on the surface of the sphere. Our design goal was to derive an algorithm that would be both robust to misclassification of the observed data and suitable for online processing. We will then go on to derive a mapping which translates the learned solution for the separating hyper-plane back to the physics of the problem, that is, the probable source type and orientation. For reproducibility, we will demonstrate our algorithm using the example data provided with the HASH earthquake classification software, which is available online.

Index Terms—machine learning, earthquake, hazard, classification.

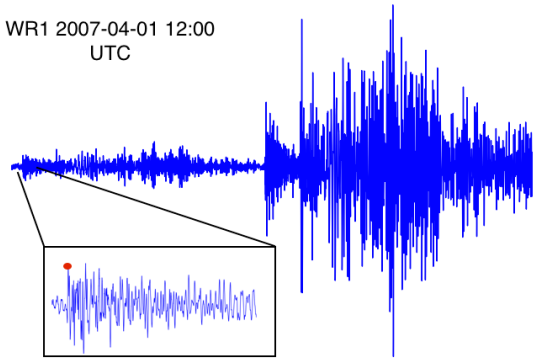

Fig. 1: *A seismogram measured at the WR1 node of the Warramunga seismic array showing displacement due to an aftershock of the April 2007 Solomon Islands earthquake. The red dot indicates the first break motion. The data was obtained by querying the IRIS database http://ds.iris.edu/ds/nodes/dmc/data/types/events/.*

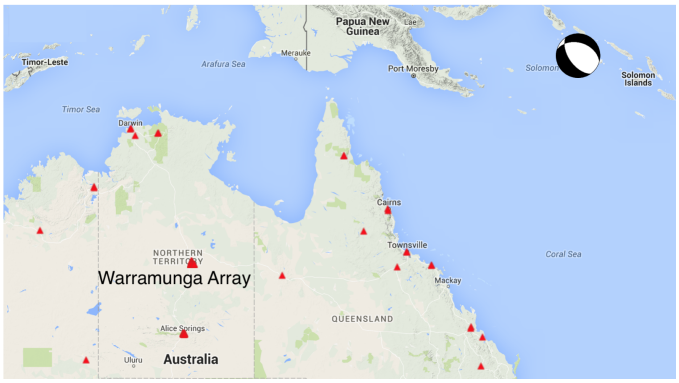

Fig. 2: *A portion of the Australian seismic network showing the location of the Warramunga seismic array, the map was obtained from http://www.fdsn.org/networks/detail/AU/.*

Introduction

In this paper we are going to explain how to classify earthquake data using a support vector classifier (SVC) and then how to interpret the result physically. We will be drawing on the scikit-learn [sklearn] project for the SVC, the ObsPy seismological Python package [ObsPy] for some utility routines and mplstereonet [mplstereonet], which is a matplotlib [mpl] plugin for visualization.

Much of the discussion will center around deriving a mapping from the solution of the SVC to the physical process that originated the earthquake. The key concept we will be elaborating on is understanding the relationship between what we call the input and feature spaces of the SVC. The results of the classification are curves separating points on the surface of the focal sphere (the input space), which is the domain of the input data. However, the physics and understanding of the result lies in the representation of the solution in the feature space, which a higher dimensional space where the classifier may linearly separate the data.

For the sake of reproducibility, the demonstration will use the same dataset provided with the US Geological Survey (USGS) HASH software. HASH [HASH] is an earthquake classification code provided by the USGS and it is built upon an earlier package called FPFIT, which implements a least squares classifier. For each case we will be comparing and contrasting our solutions with those generated by HASH, which we generally expect to be similar.

Our discussion will include annotated software explaining the important steps in the computation. We will be contributing software[1] to reproduce the results of this paper.

* *Corresponding author: blasscoc@gmail.com*
‡ *Geotrace Technologies*

1. https://github.com/blasscoc/FocalMechClassifier

Problem statement

Consider a particular example where we would apply the analysis presented in this paper. In Fig. 1, we show a seismogram (recording) of an earthquake that was located in Solomon Islands. The black and white glyph is a graphical representation of the type of focal mechanism. The orientation of the nodal lines of this glyph displays the orientation of the fault plane. The recording shown was made by the WR1 node of the Warramunga seismic array, which is part of the Australian seismic network, shown partially in Fig. 2. The goal of hazard monitoring in this case would be to characterize the deformation of the earth (focal mechanism) that caused this earthquake. This analysis would involve first locating the source spatially and then classifying its focal mechanism, and importantly the orientation of the fault plane. The orientation of the fault plan is important because displacement of the sea-floor can cause the formation of tsunamis. The algorithm discussed in this paper provides the analysis of the focal mechanism, with the extension that we can also compare the spectrum of the solution with a past events located in this area. This additional information may be useful for decision making regarding what action should be taken given the risk of a historical (or perhaps modeled) scenario repeating.

We proceed by detailing exactly the parameters of the problem at hand. The raw data are recordings of the initial arrival of energy from this earthquake, measured across a seismic network. From each recording, the initial displacement (or first motion) is identified (or picked), as shown by a red dot in Fig. 2. Consider this is as a radiation amplitude from the earthquake measured at a particular location. Further measurements across the seismic network begin to inform the shape of the radiation pattern created by the event. However, a radiation pattern measured far from the event becomes distorted because of the refraction of the seismic wave as it propagates through the earth. To remove this distortion, this energy must migrated, along an estimated ray path, back to the neighborhood of the estimated source location . We call this neighborhood the focal sphere. The process of picking, locating and migrating seismic events is beyond the scope of this paper. However, seismograms can be requested from the IRIS database[2] and a suite of Python tools for processing this data is made available by the ObsPy [ObsPy] Python project.

The input data to our analysis is the polarity (signed amplitude) of the picks, and the azimuth and co-latitude of the observation migrated onto the focal sphere.The design goal is to provide an online tool for characterizing the source mechanism. The emphasis is on robustness of the algorithm, without the need for post facto processing of the data. We also need a system that provides natural metrics of similarity between seismic events.

Physically, the initial arrival of energy will be in the form of a compressional wave. The amplitude of these compressional waves are solutions to the scalar wave equation, which are the spherical harmonic functions[3]. Hence any function that classifies the polarity data should be a superposition of these spherical harmonics. We will learn this classifying function using the SVC. However, it is the spectral representation (harmonic content) of the radiation pattern that contains the physical meaning of the solution.

2. http://www.iris.edu

3. http://docs.scipy.org/doc/scipy/reference/generated/scipy.special.sph_harm.html

Source	(Fault normal/slip)	Template
Shear	(31) + (13)	$-i(Y_{12}+Y_{-12})$
Tensile	(3)	$\alpha Y_{00}+4\sqrt{5}Y_{02}$
Tangential	(3)	$Y_{02}-\frac{i}{2}(Y_{22}+Y_{-22})$.

TABLE 1: *Describes the angular variation of the displacement due to three types of earthquake sources in terms of a basis of spherical harmonic functions. The source templates summarized are shear, tensile and tangential dislocation. The brackets $(\cdot,\cdot)$ define the template direction of the fault normal and the direction of slip in rectangular coordinates. The constant $\alpha = 2+3\frac{\lambda}{\mu}$, where λ and μ are the first Lamé parameter and the shear modulus respectively.*

In Sec. Theory we will review the basic results we need from the theory of seismic sources. In Sec. Existing Least Squares Methods we will review existing methods for classifying earthquake data. The Sec. Earthquake - Learning with Kernels reviews the Python code used in the classification, and derives a mapping between the input space of the problem, to the feature space (represented by the spectrum). In Sec. Physical Interpretation we translate this spectral representation back to the physics of the problem, and explain how to evaluate the correlation metric. In Sec. Discussion we provide an example of the analysis and then we wrap things up with Sec. Conclusions.

Theory

The observed displacement created by the collective motion of particles along a fault plane is described by the theory of seismic sources. We will not go into all the details here, but the reference on seismic source theory we follow is Ben-Menahem and Singh [Ben81]. The key result we will draw upon is a formula for the displacement for various types of seismic sources summarized in Table 4.4 of [Ben81], which is presented in terms of Hansen vectors. Physically, a shear type failure would represent the slip of rock along the fault plane and a tensile failure would represent cracking of the rock. The results of [Ben81] are general, however we are only modeling the angular variation of the displacement due to the compressional wave measured radially to the focal sphere. From this simplification we can translate solutions of [Ben81] into solutions for just the angular variation using the basis of spherical harmonic functions, which we tabulate in Table 1. Notes on translating between [Ben81] and Table 1 are summarized in the Appendix. This result gives us an analytical expression for the spectral content of seismic sources given a certain orientation of the fault plane. We will use this information to find general solutions in Sec. Physical Interpretation.

The amplitude of the radiation pattern cannot typically be migrated back to the location of the event unless an accurate model of seismic attenuation is available, which is not generally the case, even in commercial applications. However, supposing the source type and orientation were known, then the sign of this radiation pattern is a function that must classify the polarity data on the focal sphere. As an example, in Fig. 3 we render in, 3-dimensions, the signed radiation pattern predicted for shear and tensile source, in a particular orientation.

The black areas of this beachball diagram represents the region where the displacement at the source is radially outward (vice versa for the white regions). The nodal lines represent the separating margin between classes of data (outward and inward displacement). For the shear source, the nodal lines are called the fault and auxiliary planes respectively.

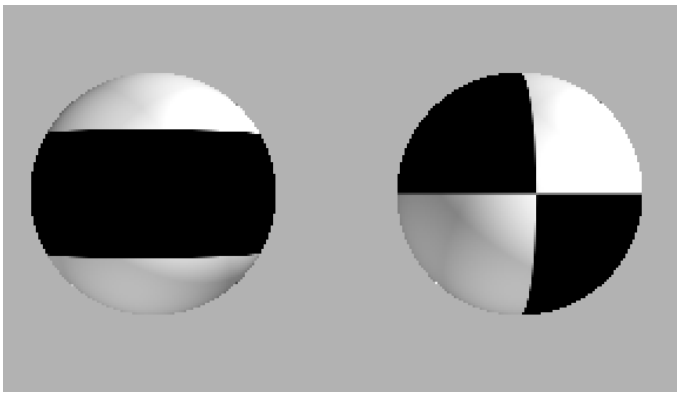

Fig. 3: *Rendered in 3-dimensions, (left) the signed radiation pattern for a possible tensile type source. (right) Similarly for the case of shear type source. Figures are generated using SciPy's spherical harmonic functions and Mayavi.*

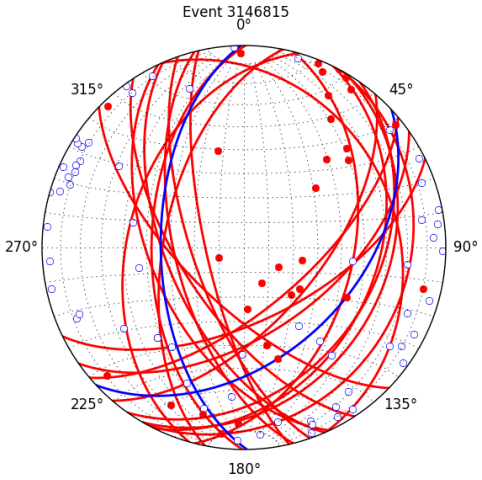

Fig. 4: *For event 3146815 from north1 dataset (blue) preferred nodal line estimated by HASH, (red) a sample from the set of acceptable estimates.*

One observation we can immediately take away from Fig 3 is that two diagrams are topologically different. The nodal lines of the shear source are great circles, which is not the case from the tensile source. That means there is no rotation or smooth deformation that can make one look like the other. This suggests that the two source are distinguishable, but also that there is some potential of identifying admixtures of the two based on their spectral content.

Existing Least Squares Methods

Currently, a common method (called FPFIT [FPFIT]) for earthquake classification is to assume that shear failure is the source mechanism, and then, through a least squares optimization, find the fault plane orientation that minimizes the rate of misclassification to the data. A modern code built upon FPFIT is the HASH algorithm [HASH]. The HASH software is available for download from the USGS[4] website. The HASH software comes with an example "NorthRidge" dataset which we will use to demonstrate our method. We compare the results of our algorithm with the results of HASH, which is the current state of the art. HashPy

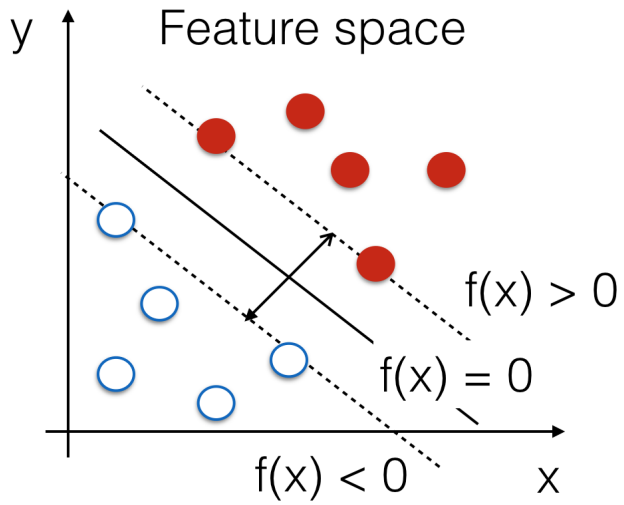

Fig. 5: *A schematic of the optimization strategy of the SVC. The dashed lines represent the edges of the separating margin. The blue open and red closed dots are the polarity data represented in a feature space. The dashed lines represent a separating margin between the two classes, the solid line represents the optimal separating hyperplane.*

[HashPy] is a Python project for that provides a wrapper for HASH.

Figure 4 demonstrates how the FPFIT algorithm works. The coordinate system in the figure is a stereonet projection [mplstereonet] of the lower half space of a sphere. The solid red (open blue) dots are positive (negative) polarity measured across a seismic network for the 3146815 event, which was taken from the Northridge dataset "north1.phase" supplied with the HASH software. Recall, FPFIT is a least squares method, however the function it is optimizing need not be convex. As such, there are many solutions that have a similar goodness of fit. Using a grid search method, FPFIT draws a ensemble of these possible solutions (red lines). The blue line is the preferred or most likely solution.

Earthquake - Learning with Kernels

In this section we discuss the classification algorithm we develop using the scikit-learn [sklearn] library. Whilst our interest was classification of earthquakes, the algorithm is applicable for any classification problem defined on a sphere.

Define the input space of the problem as the surface of the focal sphere, represented for example by the stereonet in Fig. 4. The data is not linearly separable on this space. The strategy of the SVC is to project the problem into a higher dimensional feature space. And in this feature space, determine the best hyperplane to separate the two classes of data by maximizing the width of the separating margin, subject to the constraint that the classes are either side of the separating margin, Fig. 5 shows a schematic of the algorithm. An important feature of the SVC is that it is robust to misclassification close to the decision boundary. Physically these are curves where the amplitude of the radiation is becoming small and then changing sign. What we believe to

4. http://earthquake.usgs.gov/research/software/index.php

be more important than the overall rate of misclassification of the algorithm, is the stability of the result given erroneous input data.

```
from sklearn import svm

def classify(cartesian_coords, polarity,
                              kernel_degree=2):
    """
    cartesian_coords - x, y, z coordinates on
    sphere polarity (1,-1) first break polarity
    kernel_degree - truncates the dimension of
    expansion.
    """

    # C : slack variable, use the default of 1.0
    poly_svc = svm.SVC(kernel='poly',
         degree=kernel_degree,
         coef0=1, C=1.0).fit(cartesian_coords,
                              polarity)

    intercept = poly_svc.intercept_
    # Angle [0,pi] - the colatitude
    colat = arccos(poly_svc.support_vectors_[:,2])
    # Angle [0,2*pi] measured as azimuth
    azim = arctan2(poly_svc.support_vectors_[:,1],
                   poly_svc.support_vectors_[:,0])
    # The lagrange multipliers * class,
    # classes are labeled -1 or 1.
    dual_coeff = poly_svc.dual_coef_[0,:]
    # Remember which points where mis-classified
    in_sample = poly_svc.predict(c_[inputs])

    return (dual_coeff, azim, colat,
                  intercept, in_sample)
```

A Python implementation of the support vector classifier[5] is included in scikit-learn. The projection to a higher dimensional space is done using a kernel, and evaluated in the input space using the kernel trick. For classification on a sphere, we need to use an inner product kernel, which has the form

$$k(\vec{x},\vec{x}_i) = (\langle \vec{x},\vec{x}_i\rangle + 1)^d \ .$$

Here "d" is the degree of the kernel. The parameter "C" in the above code snippet is a slack variable. This provides a soft thresholding, which allows for some misclassification; the default value is usually sufficient. Given a set of data y_i, the support vector machine learns a corresponding set of coefficients α_i and intercept β_0, which determines a classifying function in the input space,

$$f(\vec{x}) = \sum_{i=1}^{N} \alpha_i y_i k(\vec{x},\vec{x}_i) + \beta_0 \ . \tag{1}$$

In our application, the zero of this function is the nodal line, and the sign of the function is a prediction for the direction of the displacement radial to the focal sphere, given the observed data. Not all of the data is relevant for determining the best separating margin, many of the coefficients α_i may be zero. The support vectors are the locations of the data where α_i are non-zero. The product $\alpha_i y_i$ associated with each of the support vectors are called the dual coefficients (see the code snippet).

In Fig. 6 we demonstrate the SVC classifier applied to an event from the Northridge dataset. The red line represents zeros of the classifying function f(x), the green line is the solution for the fault (and auxiliary) planes determined by HASH. Note that the auxiliary plane is computed using the aux_plane function provided by the ObsPy library [ObsPy]. The learned nodal line is simply connected, the zeros of the classifying function f(x) have been determined using matplotlib's contour function.

5. http://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html

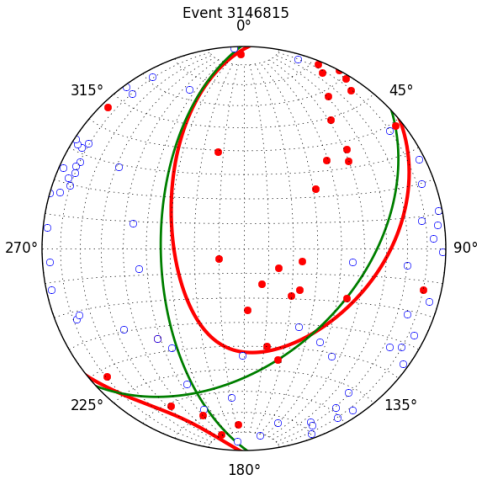

Fig. 6: *For event 3146815 from the NorthRidge dataset. The green nodal line is estimated by HASH and the red nodal line is estimated by the SVC.*

Both the HASH solution and the learned solution have a similar rate of misclassification. However the learned solution is still unsatisfactory to us because we cannot make physical sense of the result. What we want is an explanation of the type of source mechanism and its orientation. To be physically meaningful, we need an expression for the nodal lines in terms of its spectrum in the basis of spherical harmonic functions. In this basis we can then use the seismic source theory of [Ben81] to relate the result to a physical process. What we want is to determine the spectral content of f(x),

$$f(\vec{x}) = \sum_{l=1}^{\infty} \sum_{m=-l}^{l} \hat{f}_{lm} Y_{lm}(\theta,\phi)$$

that is, we want to derive its representation in the feature space. Here, the azimuth θ and colatitude ϕ, are the angles that orientate the unit vector $\vec{x}$. The steps in deriving this representation are to first expand the inner product kernel in terms of the Legendre polynomials [Scholkopf],

$$a_l = \int_{-1}^{1} dx\ (x+1)^d P_l(x) \tag{2}$$

$$a_l = \begin{cases} \frac{2^{d+1}\Gamma(d+1)}{\Gamma(d+2+l)\Gamma(d+1-l)} + \frac{1}{2}\sqrt{\frac{1}{\pi}}\beta_0\delta_{l0} & \text{if } l \le d \\ 0 & \text{otherwise} \end{cases} .$$

When we do this, we see that the degree parameter provides a natural truncation on the complexity of the function we are learning. This gives us an intermediate result which expresses the separating margin in terms of Legendre polynomials

$$f(\vec{x}) = \sum_{i=1}^{N} \alpha_i y_i \sum_{l=1}^{\infty} a_l P_l(\langle \vec{x},\vec{x}_i\rangle) \ .$$

The next step is to apply the addition theorem to express this in terms of the spherical harmonics,

$$P_l(\langle \vec{x},\vec{x}_i\rangle) = \sum_{m=-l}^{l} Y^*_{lm}(\theta',\phi')\ Y_{lm}(\theta,\phi) \ .$$

The result is a formula for the spectral content of the focal mechanism given the dual coefficients estimated by the support vector classifier,

$$\hat{f}_{lm} = \frac{4\pi}{2l+1} \sum_{i=1}^{N} \alpha_i y_i a_l \; Y^*_{lm}(\theta', \phi') \; .$$

Finally, suppose we have solutions for the classification from two different sources, either observed or modeled from Table 1. A natural metric for comparing the two sources is a correlation coefficient,

$$\rho = \frac{\|\langle g, f \rangle\|^2}{\|g\| \|f\|} \; . \tag{3}$$

Using the orthogonality condition of the spherical harmonic functions, we can show that inner product is,

$$\begin{aligned} \langle g, f \rangle &= \int d^3x \; g^*(\vec{x}) \; f(\vec{x}) \\ &= \sum_{l=0}^{\infty} \sum_{m,n} \hat{g}^*_{ln} \hat{f}_{lm} \; , \end{aligned}$$

here the integral is over the surface of the focal sphere and the star-notation means complex conjugation.

In the context of hazard monitoring, we could use the as a metric of risk, without having to propose a source mechanism or fault plane orientation.

Physical Interpretation

In the previous section we derived the general earthquake classification algorithm and a metric of correlation. Now suppose we were to assume a model for the source mechanism (e.g shear failure), how would we estimate the most likely orientation of the fault plane in this model?

First of all, in Table 1, we have a template for the spectral content of the shear source given a particular orientation. Using this template we compute a function g(x), and then generate a rotation in the input space to realign it with the classifying function f(x). This rotation would be estimated by optimizing a correlation coefficient with respect to the Euler angles,

$$\langle g, f \rangle = \arg\max_{\alpha,\beta,\gamma} \int d^3x \; g^*(R(\alpha,\beta,\gamma)\vec{x}) \; f(\vec{x})$$

Here, R represents a rotation matrix. This would be a relatively complicated procedure in the input space because we would need to re-evaluate the function g(x) at each iteration of the optimization. It is far more efficient to instead generate rotations in the feature space. To do this we borrow from quantum theory, and present Wigner's D-matrices,

$$g(R(\alpha,\beta,\gamma)\vec{x}) = \sum_{l=0}^{\infty} \sum_{m,n} D^l_{mn}(\alpha,\beta,\gamma) \hat{g}_{ln} Y_{lm}(\theta,\phi) \; .$$

Wigner's D-matrices are operators which generate rotations in the feature space of the problem. This means that we can translate a template solution (Table 1.) in a particular orientation, to a solution in any arbitrary orientation, by acting on its spectral content.

```
from scipy.optimize import minimize

def _corr_shear(x, alm):
    strike, dip, rake = x
    # Wigner is ZYZ Euler rotation, \gamma = -rake
    D = WignerD2(strike, dip, -rake).conjugate()
    # Template (13)/(31) : glm = (0, -1j, 0, -1j, 0)
    prop = (inner(D[:,3], alm) +
                  inner(D[:,1], alm))*1j
    # Maximize, not minimize.
    return -norm(prop)

def corr_shear(Alm):
    # pick a good starting point.
    x0 = _scan_shear(alm)
    f = lambda x : _corr_shear(x,alm)
    results = minimize(f, x0=x0,
              bounds=((0,2*pi), (0,pi), (0,2*pi)))
    return rad2deg(results.x), results.fun
```

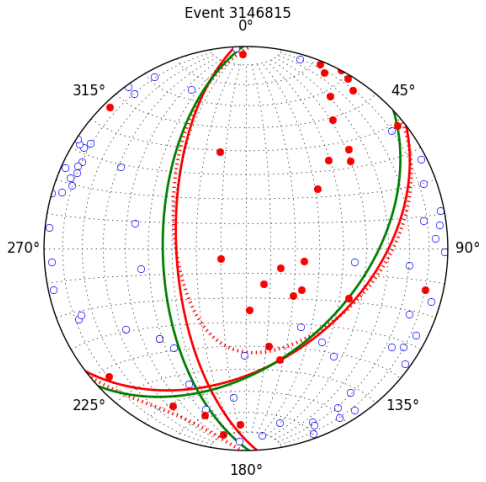

Fig. 7: *For event 314681 from NorthRidge dataset. The green nodal line estimated by HASH and the solid red line is the optimal solution for the nodal lines derived from the SVC assuming a shear source. The dashed red line is the nodal line estimated by the SVC.*

The function corr_shear shown in the code snippet implements the optimization of the above equation. The function WignerD2 implements the Wigner-D matrices defined in [Morrison], the variable "prop" is the projection of the learned solution onto the rotated template shear solution shown in Table 1, and Alm is the learned spectral content of the source. The initial guess is found scanning a coarse grid to find the best the quadrant with the highest initial correlation. This stops SciPy's default minimization [scipy] getting stuck in a local minima.

As an example, in Fig. 7 we show the classification results for the 3146815 event. The (dashed red) line shows the nodal line of the classifier function. The (solid red) line is the template shear solution, orientated by optimizing the correlation function, and the (solid green) line shows the preferred solution estimated by HASH.

Discussion

In Figures 6 and 7 we have shown examples of the classification and fault plane estimation methods. In this section we want to explore the robustness of the algorithm and try to gain some insight into the utility of the correlation functions.

The HASH program has an input (scsn.reverse) which identifies stations whose polarity was found to be erroneous in the past. These reversals are applied post facto to correct the input polarity data. We will use this feature to demonstrate an example where the support vector and least squares classifiers behave differently. In Fig 8 we give an example where we flipped the polarity of a single datum (indicated by the black arrow). The corresponding solutions are shown with (solid lines) and without (dashed lines) the benefit

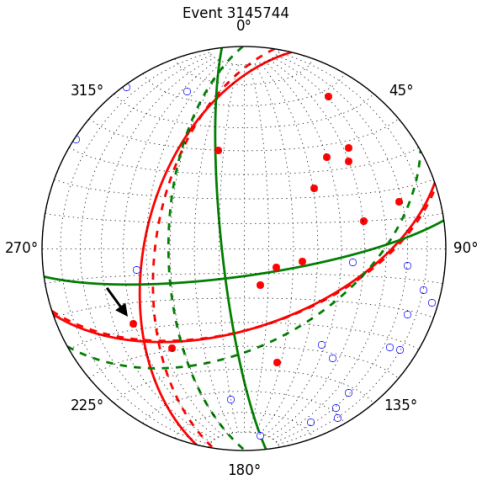

Fig. 8: *For event 3145744 from the NorthRidge dataset. The color scheme for each subplot as in Fig. 7, the dashed lines are solutions without the station reversal being applied. The black arrow points to datum for which the polarity is flipped.*

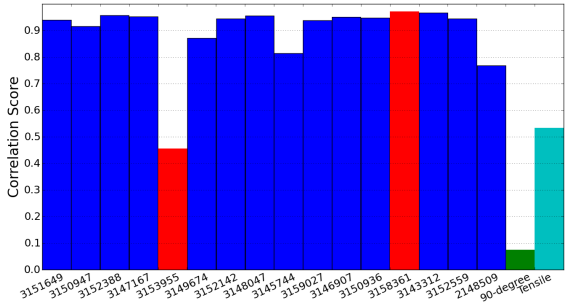

Fig. 9: *The correlation score for each event in the Northridge dataset, comparing to event 3146815. (red) The events with maximum and minimum correlation score. (green) The correlation between 3146815 and itself rotated by 90-degrees strike, (cyan) the correlation between 3146815 and the tensile source found in Table 1.*

of the polarity correction. The datum that was changed is close to the nodal line estimated by the SVC, which, given the soft thresholding, is forgiving of misclassification along its separating margin. The SVC solution for the nodal line is largely unchanged. On the other hand, the strategy of FPFIT is to minimize the overall rate of misclassification. And indeed, in each case, it finds the optimal solution on this basis. In fact, in terms of misclassified points, FPFIT outperforms the SVC classifier. But we would question whether minimizing the overall rate of misclassification is reasonable from an applied perspective. Consider that since the nodal line represents a point where the radiation pattern is changing sign, we expect that the signal to noise level will be smaller in this region. Conversely, from the point of view of the SVC, these are also the points that are most informative to the proper location of its separating margin. Indeed, many of the best quality picks far from the nodal lines will not influence the solution for the separating plane (recall dual coefficients can be zero). And it is reasonable that data of the correct class located far from the separating margin should not influence the solution. Looking at the problem from this perspective the solution of the SVC is more reasonable.

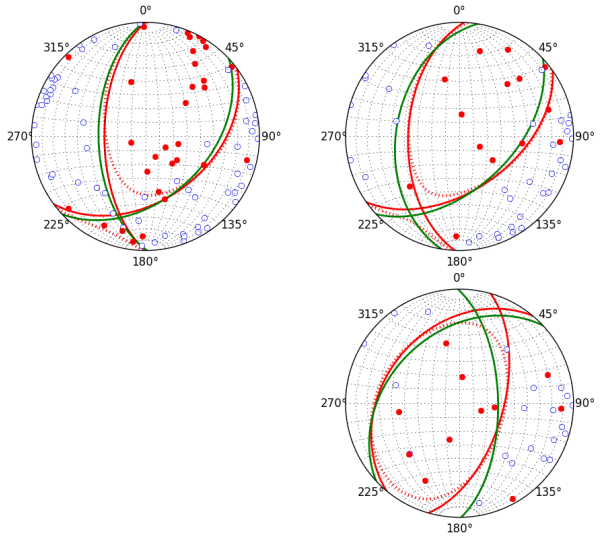

Fig. 10: *The color scheme for each subplot as in Fig. 7. (top left) The solution for event 3146815, (top right) the solution for events 3158361 and (bottom right) 3153955. Events 3158361 and 3153955 represent the maximum and minimum correlation score with event 3146815.*

Finally, we derived a metric of similarity based on a correlation score Eq. 3. To provide an example of how we might use this correlation score, we take the event 3146815, which has the largest number of data associate with it, and compute the correlation coefficient with each of the other events in the Northridge dataset. According to [HASH], the NorthRidge dataset we analyzed is expected to contain similar source mechanisms and certainly we see that the correlation score is high for the majority of the events. To test the sensitivity of the metric, we also compute the correlation between event 3146815 and itself rotated by 90-degrees strike, and we see that this has low correlation, which we would expect.

In Fig. 10 we provide a visualization of the events with the highest (top right) and lowest (bottom right) correlation score comparing with event 3146815 (top left). The orientation of the nodal lines for event 3153955, which has the lowest correlation score, indeed is qualitatively different than the solution for event 3146815. Qualitatively, we have demonstrated that the correlation score is a reasonable metric of similarity. Determining the actual statistical significance of the correlation score is left as future work.

Conclusions

We have presented a tool for classifying and comparing earthquake source mechanisms using tools from the scientific Python ecosystem. The important steps were to define the problem in terms of classification, which is solved robustly by the scikit-learn [sklearn] support vector classifier. We then used results from seismic source theory [Ben81] to derive a mapping between the input and feature spaces of the classification problem. Using the representation of the solution in the feature space, we derived a correlation coefficient.

This allowed us to generalize the earthquake classification to support both shear and tensile sources. As a particular example, we showed how maximizing correlation with the template shear solution could be used to estimate fault plane orientation. The key

to efficiency here was to generate rotations in the feature space of the problem using Wigner's D matrices.

At each step along the way, we made a comparison with similar solutions obtained with the HASH algorithm [HASH], and found good general agreement. However, we argued that for this application, the optimization strategy of the SVC should prove more robust to misclassification than the least squares method.

Finally, we showed qualitatively, that the correlation coefficient provided a good metric for comparison between sources within the Northridge dataset. This technique has some promise as a tool for earthquake monitoring.

Appendix

The template solutions shown in Table 1 were derived from solutions tabulated in Table 4.4 of [Ben81]. Here, [Ben81] gives the solutions for the first P-wave arrival in terms of the Hansen vector L (in spherical polar coordinates) of the form,

$$\vec{L}_{lm}(r,\theta,\phi) = \vec{\nabla} h_l^2(r) \tilde{Y}_{lm}(\theta,\phi) \ ,$$

where "h" is the spherical Hankel functions of a second kind. The amplitudes of the first break are required to be measured radially to the focal sphere, the projection of the Hansen vector radially is,

$$\hat{r} \cdot \vec{L}_{lm}(r,\theta,\phi) = \frac{\partial}{\partial r} h_l^2(r) \tilde{Y}_{lm}(\theta,\phi) \ .$$

The angular variation is given by the spherical harmonic function, up to an overall phase associated with radial component. Asymptotically (measurements are made far from the source), in this limit the Hankel functions tend to [Morse53],

$$h_l^2(x) = \frac{1}{x}(i)^{l+1} \exp^{-ix} \ ,$$

which introduces a relative phase when collecting terms of different degree. We also note that the normalization of the spherical harmonics used in [Ben81] does not include the Cordon Shortley phase convention. Since we are using Wigner-D matrices to generate rotations, it is convenient to use that convention,

$$\tilde{Y}_{lm}(\theta,\phi) = (-1)^m \sqrt{\frac{4\pi(l+m)!}{(2l+1)(l-m)!}} Y_{lm}(\theta,\phi) \ .$$

The reference implementation[6] includes its own sph_harm function to add this phase factor. With these adjustments, the amplitudes (up to an overall constant) for a common set of source mechanism, in terms of the spherical harmonics, are given in Table 1.

References

[Ben81] A. Ben-Menahem and S. J. Singh *Seismic Waves and Sources* Springer-Verlag New York Inc., 1981

[Aki02] K. Aki and P. G. Richards *Quantitative Seismology, second ed.* University Science Books, 2002

[Morse53] M. Morse and F. Feshbach, Methods of theoretical physics Feschbach Publishing LLC, 1953

[HASH] J. L. Hardeback and P. M. Shearer, A New Method for Determining First-Motion Focal Mechanisms, Bulletin of the Seismological Socity of America, Vol. 92, pp 2264-2276, 2002

[FPFIT] Reasenberg, P., and D. Oppenheimer (1985). FPFIT, FPPLOT, and FPPAGE: FORTRAN computer programs for calculating and displaying earthquake faultplane solutions, U.S. Geol. Surv. Open-File Rept. 85-739, 109 Pp.

[Morrison] M. A. Morrison and G. A. Parker, Australian Journal of Physics 40, 465 (1987).

[Scholkopf] B. Scholkopf and A. Smola, Learning with Kernels, The MIT Press, 2002

[sklearn] 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)

[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 (publisher link)

[mpl] John D. Hunter. Matplotlib: A 2D Graphics Environment, Computing in Science & Engineering, 9, 90-95 (2007), DOI:10.1109/MCSE.2007.55

[ObsPy] M. Beyreuther, R. Barsch, L. Krischer, T. Megies, Y. Behr and J. Wassermann (2010), ObsPy: A Python Toolbox for Seismology, SRL, 81(3), 530-533, DOI: 10.1785/gssrl.81.3.530

[HashPy] hashpy, https://github.com/markcwill/hashpy, DOI:10.5281/zenodo.9808

[mplstereonet] mplstereonet, https://pypi.python.org/pypi/mplstereonet

6. https://github.com/blasscoc/FocalMechClassifier

cesium: Open-Source Platform for Time-Series Inference

Brett Naul[‡*], Stéfan van der Walt[‡], Arien Crellin-Quick[‡], Joshua S. Bloom[§‡], Fernando Pérez[§‡]

https://youtu.be/ZgHGCfwExw0

Abstract—Inference on time series data is a common requirement in many scientific disciplines and internet of things (IoT) applications, yet there are few resources available to domain scientists to easily, robustly, and repeatably build such complex inference workflows: traditional statistical models of time series are often too rigid to explain complex time domain behavior, while popular machine learning packages require already-featurized dataset inputs. Moreover, the software engineering tasks required to instantiate the computational platform are daunting. `cesium` is an end-to-end time series analysis framework, consisting of a Python library as well as a web front-end interface, that allows researchers to featurize raw data and apply modern machine learning techniques in a simple, reproducible, and extensible way. Users can apply out-of-the-box feature engineering workflows as well as save and replay their own analyses. Any steps taken in the front end can also be exported to a Jupyter notebook, so users can iterate between possible models within the front end and then fine-tune their analysis using the additional capabilities of the back-end library. The open-source packages make us of many use modern Python toolkits, including `xarray`, `dask`, Celery, Flask, and `scikit-learn`.

Index Terms—time series, machine learning, reproducible science

Introduction

From the reading of electroencephalograms (EEGs) to earthquake seismograms to light curves of astronomical variable stars, gleaning insight from time series data has been central to a broad range of scientific disciplines. When simple analytical thresholds or models suffice, technicians and experts can be easily removed from the process of inspection and discovery by employing custom algorithms. But when dynamical systems are not easily modeled (e.g., through physics-based models or standard regression techniques), classification and anomaly detection have traditionally been reserved for the domain expert: digitally recorded data are visually scanned to ascertain the nature of the time variability and find important (perhaps life-threatening) outliers. *Does this person have an irregular heartbeat? What type of supernova occurred in that galaxy?* Even in the presence of sensor noise and intrinsic diversity of the samples, well-trained domain specialists show a remarkable ability to make discerning statements about complex data.

* *Corresponding author: bnaul@berkeley.edu*
‡ *University of California, Berkeley*
§ *Lawrence Berkeley National Laboratory*

In an era when more time series data are being collected than can be visually inspected by domain experts, computational frameworks must necessarily act as human surrogates. Capturing the subtleties that domain experts intuit in time series data (let alone besting the experts) is a non-trivial task. In this respect, machine learning has already been used to great success in several disciplines, including text classification, image retrieval, segmentation of remote sensing data, internet traffic classification, video analysis, and classification of medical data. Even if the results are similar, some obvious advantages over human involvement are that machine learning algorithms are tunable, repeatable, and deterministic. A computational framework built with elasticity can scale, whereas experts (and even crowdsourcing) cannot.

Despite the importance of time series in scientific research, there are few resources available that allow domain scientists to easily build robust computational inference workflows for their own time series data, let alone data gathered more broadly in their field. The difficulties involved in constructing such a framework can often greatly outweigh those of analyzing the data itself:

> It may be surprising to the academic community to know that only a tiny fraction of the code in many machine learning systems is actually doing "machine learning"...a mature system might end up being (at most) 5% machine learning code and (at least) 95% glue code. [SHG+14]

Even if a domain scientist works closely with machine learning experts, the software engineering requirements can be daunting. It is our opinion that being a modern data-driven scientist should not require an army of software engineers, machine learning experts, statisticians and production operators. `cesium` [cT16] was created to allow domain experts to focus on the inference questions at hand rather than needing to assemble a complete engineering project.

The analysis workflow of `cesium` can be used in two forms: a web front end which allows researchers to upload their data, perform analyses, and visualize their models all within the browser; and a Python library which exposes more flexible interfaces to the same analysis tools. The web front end is designed to handle many of the more cumbersome aspects of machine learning analysis, including data uploading and management, scaling of computational resources, and tracking of results from previous experiments. The Python library is used within the web back end for the main steps of the analysis workflow: extracting features from raw time series, building models from these features, and

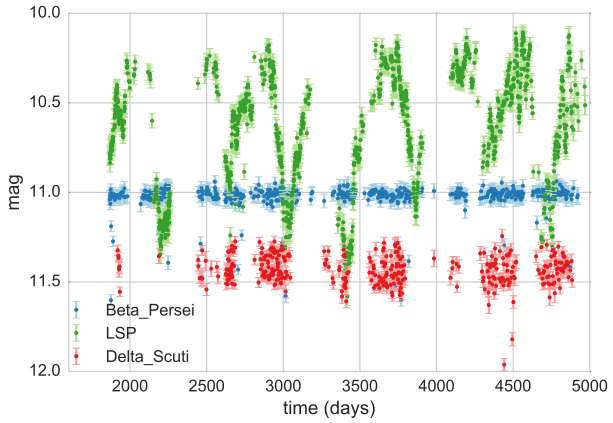

Fig. 1: *Typical data for a classification task on variable stars from the All Sky Automated Survey; shown are flux measurements for three stars irregularly sampled in time [RSM+12].*

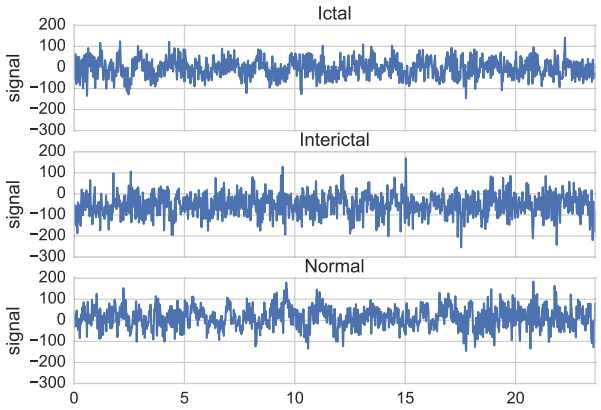

Fig. 2: *EEG signals from patients with epilepsy [ALM+01].*

generating predictions. The library also supplies data structures for storing time series (including support for irregularly-sampled time series and measurement errors), features, and other relevant metadata.

In the next section, we describe a few motivating examples of scientific time series analysis problems. The subsequent sections describe in detail the `cesium` library and web front end, including the different pieces of functionality provided and various design questions and decisions that arose during the development process. Finally, we present an end-to-end analysis of an EEG seizure dataset, first using the Python library and then via the web front end.

Example time series machine learning problems

`cesium` was designed with several time series inference use cases across various scientific disciplines in mind.

1) **Astronomical time series classification.** Beginning in 2020, the Large Synoptic Survey Telescope (LSST) will survey the entire night's sky every few days producing high-quality time series data on approximately 800 million transient events and sources with variable brightness (Figure 1 depicts the brightness of several types of star over the course of several years) [AAA+09]. Much of the best science in the time domain (e.g., the discovery of the accelerating universe and dark energy using Type Ia supernovae [PAG+99], [RFC+98]) consists of first identifying possible phenomena of interest using broad data mining approaches and following up by collecting more detailed data using other, more precise observational tools. For many transient events, the time scale during which observations can be collected can be on the order of days or hours. Not knowing which of the millions of variable sources to examine more closely with larger telescopes and specialized instruments is tantamount to not having discovered those sources at all. Discoveries must be identified quickly or in real time so that informed decisions can be made about how best to allocate additional observational resources.

2) **Neuroscience time series classification.** that might need to be classified in order to make treatment decisions. Neuroscience experiments now produce vast amounts of time series data that can have entirely different structures and spatial/temporal resolutions, depending on the recording technique. Figure 2 shows an example of different types of EEG signals The neuroscience community is turning to the use of large-scale machine learning tools to extract insight from large, complex datasets [LCL+07]. However, the community lacks tools to validate and compare data analysis approaches in a robust, efficient and reproducible manner: even recent expert reviews on the matter leave many of these critical methodological questions open for the user to explore in an ad hoc way and with little principled guidance [PG07].

3) **Earthquake detection, characterization and warning.** Earthquake early warning (EEW) systems are currently in operation in Japan, Mexico, Turkey, Taiwan and Romania [AGKB09] and are under development in the US [BAH+11]. These systems have employed sophisticated remote sensors, real-time connectivity to major broadcast outlets (such as TV and radio), and have a growing resumé of successful rapid assessment of threat levels to populations and industry. Traditionally these warning systems trigger from data obtained by high-quality seismic networks with sensors placed every ~10 km. Today, however, accelerometers are embedded in many consumer electronics including computers and smartphones. There is tremendous potential to improve earthquake detection methods using streaming classification analysis both using traditional network data and also harnessing massive data from consumer electronics.

Simple and reproducible workflows

In recent years, there has been rapid growth in the availability of open-source tools that implement a wide variety of machine learning algorithms: packages within the R [T+13] and Python programming languages [PVG+11], standalone Java-based packages such as Moa [BHKP10] and Weka [HFH+09], and online webservices such as the Google Prediction API, to name a few. To a domain scientist that does not have formal training in

machine learning, however, the availability of such packages is both a blessing and a curse. On one hand, most machine learning algorithms are now widely accessible to all researchers. At the same time, these algorithms tend to be black boxes with potentially many enigmatic knobs to turn. A domain scientist may rightfully ask just which of the many algorithms to use, which parameters to tune, and what the results actually mean.

The goal of `cesium` is to simplify the analysis pipeline so that scientists can spend less time solving technical computing problems and more time answering scientific questions. `cesium` provides a library of feature extraction techniques inspired by analyses from many scientific disciplines, as well as a surrounding framework for building and analyzing models from the resulting feature information using `scikit-learn` (or potentially other machine learning tools).

By recording the inputs, parameters, and outputs of previous experiments, *cesium'* allows researchers to answer new questions that arise out of previous lines of inquiry. Saved `cesium` workflows can be applied to new data as it arrives and shared with collaborators or published so that others may apply the same beginning-to-end analysis for their own data.

For advanced users or users who wish to delve into the source code corresponding to a workflow produced through the `cesium` web front end, we are implementing the ability to produce a Jupyter notebook [PG07] from a saved workflow with a single click. While our goal is to have the front end to be as robust and flexible as possible, ultimately there will always be special cases where an analysis requires tools which have not been anticipated, or where the debugging process requires a more detailed look at the intermediate stages of the analysis. Exporting a workflow to a runnable notebook provides a more detailed, lower-level look at how the analysis is being performed, and can also allow the user to reuse certain steps from a given analysis within any other Python program.

`cesium` library

The first half of the `cesium` framework is the back-end Python-based library, aimed at addressing the following uses cases:

1) A domain scientist who is comfortable with programming but is **unfamiliar with time series analysis or machine learning**.
2) A scientist who is experienced with time series analysis but is looking for **new features** that can better capture patterns within their data.
3) A user of the `cesium` web front end who realizes they require additional functionality and wishes to add additional stages to their workflow.

Our framework primarily implements "feature-based methods", wherein the raw input time series data is used to compute "features" that compactly capture the complexity of the signal space within a lower-dimensional feature space. Standard machine learning approaches (such as random forests [Bre01] and support vector machines [SV99]) may then be used for supervised classification or regression.

`cesium` allows users to select from a large library of features, including both general time series features and domain-specific features drawn from various scientific disciplines. Some specific advantages of the `cesium` featurization process include:

- Support for both regularly and irregularly sampled time series.

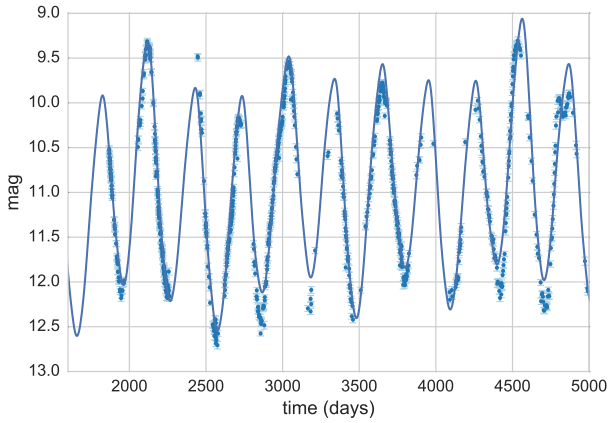

Fig. 3: *Fitted multi-harmonic Lomb-Scargle model for a light curve from a periodic Mira-class star.* `cesium` *automatically generates numerous features based on Lomb-Scargle periodogram analysis.*

- Ability to incorporate measurement errors, which can be provided for each data point of each time series (if applicable).
- Support for multi-channel data, in which case features are computed separately for each dimension of the input data.

Example features

Some `cesium` features are extremely simple and intuitive: summary statistics such as maximum/minimum values, mean/median values, and standard deviation or median absolute deviation are a few such examples. Other features involve measurement errors if they are available: for example, a mean and standard deviation that is weighted by measurement errors allows noisy data with large outliers to be modeled more precisely.

Other more involved features could be the estimated parameters for various fitted statistical models: Figure 3 shows a multi-frequency, multi-harmonic Lomb-Scargle model that describes the rich periodic behavior in an example time series [Lom76], [Sca82]. The Lomb-Scargle method is one approach for generalizing the process of Fourier analysis of frequency spectra to the case of irregularly sampled time series. In particular, a time series is modeled as a superposition of periodic functions

$$\tilde{y}(t) = \sum_{k=1}^{m} \sum_{l=1}^{n} A_{kl} \cos k\omega_l t + B_{kl} \sin k\omega_l t,$$

where the parameters A_{kl}, B_{kl}, and ω_l are selected via non-convex optimization to minimize the residual sum of squares (weighted by measurement errors if applicable). The estimated periods, amplitudes, phases, goodness-of-fits, and power spectrum can then be used as features which broadly characterize the periodicity of the input time series.

Usage overview

Here we provide a few examples of the main `cesium` API components that would be used in a typical analysis task. A workflow will typically consist of three steps: featurization, model building, and prediction on new data. The majority of `cesium` functionality is contained within the `cesium.featurize` module; the `cesium.build_model` and `cesium.predict` modules primarily provide interfaces between sets of feature data, which

contain both feature data and a variety of metadata about the input time series, and machine learning models from `scikit-learn` [PVG+11], which require dense, rectangular input data. Note that, as `cesium` is under active development, some of the following details are subject to change.

The featurization step is performed using one of two main functions:

- `featurize_time_series(times, values, errors, ...)`
  - Takes in data that is already present in memory and computes the requested features (passed in as string feature names) for each time series.
  - Features can be computed in parallel across workers via Celery, a Python distributed task queue [Sol14], or locally in serial.
  - Class labels/regression targets and metadata/features with known values are passed in and stored in the output dataset.
  - Additional feature functions can be passed in as `custom_functions`.
- `featurize_data_files(uris, ...)`,
  - Takes in a list of file paths or URIs and dispatches featurization tasks to be computed in parallel via Celery.
  - Data is loaded only remotely by the workers rather than being copied, so this approach should be preferred for very large input datasets.
  - Features, metadata, and custom feature functions are passed in the same way as `featurize_data_files`.

The output of both functions is a `Dataset` object from the `xarray` library [Hoy15], which will also be referred to here as a "feature set" (more about `xarray` is given in the next section). The feature set stores the computed feature values for each function (indexed by channel, if the input data is multi-channel), as well as time series filenames or labels, class labels or regression targets, and other arbitrary metadata to be used in building a statistical model.

The `build_model` contains tools meant to to simplify the process of building `sckit-learn` models from (non-rectangular) feature set data:

- `model_from_featureset(featureset, ...)`
  - Returns a fitted `scikit-learn` model based on the input feature data.
  - A pre-initialized (but untrained) model can be passed in, or the model type can be passed in as a string.
  - Model parameters can be passed in as fixed values, or as ranges of values from which to select via cross-validation.

Analogous helper functions for prediction are available in the `predict` module:

- `model_predictions(featureset, model, ...)`
  - Generates predictions from a feature set outputted by `featurize_time_series` or `featurize_data_files`.
- `predict_data_files(file_paths, model, ...)`
  - Like `featurize_data_files`, generate predictions for time series which have not yet been featurized by dispatching featurization tasks to Celery workers and then passing the resulting featureset to `model_predictions`.

After a model is initially trained or predictions have been made, new models can be trained with more features or uninformative features can be removed until the result is satisfactory.

Implementation details

`cesium` is implemented in Python, along with some C code (integrated via Cython) for especially computationally-intensive feature calculations. Our library also relies upon many other open source Python projects, including `scikit-learn`, `pandas`, `xarray`, and `dask`. As the first two choices are somewhat obvious, here we briefly describe the roles of the latter two libraries.

As mentioned above, feature data generated by `cesium` is returned as a `Dataset` object from the `xarray` package, which according to the documentation "resembles an in-memory representation of a NetCDF file, and consists of variables, coordinates and attributes which together form a self describing dataset". A `Dataset` allows multi-channel feature data to be faithfully represented in memory as a multidimensional array so that the effects of each feature (across all channels) or channel (across all features) can be evaluated directly, while also storing metadata and features that are not channel-specific. Storing feature outputs in NetCDF format allows for faster and more space-efficient serialization and loading of results (as compared to a text-based format).

The `dask` library provides a wide range of tools for organizing computational full process of exporting tasks. `cesium` makes use of only one small component: within `dask`, tasks are organized as a directed acyclic graph (DAG), with the results of some tasks serving as the inputs to others. Tasks can then be computed in an efficient order by `dask`'s scheduler. Within `cesium`, many features rely on other features as inputs, so internally we represent our computations as `dask` graphs in order to minimize redundant computations and peak memory usage. Part of an example DAG involving the Lomb-Scargle periodogram is depicted in Figure 4: circles represent functions, and rectangles the inputs/outputs of the various steps. In addition to the built-in features, custom feature functions passed in directly by the user can similarly make use of the internal `dask` representation so that built-in features can be reused for the evaluation of user-specified functions.

Web front end

The `cesium` front end provides web-based access to time series analysis, addressing three common use cases:

1) A scientist needs to perform time series analysis, but is **unfamiliar with programming** and library usage.
2) A group of scientists want to **collaboratively explore** different methods for time-series analysis.
3) A scientist is unfamiliar with time-series analysis, and wants to **learn** how to apply various methods to their data, using **industry best practices**.

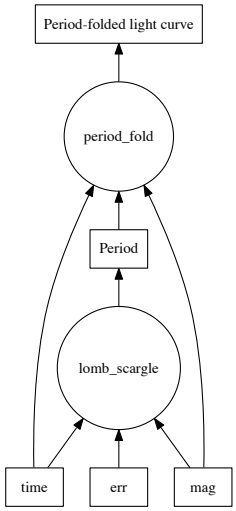

Fig. 4: *Example of a directed feature computation graph using* `dask`.

The front-end system (together with its deployed back end), offers the following features:

- Distributed, parallelized fitting of machine learning models.
- Isolated[1], cloud-based execution of user-uploaded featurization code.
- Visualization and analysis of results.
- Tracking of an entire exploratory workflow from start-to-finish for reproducibility (in progress).
- Downloads of Jupyter notebooks to replicate analyses[2].

Implementation

The `cesium` web front end consists of several components:

- A Python-based Flask [Ron15] server which provides a REST API for managing datasets and launching featurization, model-building, and prediction tasks.
- A JavaScript-based web interface implemented using React [Gac15b] and Redux [Gac15a] to display results to users.
- A custom WebSocket communication system (which we informally call *message flow*) that notifies the front end when back-end tasks complete.

While the deployment details of the web front end are beyond the scope of this paper, it should be noted that it was designed with scalability in mind. The overarching design principle is to connect several small components, each performing only one, simple task. An NGINX proxy exposes a pool of WebSocket and Web Server Gateway Interface (WSGI) servers to the user. This gives us the flexibility to choose the best implementation of each. Communications between WSGI servers and WebSocket servers happen through a ZeroMq XPub-XSub (multi-publisher publisher-subscriber) pipeline [Hin13], but could be replaced with any other broker, e.g., RabbitMQ [VW12]. The "message flow" paradigm adds WebSocket support to any Python WSGI server (Flask, Django[3], Pylons, etc.), and allows scaling up as demand increases. It also implement trivially modern data flow models such as Flux/Redux, where information always flows in one direction: from front end to back end via HTTP (Hypertext Transfer Protocol) calls, and from back end to front end via WebSocket communication.

Computational Scalability

In many fields, the volumes of available time series data can be immense. `cesium` includes features to help parallelize and scale an analysis from a single system to a large cluster.

Both the back-end library and web front end make use of Celery [Sol14] for distributing featurization tasks to multiple workers; this could be used for anything from automatically utilizing all the available cores of a single machine, to assigning jobs across a large cluster. Similarly, both parts of the `cesium` framework include support for various distributed filesystems, so that analyses can be performed without copying the entire dataset into a centralized location.

While the `cesium` library is written in pure Python, the overhead of the featurization tasks is minimal; the majority of the work is done by the feature code itself. Most of the built-in features are based on high-performance `numpy` functions; others are written in pure C with interfaces in Cython. The use of `dask` graphs to eliminate redundant computations also serves to minimize memory footprint and reduce computation times.

Automated testing and documentation

Because the back-end library and web front end are developed in separate GitHub repositories, the connections between the two somewhat complicate the continuous integration testing setup. Both repositories are integrated with Travis CI for automatic testing of all branches and pull requests; in addition, any new pushes to `cesium/master` trigger a set of tests of the front end using the new version of the back-end library, with any failures being reported but not causing the `cesium` build to fail (the reasoning being that the back-end library API should be the "ground truth", so any updates represent a required change to the front end, not a bug *per se*).

Documentation for the back-end API is automatically generated in ReStructured Text format via `numpydoc`; the result is combined with the rest of our documentation and rendered as HTML using `sphinx`. Code examples (without output) are stored in the repository in Markdown format as opposed to Jupyter notebooks since this format is better suited to version control. During the doc-build process, the Markdown is converted to Jupyter notebook format using `notedown`, then executed using `nbconvert` and converted back to Markdown (with outputs included), to be finally rendered by `sphinx`. This allows the code examples to be saved in a human-readable and version control-friendly format while still allowing the user to execute the code themselves via a downloadable notebook.

1. Isolation is currently provided by limiting the user to non-privileged access inside a Docker [Mer14] container.

2. Our current implementation of the front end includes the ability to track all of a user's actions in order to produce a notebook version, but the full process of generating the notebook is still a work in progress.

3. At PyCon2016, Andrew Godwin presented a similar solution for Django called "channels". The work described here happened before we became aware of Andrew's, and generalizes beyond Django to, e.g., Flask, the web framework we use.

Example EEG dataset analysis

In this example we compare various techniques for epilepsy detection using a classic EEG time series dataset from Andrzejak et al. [ALM+01]. The raw data are separated into five classes: Z, O, N, F, and S; we consider a three-class classification problem of distinguishing normal (Z, O), interictal (N, F), and ictal (S) signals. We show how to perform the same analysis using both the back-end Python library and the web front end.

Python library

First, we load the data and inspect a representative time series from each class: Figure 2 shows one time series from each of the three classes, after the time series are loaded from `cesium.datasets.andrzejak`.

Once the data is loaded, we can generate features for each time series using the `cesium.featurize` module. The `featurize` module includes many built-in choices of features which can be applied for any type of time series data; here we've chosen a few generic features that do not have any special biological significance.

If Celery is running, the time series will automatically be split among the available workers and featurized in parallel; setting `use_celery=False` will cause the time series to be featurized serially.

```
from cesium import featurize

features_to_use = ['amplitude', 'maximum',
                   'max_slope', 'median',
                   'median_absolute_deviation',
                   'percent_beyond_1_std',
                   'percent_close_to_median',
                   'minimum', 'skew', 'std',
                   'weighted_average']
fset_cesium = featurize.featurize_time_series(
                  times=eeg["times"],
                  values=eeg["measurements"],
                  errors=None,
                  features_to_use=features_to_use,
                  targets=eeg["classes"])
```

```
<xarray.Dataset>
Dimensions:    (channel: 1, name: 500)
Coordinates:
* channel    (channel) int64 0
* name       (name) int64 0 1 ...
  target     (name) object 'Normal' 'Normal' ...
Data variables:
  minimum    (name, channel) float64 -146.0 -254.0 ...
  amplitude  (name, channel) float64 143.5 211.5 ...
  ...
```

The resulting `Dataset` contains all the feature information needed to train a machine learning model: feature values are stored as data variables, and the time series index/class label are stored as coordinates (a `channel` coordinate will also be used later for multi-channel data).

Custom feature functions not built into `cesium` may be passed in using the `custom_functions` keyword, either as a dictionary `{feature_name: function}`, or as a `dask` graph. Functions should take three arrays `times, measurements, errors` as inputs; details can be found in the `cesium.featurize` documentation. Here we compute five standard features for EEG analysis suggested by Guo et al. [GRD+11]:

```
import numpy as np, scipy.stats

def mean_signal(t, m, e):
    return np.mean(m)

def std_signal(t, m, e):
    return np.std(m)

def mean_square_signal(t, m, e):
    return np.mean(m ** 2)

def abs_diffs_signal(t, m, e):
    return np.sum(np.abs(np.diff(m)))

def skew_signal(t, m, e):
    return scipy.stats.skew(m)
```

Now we pass the desired feature functions as a dictionary via the `custom_functions` keyword argument (functions can also be passed in as a list or a `dask` graph).

```
guo_features = {
    'mean': mean_signal,
    'std': std_signal,
    'mean2': mean_square_signal,
    'abs_diffs': abs_diffs_signal,
    'skew': skew_signal
}
fset_guo = featurize.featurize_time_series(
               times=eeg["times"],
               values=eeg["measurements"],
               errors=None, targets=eeg["classes"],
               features_to_use=guo_features.keys(),
               custom_functions=guo_features)
```

```
<xarray.Dataset>
Dimensions:    (channel: 1, name: 500)
Coordinates:
* channel    (channel) int64 0
* name       (name) int64 0 1 ...
  target     (name) object 'Normal' 'Normal' ...
Data variables:
  abs_diffs  (name, channel) float64 4695.2 6112.6 ...
  mean       (name, channel) float64 -4.132 -52.44 ...
  ...
```

The EEG time series considered here consist of univariate signal measurements along a uniform time grid. But `featurize_time_series` also accepts multi-channel data. To demonstrate this, we will decompose each signal into five frequency bands using a discrete wavelet transform as suggested by Subasi [Sub07], and then featurize each band separately using the five functions from above.

```
import pywt

eeg["dwts"] = [pywt.wavedec(m, pywt.Wavelet('db1'),
                            level=4)
               for m in eeg["measurements"]]
fset_dwt = featurize.featurize_time_series(
               times=None, values=eeg["dwts"], errors=None,
               features_to_use=guo_features.keys(),
               targets=eeg["classes"],
               custom_functions=guo_features)
```

```
<xarray.Dataset>
Dimensions:    (channel: 5, name: 500)
Coordinates:
* channel    (channel) int64 0 1 ...
* name       (name) int64 0 1 ...
  target     (name) object 'Normal' 'Normal' ...
Data variables:
  abs_diffs  (name, channel) float64 25131 18069 ...
```

```
  skew       (name, channel) float64 -0.0433 0.06578 ...
  ...
```

The output feature set has the same form as before, except now the `channel` coordinate is used to index the features by the corresponding frequency band. The functions in `cesium.build_model` and `cesium.predict` all accept feature sets from single- or multi-channel data, so no additional steps are required to train models or make predictions for multi-channel feature sets using the `cesium` library.

Model building in `cesium` is handled by the `model_from_featureset` function in the `cesium.build_model` module. The feature set output by `featurize_time_series` contains both the feature and target information needed to train a model; `model_from_featureset` is simply a wrapper that calls the `fit` method of a given `scikit-learn` model with the appropriate inputs. In the case of multichannel features, it also handles reshaping the feature set into a (rectangular) form that is compatible with `scikit-learn`.

For this example, we test a random forest classifier for the built-in `cesium` features, and a 3-nearest neighbors classifier for the others, as in [GRD+11].

```
from cesium.build_model import model_from_featureset
from sklearn.ensemble import RandomForestClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.cross_validation import train_test_split

train, test = train_test_split(500)

rfc_param_grid = {'n_estimators': [8, 32, 128, 512]}
model_cesium = model_from_featureset(
                    fset_cesium.isel(name=train),
                    RandomForestClassifier(),
                    params_to_optimize=rfc_param_grid)

knn_param_grid = {'n_neighbors': [1, 2, 3, 4]}
model_guo = model_from_featureset(
                fset_guo.isel(name=train),
                KNeighborsClassifier(),
                params_to_optimize=knn_param_grid)
model_dwt = model_from_featureset(
                fset_dwt.isel(name=train),
                KNeighborsClassifier(),
                params_to_optimize=knn_param_grid)
```

Making predictions for new time series based on these models follows the same pattern: first the time series are featurized using `featurize_timeseries` and then predictions are made based on these features using `predict.model_predictions`,

```
from cesium.predict import model_predictions
preds_cesium = model_predictions(
                    fset_cesium, model_cesium,
                    return_probs=False)
preds_guo = model_predictions(fset_guo, model_guo,
                    return_probs=False)
preds_dwt = model_predictions(fset_dwt, model_dwt,
                    return_probs=False)
```

And finally, checking the accuracy of our various models, we find:

```
Builtin: train acc=100.00%, test acc=83.20%
Guo et al.: train acc=90.93%, test acc=84.80%
Wavelets: train acc=100.00%, test acc=95.20%
```

The workflow presented here is intentionally simplistic and omits many important steps such as feature selection, model parameter selection, etc., which may all be incorporated just as they would for any other `scikit-learn` analysis. But with essentially three function calls (`featurize_time_series`, `model_from_featureset`, and `model_predictions`), we are able to build a model from a set of time series and make predictions on new, unlabeled data. In the next section we introduce the web front end for `cesium` and describe how the same analysis can be performed in a browser with no setup or coding required.

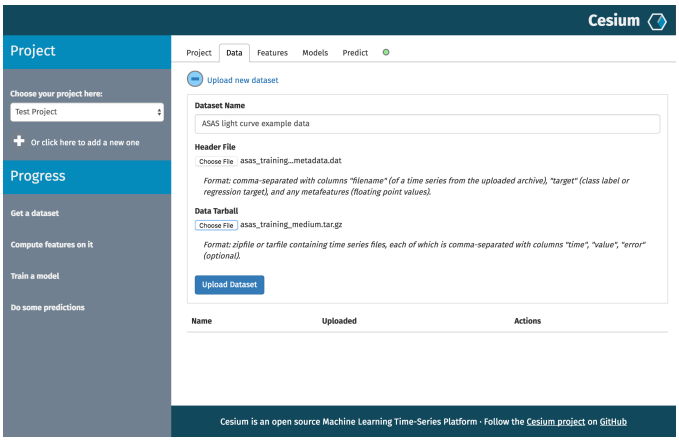

Fig. 5: *"Data" tab*

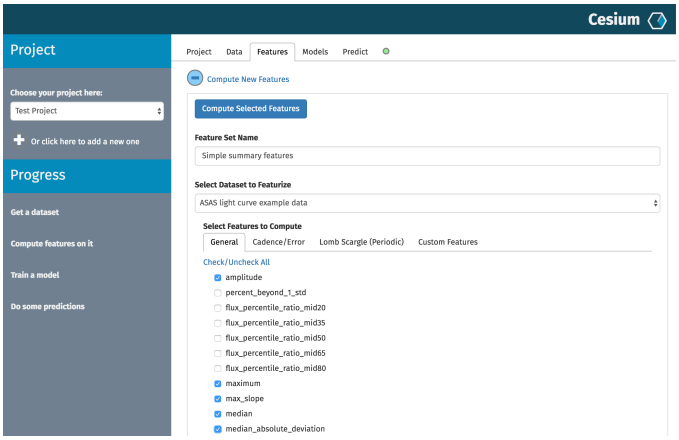

Fig. 6: *"Featurize" tab*

Web front end

Here we briefly demonstrate how the above analysis could be conducted using only the web front end. Note that the user interface presented here is a preliminary version and is undergoing frequent updates and additions. The basic workflow follows the same *featurize—build model—predict* pattern. First, data is uploaded as in Figure 5. Features are selected from available built-in functions as in Figure 6, or may be computed from user-uploaded Python code which is securely executed within a Docker container. Once features have been extracted, models can be created as in Figure 7, and finally predictions can be made as in Figure 8. Currently the options for exploring feature importance and model accuracy are limited, but this is again an area of active development.

Future work

The `cesium` project is under active development. Some of our upcoming goals include:

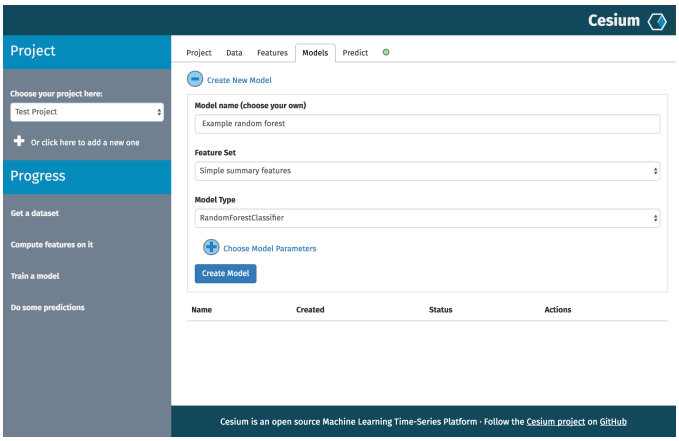

Fig. 7: *"Build Model" tab*

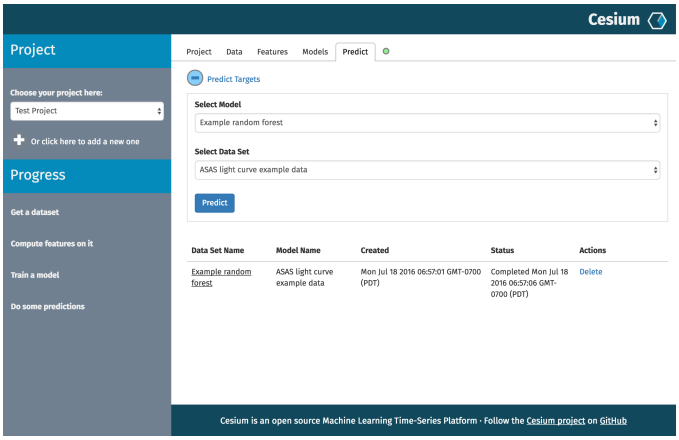

Fig. 8: *"Predict" tab*

- Full support for exporting Jupyter notebooks from any workflow created within the web front end.
- Additional features from other scientific disciplines (currently the majority of available features are taken from applications in astronomy).
- Improved web front end user interface with more tools for visualizing and exploring a user's raw data, feature values, and model outputs.
- More tools to streamline the process of iteratively exploring new models based on results of previous experiments.
- Better support for sharing data and results among teams.
- Extension to unsupervised problems.

Conclusion

The `cesium` framework provides tools that allow anyone from machine learning specialists to domain experts without any machine learning experience to rapidly prototype explanatory models for their time series data and generate predictions for new, unlabeled data. Aside from the applications to time domain informatics, our project has several aspects which are relevant to the broader scientific Python community.

First, the dual nature of the project (Python back end vs. web front end) presents both unique challenges and interesting opportunities in striking a balance between accessibility and flexibility of the two components. Second, the `cesium` project places a strong emphasis on reproducible workflows: all actions performed within the web front end are logged and can be easily exported to a Jupyter notebook that exactly reproduces the steps of the analysis. Finally, the scope of our project is simultaneously both narrow (time series analysis) and broad (numerous distinct scientific disciplines), so determining how much domain-specific functionality to include is an ongoing challenge.

References

[AAA+09] Paul A Abell, Julius Allison, Scott F Anderson, John R Andrew, J Roger P Angel, Lee Armus, David Arnett, SJ Asztalos, Tim S Axelrod, Stephen Bailey, et al. Lsst science book, version 2.0. *arXiv preprint arXiv:0912.0201*, 2009.

[AGKB09] Richard M Allen, Paolo Gasparini, Osamu Kamigaichi, and Maren Böse. The status of earthquake early warning around the world: An introductory overview. *Seismological Research Letters*, 80(5):682–693, 2009.

[ALM+01] Ralph G. Andrzejak, Klaus Lehnertz, Florian Mormann, Christoph Rieke, Peter David, and Christian E. Elger. Indications of nonlinear deterministic and finite-dimensional structures in time series of brain electrical activity: Dependence on recording region and brain state. *Physical Review E*, 64(6):061907, 2001. doi:10.1103/PhysRevE.64.061907.

[BAH+11] Holly M Brown, Richard M Allen, Margaret Hellweg, Oleg Khainovski, Douglas Neuhauser, and Adeline Souf. Development of the elarms methodology for earthquake early warning: Real-time application in california and offline testing in japan. *Soil Dynamics and Earthquake Engineering*, 31(2):188–200, 2011.

[BHKP10] Albert Bifet, Geoff Holmes, Richard Kirkby, and Bernhard Pfahringer. Moa: Massive online analysis. *Journal of Machine Learning Research*, 11(May):1601–1604, 2010.

[Bre01] Leo Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[cT16] cesium Team. *cesium: Open-Source Machine Learning for Time Series Analysis*. 2016. URL: http://cesium.ml.

[Gac15a] Cory Gackenheimer. Introducing flux: An application architecture for react. In *Introduction to React*, pages 87–106. Springer, 2015.

[Gac15b] Cory Gackenheimer. What is react? In *Introduction to React*, pages 1–20. Springer, 2015.

[GRD+11] Ling Guo, Daniel Rivero, Julián Dorado, Cristian R. Munteanu, and Alejandro Pazos. Automatic feature extraction using genetic programming: An application to epileptic EEG classification. *Expert Systems with Applications*, 38(8):10425–10436, 2011. doi:10.1016/j.eswa.2011.02.118.

[HFH+09] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H Witten. The weka data mining software: an update. *ACM SIGKDD explorations newsletter*, 11(1):10–18, 2009.

[Hin13] Pieter Hintjens. *ZeroMQ: Messaging for Many Applications*. " O'Reilly Media, Inc.", 2013.

[Hoy15] S Hoyer. xray: ND labeled arrays and datasets in Python. 2015. URL: http://github.com/xray/xray.

[LCL+07] Fabien Lotte, Marco Congedo, Anatole Lécuyer, Fabrice Lamarche, and Bruno Arnaldi. A review of classification algorithms for EEG-based brain–computer interfaces. *Journal of neural engineering*, 4(2):R1, 2007.

[Lom76] Nicholas R Lomb. Least-squares frequency analysis of unequally spaced data. *Astrophysics and space science*, 39(2):447–462, 1976.

[Mer14] Dirk Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux Journal*, 2014(239):2, 2014.

[PAG+99] Saul Perlmutter, G Aldering, G Goldhaber, RA Knop, P Nugent, PG Castro, S Deustua, S Fabbro, A Goobar, DE Groom, et al. Measurements of ω and λ from 42 high-redshift supernovae. *The Astrophysical Journal*, 517(2):565, 1999.

[PG07] Fernando Pérez and Brian E Granger. IPython: a system for interactive scientific computing. *Computing in Science & Engineering*, 9(3):21–29, 2007.

[PVG+11] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12(Oct):2825–2830, 2011.

[RFC+98] Adam G Riess, Alexei V Filippenko, Peter Challis, Alejandro Clocchiatti, Alan Diercks, Peter M Garnavich, Ron L Gilliland, Craig J Hogan, Saurabh Jha, Robert P Kirshner, et al. Observational evidence from supernovae for an accelerating universe and a cosmological constant. *The Astronomical Journal*, 116(3):1009, 1998.

[Ron15] Armin Ronacher. Flask (A Python Microframework), 2015.

[RSM+12] Joseph W Richards, Dan L Starr, Adam A Miller, Joshua S Bloom, Nathaniel R Butler, Henrik Brink, and Arien Crellin-Quick. Construction of a calibrated probabilistic classification catalog: Application to 50k variable sources in the all-sky automated survey. *The Astrophysical Journal Supplement Series*, 203(2):32, 2012.

[Sca82] Jeffrey D Scargle. Studies in astronomical time series analysis. ii-statistical aspects of spectral analysis of unevenly spaced data. *The Astrophysical Journal*, 263:835–853, 1982.

[SHG+14] D. Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, and Michael Young. Machine learning: The high interest credit card of technical debt. In *SE4ML: Software Engineering for Machine Learning (NIPS 2014 Workshop)*, 2014.

[Sol14] Ask Solem. Celery: Distributed task queue, 2014. URL: http://celeryproject.org.

[Sub07] Abdulhamit Subasi. EEG signal classification using wavelet feature extraction and a mixture of expert model. *Expert Systems with Applications*, 32(4):1084–1093, 2007.

[SV99] Johan AK Suykens and Joos Vandewalle. Least squares support vector machine classifiers. *Neural processing letters*, 9(3):293–300, 1999.

[T+13] R Core Team et al. R: A language and environment for statistical computing. 2013.

[VW12] Alvaro Videla and Jason JW Williams. *RabbitMQ in action*. Manning, 2012.

UConnRCMPy: Python-based data analysis for Rapid Compression Machines

Bryan W. Weber‡*, Chih-Jen Sung‡

https://youtu.be/tsjqkIAh8cw

Abstract—The ignition delay of a fuel/air mixture is an important quantity in designing combustion devices, and these data are also used to validate computational kinetic models for combustion. One of the typical experimental devices used to measure the ignition delay is called a Rapid Compression Machine (RCM). This paper presents UConnRCMPy, an open-source Python package to process experimental data from the RCM at the University of Connecticut. Given an experimental measurement, UConnRCMPy computes the thermodynamic conditions in the reaction chamber of the RCM during an experiment along with the ignition delay. UConnRCMPy relies on several packages from the SciPy stack and the broader scientific Python community. UConnRCMPy implements an extensible framework, so that alternative experimental data formats can be incorporated easily. In this way, UConnRCMPy improves the consistency of RCM data processing and enables reproducible analysis of the data.

Index Terms—rapid compression machine, engineering, kinetic models

Introduction

The world relies heavily on combustion to provide energy in useful and clean forms for human consumption; in particular, the transportation sector accounts for nearly 30% of the energy use in the United States and of that, more than 90% is supplied by combustion of fossil fuels [US 16]. Unfortunately, emissions from the combustion of traditional fossil fuels have been implicated in a host of deleterious effects on human health and the environment [ADF+02]. Two methods are being considered to reduce the impact of fossil fuel combustion in transportation on the environment, namely: 1) development of new fuel sources and 2) development of new engine technologies.

The challenge for engineers is that it is not straightforward to combine new fuels with newly designed engines. Employing computer-aided design and modeling of new engines with new fuels will be critical to develop advanced engines to be able to utilize multiple conventional and alternative fuels. The key to this process is the development of accurate and predictive combustion models.

These models of combustion are typically descriptions of the chemical kinetic pathways the hydrocarbon fuel and oxidizer undergo as they break down into carbon dioxide and water. There may be as many as several tens of thousands of pathways in the model for combustion of a particular fuel, with each pathway requiring several parameters to describe its rate. Therefore, it is important to thoroughly validate the operation of the model by comparison to experimental data collected over a wide range of conditions.

One type of data that is particularly relevant for transportation applications is the ignition delay. The ignition delay is a global combustion property depending on the interaction of many of the pathways present in the model. There are several methods to measure the ignition delay at engine-relevant conditions, including shock tubes and rapid compression machines (RCMs).

An RCM is typically designed with one or two pistons that rapidly compress a homogeneous fuel and oxidizer mixture inside a reaction chamber. After the end of compression (EOC), the piston(s) is (are) locked in place, creating a constant volume reaction chamber. The primary diagnostic in most RCM experiments is the pressure measured as a function of time in the reaction chamber. This pressure trace is then processed to extract the ignition delay.

In this paper, the design and operation of a software package to process the pressure data collected from RCMs is described. Our package, called UConnRCMPy [Web16], is designed to analyze the data acquired from the RCM at the University of Connecticut (UConn). Despite the initial focus on data from the UConn RCM, the package is designed to be extensible so that it can be used for data in different formats while providing a consistent interface to the user.

Recognizing that reproducible research is an important goal for the scientific community [Nat16], and that the code used to process experimental data is an important part of reproducing research, the primary goal of UConnRCMPy is to enable consistent, reproducible analysis of RCM data. Thus, UConnRCMPy offers all of the features required to process standard RCM data including:

- Filtering and smoothing the raw voltage generated by the pressure transducer
- Converting the voltage trace into a pressure trace using settings recorded from the RCM
- Processing the pressure trace to determine parameters of interest in reporting the experiments, including the ignition delay and machine-specific effects on the experiment
- Conducting simulations utilizing the experimental information to calculate the temperature during the experiment

Previous software used to analyze RCM data has generally

* Corresponding author: bryan.w.weber@gmail.com
‡ Mechanical Engineering Department, University of Connecticut, Storrs, CT 06269

been undocumented and untested code specific to the researcher conducting the experiments. Moreover, the software typically used to estimate the temperature in the experiments is difficult to integrate with the data processing code. To the best of the authors' knowledge, UConnRCMPy is the first package for analysis of standard RCM data to be presented in detail in the literature, and it tightly integrates the temperature estimation routine into the workflow, reducing errors and inefficiencies.

This paper serves to describe some of the important aspects of RCM data processing, particularly the choices that the operator must make that are rarely documented. In addition, as a complement to the in-source documentation, this paper documents the design choices, interface, and flexibility of UConnRCMPy.

Background

The RCMs at the University of Connecticut have been described extensively elsewhere [DSZM12], [MS07], and will be summarized here for reference. The RCMs use a single pneumatically accelerated and hydraulically decelerated piston. In a typical experiment, the reaction chamber is evacuated to an absolute pressure near 1 Torr, measured by a high-accuracy static pressure transducer. Next, the reactants are filled in to the desired initial pressure (P_0), and a valve on the reaction chamber is closed. Compression is triggered by a digital control circuit. After compression, the piston is held in place to create a constant volume chamber in which reactions proceed. For appropriate combinations of pressure, temperature, and mixture composition, ignition will occur after some delay period. A single compression-delay-ignition sequence is referred to as an experiment or a run. Each experiment is repeated approximately 5 times at the same nominal initial conditions to ensure repeatability of the data, and this set of experiments is referred to in the following as a condition.

The primary diagnostic on the RCM is the reaction chamber pressure, measured by a dynamic pressure transducer (separate from the static transducer used to measure P_0). The pressure trace is processed to determine the quantities of interest, including the pressure and temperature at the EOC, P_C and T_C respectively, and the ignition delay, τ. The ignition delay is typically measured at several values of T_C for a given value of P_C and mixture composition; this is referred to in the following as a data set.

RCM Signal Processing Procedure

Signal measurement

The dynamic pressure transducer outputs a charge signal that is converted to a voltage signal by a charge amplifier with a nominal output of 0 V prior to the start of compression. In addition, the output range of 0 V to 10 V is set by the operator to correspond to a particular pressure range by setting a "scale factor". Typical values for the scale factor range between 10 bar/V and 100 bar/V.

The voltage output from the charge amplifier is digitized by a hardware data acquisition system (DAQ) and recorded into a plain text file by a LabView Virtual Instrument. The voltage is sampled at a rate chosen by the operator, typically between 50 kHz and 100 kHz. This provides sufficient resolution for events on the order of milliseconds; the typical ignition delay measured with this RCM approximately ranges from 5 ms to 100 ms.

Figure 1 shows a typical voltage trace measured from the RCM at UConn. Several features are apparent from this figure. First, the compression stroke takes approximately 30 ms to 40 ms and approximately 50% of the pressure rise occurs in the last 5 ms of compression. Second, there is a slow pressure decrease after the EOC due to heat transfer from the reactants to the relatively colder chamber walls. Third, after some delay period there is a spike in the pressure corresponding to rapid heat release due to combustion. Finally, the signal can be somewhat noisy, requiring filtering and/or smoothing to produce a useful pressure trace.

Filtering and Smoothing

In the current version of UConnRCMPy [Web16], the voltage is filtered using a low-pass filter with a cutoff frequency of 10 kHz. The filter is constructed using the `firwin()` function from the `signals` module of SciPy [JOPosh] with the Blackman window [BT58], [OSB99] and a filter order of $2^{14}-1$. The cutoff frequency, window type, and filter order were determined empirically, based on Fig. 2. Methods to select a cutoff frequency that optimizes the signal-to-noise ratio are currently being investigated.

After filtering, the signal is smoothed by a moving average filter with a width of 21 points. This width was selected empirically based on Fig. 1 to minimize the deviation of the smoothed voltage from the raw voltage during the ignition, and methods to automatically choose an optimal width are being investigated. It is desired that the signal remain the same length through this operation, but the convolution operation used to apply the moving average zero-pads the first and last 10 points. To avoid a bias in the initial voltage, the first 10 points are set equal to the value of the 11th point; the final 10 points are not important in the rest of the analysis and are ignored. The result of the filtering and smoothing operations is shown on Fig. 1.

Offset Correction and Pressure Calculation

In general, the voltage trace can be converted to a pressure trace by

$$P(t) = F \cdot \overline{V}(t) + P_0 \tag{1}$$

where $\overline{V}(t)$ is the filtered and smoothed voltage trace and F is the scale factor from the charge amplifier. However, as can be seen in Fig. 1b there is a small offset in the initial voltage relative to the nominal value of 0 V. To correct for this offset, it can be subtracted from the voltage trace

$$P(t) = F \cdot \left[\overline{V}(t) - \overline{V}(0)\right] + P_0 \tag{2}$$

where $\overline{V}(0)$ is the initial voltage of the filtered and smoothed signal. Assuming the noise in the signal has an equal probability of being above or below the mean voltage, choosing the initial point (i.e., $\overline{V}(0)$) to set the voltage offset is equivalent to choosing any other point prior to the start of compression. The result is a vector of pressure values that must be further processed to determine the time of the EOC and the ignition delay.

Finding the EOC

In the current version of UConnRCMPy [Web16], the EOC is determined by finding the local maximum of the pressure prior to ignition. This is done by searching backwards in time from the global maximum pressure in the pressure trace (typically, the global maximum of the pressure is due to ignition) until a minimum in the pressure is reached. Since the precise time of the minimum is not important for this method, the search is done by comparing the pressure at a given index i to the pressure at point $i-50$, starting with the index of the global maximum pressure. The comparison is not made to the adjacent point to avoid the influence of noise. If $P(i) \geq P(i-50)$, the index is decremented

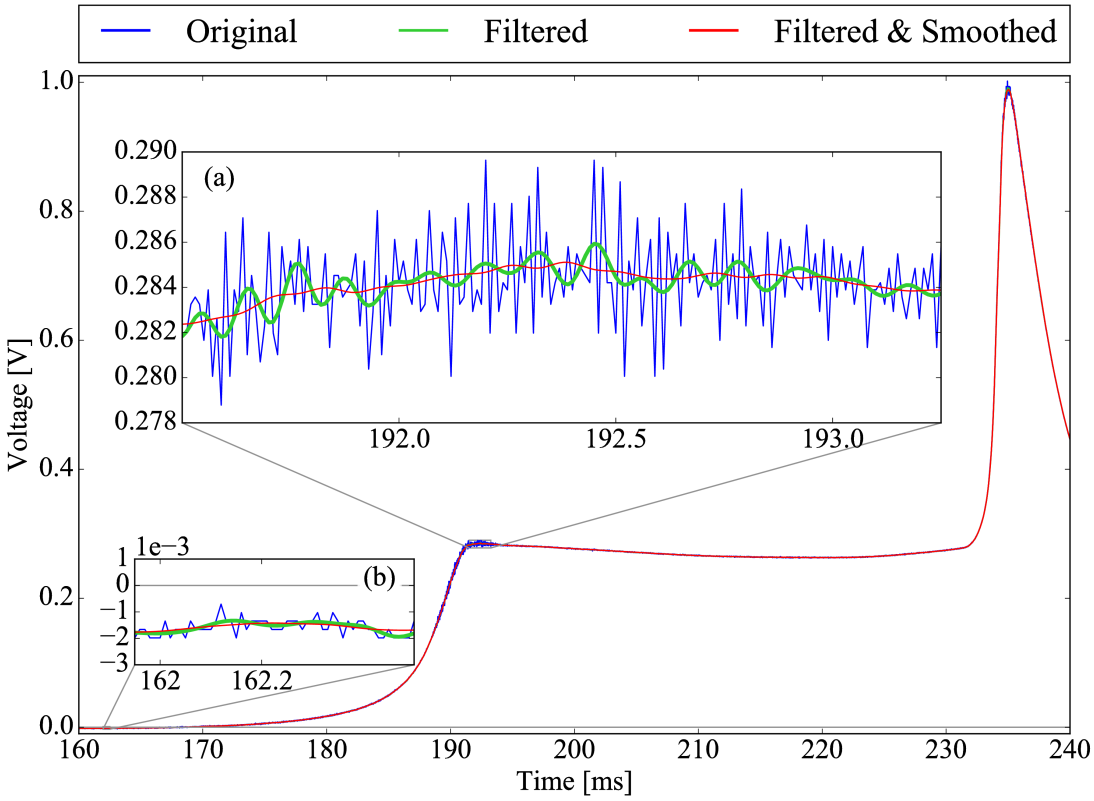

Fig. 1: *Raw voltage trace and the voltage trace after filtering and smoothing from a typical RCM experiment. Note that the voltage in the figure varies from 0 V to 1 V because the scale factor is 100 bar/V and the maximum pressure for this case is near 100 bar. (a): Close up of the time around the EOC, demonstrating the fidelity of the smoothed and filtered signal with the original signal. (b): Close up of the time before the start of compression, demonstrating the offset of the initial voltage slightly below 0 V.*

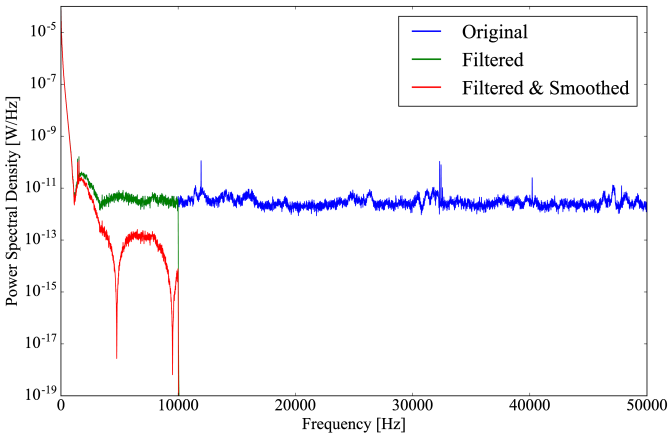

Fig. 2: *Power spectral density profiles of the original, filtered, and filtered and smoothed signals, showing the peaks of noise above 10 kHz.*

and the process is repeated until $P(i) < P(i-50)$. This value of i is approximately at the minimum of pressure prior to ignition, so the maximum of the pressure in points to the left of the minimum will be the EOC.

This method is generally robust, but it fails when there is no minimum in the pressure between the EOC and ignition, or the minimum pressure is very close to the EOC pressure. This may be the case for short ignition delays, on the order of 5 ms or less. In these cases, the comparison offset (which is set to 50 points by default) can be reduced to improve the granularity of the search; if the method still fails, manual intervention is necessary to determine the EOC. In either case, the value of the pressure at the EOC, P_C, is recorded and the time at the EOC is taken to be $t = 0$.

Calculating Ignition Delay

The ignition delay is determined as the time difference between the EOC and the point of ignition. There are several definitions of the point of ignition; the most commonly used in RCM experiments is the inflection point in the pressure trace due to ignition. As before, finding zero crossings of the second time derivative of the pressure to define the inflection point is difficult due to noise; however, finding the maximum of the first derivative is trivial, particularly since the time before and shortly after the EOC can be excluded to avoid the peak in the derivative around the EOC.

In the current version of UConnRCMPy [Web16], the first derivative of the experimental pressure trace is computed by a second-order forward differencing method. The derivative is then smoothed by the moving average algorithm with a width of 151 points. This value for the moving average window was chosen empirically.

For some conditions, the reactants may undergo two distinct stages of ignition. These cases can be distinguished by a pair of peaks in the first time derivative of the pressure. For some two-stage ignition cases, the first-stage pressure rise, and consequently the peak in the derivative, are relatively weak, making it hard to distinguish the peak due to ignition from the background noise.

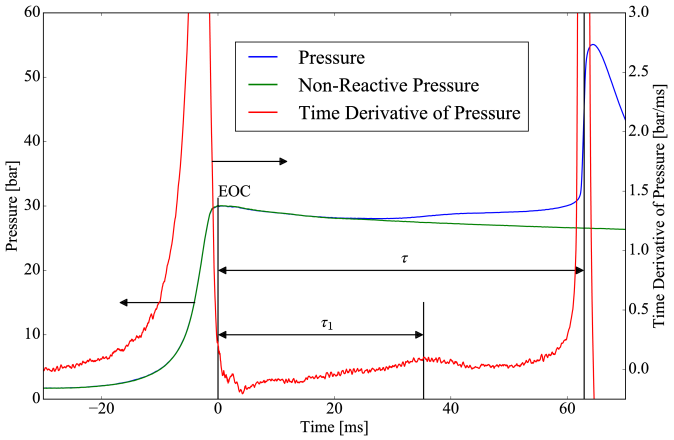

Fig. 3: *Illustration of the definition of the ignition delay in a two-stage ignition case.*

This is currently the area requiring the most manual intervention, and one area where significant improvements can be made by refining the differentiation and filtering/smoothing algorithms. An experiment that shows two clear peaks in the derivative is shown in Fig. 3 to demonstrate the definition of the ignition delays.

Calculating the EOC Temperature

The final parameter of interest presently is the EOC temperature, T_C. This temperature is often used as the reference temperature when reporting ignition delays. In general, it is difficult to measure the temperature as a function of time in the reaction chamber of the RCM, so methods to estimate the temperature from the pressure trace are generally used.

The law of conservation of energy written for the ideal gases in the reaction chamber is:

$$c_v \frac{dT}{dt} = -P\frac{dv}{dt} - \sum_k u_k \frac{dY_k}{dt} \tag{3}$$

where c_v is the specific heat at constant volume of the mixture, v is the specific volume, u_k and Y_k are the specific internal energy and mass fraction of the species k, and t is time. For a constant-area piston, the rate of change of the volume is equal to the piston velocity. In UConnRCMPy, Eq. 3 is integrated by Cantera [GMS16].

In Cantera, intensive thermodynamic information about the system is stored in an instance of the `Solution` class. The `Solution` classes used in this study model simple, compressible systems and require two independent properties, plus the composition, to fix the state. The two properties must be intensive (i.e., not dependent on system size), and are typically chosen from the pressure, temperature, and density. The thermodynamic information for each species is read from a file in the CTI format, described in the Cantera documentation [GMS16], when a `Solution` instance is created.

In addition to evaluating thermodynamic data, Cantera [GMS16] contains several objects used to model homogeneous reacting systems; the two used in UConnRCMPy are the `Reservoir` and the `IdealGasReactor`, which are subclasses of the generic `Reactor` class. A `Solution` object is installed in each `Reactor` subclass instance to manage the state information and evaluate thermodynamic properties. The difference between the `Reservoir` and the `IdealGasReactor` is simply that the state (i.e., the pressure, temperature, and chemical composition) of the `Solution` in a `Reservoir` is fixed.

Integrating Eq. 3 requires knowledge of the volume of the reaction chamber as a function of time. To calculate the volume as a function of time, it is assumed that there is a core of gas in the reaction chamber that undergoes an isentropic compression [LH98]. Furthermore, it is assumed that there is negligible reactant consumption during the compression stroke.

Constructing the volume trace is triggered by the user by running the `create_volume_trace()` method that implements the following procedure. A Cantera `Solution` object is initialized at the initial temperature, pressure, and composition of the reaction chamber. After initialization, UConnRCMPy stores the initial mass-specific entropy (s_0) and density (ρ_0). The initial volume is arbitrarily taken to be $V_0 = 1.0\,\mathrm{m}^3$. The initial volume used in constructing the volume trace is arbitrary provided that the same value is used for the initial volume in the simulations described below. However, extensive quantities such as the total heat release during ignition cannot be compared to experimental values.

The measured pressure at each point in the pressure trace (P_i) is used with the previously recorded initial entropy (s_0) to set the state of the `Solution` object sequentially. At each point, the volume is computed by applying the ideal gas law:

$$V_i = V_0 \frac{\rho_0}{\rho_i} \tag{4}$$

where ρ_i is the density at each point computed by the Cantera `Solution`. This procedure effects a constant composition isentropic compression process.

Once the volume trace has been generated, it can be utilized in the `IdealGasReactor` and the solution of Eq. 3 by installing an instance of the `Wall` class. `Walls` must be installed between instances of `Reactors`, so in UConnRCMPy a `Wall` is installed between the `IdealGasReactor` that represents the reaction chamber and an instance of the `Reservoir` class. By specifying the velocity of the `Wall` using the volume trace, the `IdealGasReactor` will proceed through the same states as the reaction chamber in the experiment. The velocity of the `Wall` is specified by using an instance of the `VolumeProfile` class from the CanSen software [Web15], which computes the first forward difference of the volume as a function of time.

Finally, the `IdealGasReactor` is installed into an instance of `ReactorNet` from Cantera [GMS16]. The `ReactorNet` implements the interface to the solver CVODES. CVODES is an adaptive-time-stepping solver, distributed as part of the SUNDIALS suite [HBG+05].

Two simulations can be triggered by the user that utilize this procedure. In the first, the multiplier for all the reaction rates is set to zero, to simulate a constant composition (non-reactive) process. In the second, the reactions are allowed to proceed as normal. Only the non-reactive simulation is necessary to determine T_C, which is defined as the simulated temperature at the EOC time.

When a reactive simulation is conducted, the user must compare the temperature traces from the two simulations to verify that the inclusion of the reactions does not change T_C, validating the assumption of adiabatic, constant composition compression. Although including reactions during the compression stroke does not affect the value of T_C, it does allow for the buildup of a small pool of radicals that can affect processes after the EOC [MCSD08]. Thus, it is critical to include reactions during the

compression stroke when conducting simulations to compare a kinetic model to experimental results.

Simulating Post-EOC Processes

As can be seen in Fig. 3, the pressure decreases after the EOC due to heat transfer from the higher temperature reactants to the reaction chamber walls. This process is specific to the machine that carried out the experiments, and to the conditions under which the experiment was conducted. Therefore, the rate of pressure decrease should be modeled and included in simulations that compare predicted ignition delays to the experimental values.

To conduct this modeling, a non-reactive experiment is conducted, where O_2 in the oxidizer is replaced with N_2 to maintain a similar specific heat ratio but suppress the oxidation reactions that lead to ignition. The pressure trace from this non-reactive experiment should closely match that from the reactive experiment during the compression stroke, further validating the assumption of adiabatic, constant composition compression. Furthermore, the non-reactive pressure trace should closely match the reactive pressure trace after the EOC until exothermic reactions cause the pressure in the reactive experiment to begin to increase.

To apply the effect of the post-compression heat loss into the simulations, the reaction chamber is modeled as undergoing an adiabatic volume expansion. Since the post-compression time is modeled as an isentropic expansion, the same procedure is used as in the computation of T_C to compute a volume trace for the post-EOC time. The only difference is that the non-reactive pressure trace is used after the EOC instead of the reactive pressure trace. Once the volume trace is generated, it can be applied to a simulation by concatenating the volume trace of the compression stroke and the post-EOC volume trace together and following the procedure outlined in Calculating the EOC Temperature. For consistency, the ignition delay in a reactive simulation is defined in the same manner as in the reactive experiments, as the maxima of the time derivative of the pressure trace. This procedure has been validated experimentally by measuring the temperature in the reaction chamber during and after the compression stroke. The temperature of the reactants was found to be within ± 5 K of the simulated temperature [DUS12], [UDS12].

Implementation of UConnRCMPy

UConnRCMPy is constructed in a hierarchical manner. The main user interface to UConnRCMPy is through the `Condition` class, the highest level of data representation. The `Condition` class contains all of the information pertaining to the experiments at a given condition. The intended use of this class is in an interactive Python interpreter (the author prefers the Jupyter Notebook with an IPython kernel [PG07]). `Condition` also contains all the methods that make up the user interface:

- `add_experiment()`
- `create_volume_trace()`
- `compare_to_sim()`

The usage of these methods will be described in detail in the Usage Example section. In general, the user will conduct several experiments and, using the `add_experiment()` method, will trigger UConnRCMPy to create instances of the `Experiment` class and extract the ignition delay.

All of the information about a particular experimental run is stored in the `Experiment` class. When initialized, the `Experiment` expects an instance of the `pathlib.Path` class; if none is provided, it prompts the user to enter a file name that is expected to be in the current working directory. The file name should point to a tab-delimited plain text file that contains the voltage trace recorded by LabView from one experimental run. Then UConnRCMPy creates an instance of `VoltageTrace`, followed by an instance of `ExperimentalPressureTrace`. The pressure trace from the latter is processed to extract the ignition delay(s).

The lowest level representation of data in UConnRCMPy is the `VoltageTrace` that contains the raw voltage signal and timing recorded from the DAQ, as well as the filtered and smoothed voltage traces. The filtering and smoothing algorithms are implemented as separate methods so they can be reused in other situations and are run automatically when the `VoltageTrace` is initialized.

One step up from the `VoltageTrace` is the `ExperimentalPressureTrace` class. This class consumes a `VoltageTrace` and processes it into a pressure trace, given the multiplication factor from the charge amplifier and the initial pressure. This class also contains methods to compute the derivative of the experimental pressure trace, as discussed in Calculating Ignition Delay.

When all the experiments are conducted and processed, `create_volume_trace()` further processes the experiments to create the volume trace necessary to run the simulations to determine T_C. The actual computation of the volume trace is done by the `VolumeFromPressure` class. First, the volume trace of the pre-EOC portion is generated using the pre-EOC pressure trace, the experimental initial temperature, and an initial volume of $V_0 = 1.0\,\mathrm{m}^3$, as discussed in Calculating the EOC Temperature. A temperature trace is also constructed for the pre-EOC pressure trace using the `TemperatureFromPressure` class.

For the post-EOC volume trace, the initial temperature is estimated as the final value of the temperature trace constructed for the pre-EOC period. Furthermore, the initial volume of the post-EOC volume trace is taken to be the final value of the pre-EOC volume trace, so that although there may be small mismatches in P_C, the volume trace will be consistent.

After generation, `create_volume_trace()` writes the volume trace out to a CSV file so that the volume trace can be used in other software. The reactive pressure trace is also written to a tab-separated file. Before writing, the volume and pressure traces are both downsampled by a factor of 5. This reduces the computational time of a simulation and does not have any effect on the simulated results. `create_volume_trace()` also generates a figure that plots the complete reactive pressure trace, a non-reactive pressure trace generated from the volume trace using the `PressureFromVolume` class, and a linear fit to the constant pressure period prior to the start of compression. This linear fit aids in determining a suitable compression time. Finally, the value of the pressure at the beginning of compression is put on the system clipboard to be pasted into a spreadsheet to record the P_0 used for simulations. This may differ slightly from the P_0 read from the static transducer due to noise in the signal.

The final step is to use the volume trace in a simulation to determine T_C. To begin the simulations, the user calls the `compare_to_sim()` method of the `Condition`. The `compare_to_sim()` method relies on the `run_simulation()` method, which in turn adds instances of the class `Simulation` to the `Condition` instance. Instances

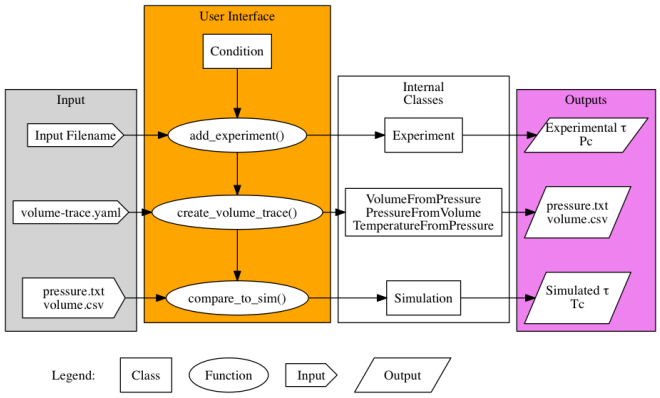

Fig. 4: *Flowchart of information in UConnRCMPy.*

of `Simulation` can represent a reactive or a non-reactive experiment; if either type of simulation has already been added to the `Condition` instance, the user is asked whether they would like to overwrite the existing simulation.

The `Simulation` class sets up the simulation in Cantera and importantly, sets the maximum time step to be the time step used in the volume trace, so that the solver does not take steps larger than the resolution of the velocity. Larger time steps may result in incorrect calculation of the state if the velocity is not properly applied to the reactor. The time, temperature, pressure, and simulated volume are stored in NumPy arrays [vCV11] and the derivative is computed using second order Lagrange polynomials, as suggested by Chapra and Canale [CC10] because the time step is not constant in the simulation. Finally, the calculated value of T_C is placed into the system clipboard. If the reactive simulation is conducted, the overall ignition delay is also copied into the system clipboard. The first stage ignition delay must be found manually because determining peaks in the derivative is currently unreliable, as mentioned in Calculating Ignition Delay for experiments.

The `compare_to_sim()` method also plots the experimental pressure trace and any of the simulated pressure traces that have been generated. If the simulated reactive pressure trace is generated, the time derivative of the pressure is also plotted, where the derivative is scaled by the maximum pressure in the reactive simulation.

The general flow of the user interaction with UConnRCMPy is shown in Fig. 4. The Inputs are required input from the user, while the User Interface are classes and functions called by the user during processing.

UConnRCMPy is documented using standard Python docstrings for functions and classes. The documentation is converted to HTML files by the Sphinx documentation generator [Bra16]. The format of the docstrings conforms to the NumPy docstring format so that the autodoc module of Sphinx can be used. The documentation is available on the web at https://bryanwweber.github.io/UConnRCMPy/.

Usage Example

In the following, two examples of using UConnRCMPy are given, first with the standard interface and second utilizing a slightly modified interface corresponding to a different data format. Both examples assume the user is running in a Jupyter Notebook with an IPython kernel.

Standard Interface

These experiments were conducted with mixtures of propane, oxygen, and nitrogen [DRW+16]. The CTI file necessary to run this example can be found in the Supplementary Material of the work by Dames et al. [DRW+16]. It must be named exactly `species.cti` and placed in the current working directory. Then, the composition of the mixture under consideration must be added to the `initial_state` parameter of the `ideal_gas()` method:

```
ideal_gas(
    name='gas',
    elements=...,
    species=...,
    reactions='all',
    initial_state=state(
        temperature=300.0, pressure=OneAtm,
        mole_fractions=(
            'C3H8:0.0403,O2:0.1008,N2:0.8589')))
```

Ellipses indicate input that was truncated to save space; the truncated input is present in the file available with the work of Dames et al. The initial temperature and pressure are arbitrary, since those are set based on information stored in the filename of the experiment, but the `mole_fractions` must be set to the appropriate values. The condition in this example is for a fuel rich mixture, with a target P_C of 30 bar. The user creates the `Condition`, then conducts a reactive experiment with the RCM and adds the experiment to the `Condition` using the `add_experiment()` method. This method creates an instance of class `Experiment` for each experiment passed in. As each experiment is processed by UConnRCMPy, the information from that run is added to the system clipboard for pasting into some spreadsheet software. In the current version, the information copied is the time of day of the experiment, the initial pressure, the initial temperature, the pressure at the EOC, the overall and first stage ignition delays, an estimate of the EOC temperature, and some information about the compression ratio of the reactor. This process is repeated 5 times to ensure repeatable data is obtained.

```
from uconnrcmpy import Condition
from pathlib import Path
%matplotlib

cond_00_in_02_mm = Condition()
# Conduct reactive experiment #1 on the RCM
cond_00_in_02_mm.add_experiment(Path(
    '00_in_02_mm_373K-1285t-100x-19-Jul-15-1620.txt'))
# Conduct reactive experiment #2 on the RCM
cond_00_in_02_mm.add_experiment(Path(
    '00_in_02_mm_373K-1282t-100x-19-Jul-15-1626.txt'))
# Conduct reactive experiment #3 on the RCM
cond_00_in_02_mm.add_experiment(Path(
    '00_in_02_mm_373K-1282t-100x-19-Jul-15-1633.txt'))
# Conduct reactive experiment #4 on the RCM
cond_00_in_02_mm.add_experiment(Path(
    '00_in_02_mm_373K-1282t-100x-19-Jul-15-1640.txt'))
# Conduct reactive experiment #5 on the RCM
cond_00_in_02_mm.add_experiment(Path(
    '00_in_02_mm_373K-1282t-100x-19-Jul-15-1646.txt'))
```

This sequence generates one figure showing all of the experiments together and one figure per experiment comparing the pressure and the time derivative of the pressure. Matplotlib is used for plotting [Hun07]. The plots are optional, and are controlled by passing a boolean keyword argument `plotting` when the `Condition` is initialized. The figures showing each experiment look similar to Fig. 3, but the non-reactive trace is not plotted and the EOC and ignition delays are not labeled.

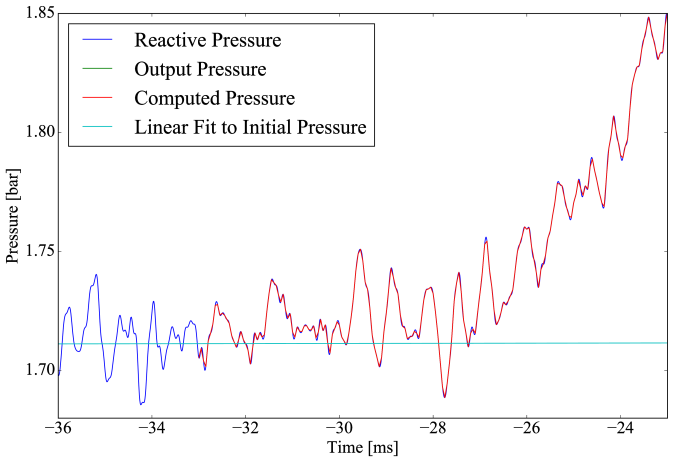

Fig. 5: *Comparison of the reactive pressure trace, the pressure trace output to the text file, the pressure trace computed from the volume trace, and the linear fit to the initial pressure demonstrating the choice of compression time. The dark blue, green, and red lines follow each other nearly exactly after the start of compression, so only the red line is visible. This is the desired result, indicating that the pressure traces agree.*

In general, for a given condition, the user will conduct and process all of the reactive experiments before conducting any non-reactive experiments. Then, the user chooses one of the reactive experiments as the reference experiment for the condition (i.e., the one whose ignition delay(s) and T_C are reported) by inspection of the data in the spreadsheet. The reference experiment is defined as the experimental run whose overall ignition delay is closest to the mean overall ignition delay among the experiments at a given condition. To select the reference experiment, the user puts the file name of the reference experiment into a YAML format file called `volume-trace.yaml` with the key `reacfile`. For this case, the reference experiment is the run that took place at 16:33:

```
reacfile: >
  00_in_02_mm_373K-1282t-100x-19-Jul-15-1633.txt
```

Note that the file must be named exactly `volume-trace.yaml` and it must be located in the current working directory. Once the reference reactive experiment is selected, the user runs non-reactive experiments at the same initial conditions as the reference experiment. The user adds non-reactive experiments to the `Condition` by the same `add_experiment()` method and UConnRCMPy automatically determines whether the experiment is reactive or non-reactive.

```
# Conduct non-reactive experiment #1 on the RCM
cond_00_in_02_mm.add_experiment(Path(
'NR_00_in_02_mm_373K-1278t-100x-19-Jul-15-1652.txt'))
```

UConnRCMPy determines that this is a non-reactive experiment and generates a new figure that compares the current non-reactive case with the reference reactive case as specified in `volume-trace.yaml`. If the user adds a non-reactive experiment before creating the `volume-trace.yaml` file, or if the file referenced in the `reacfile` key is not present in the current working directory, UConnRCMPy throws a `FileNotFound` exception. For this particular example, the pressure traces are shown in Fig. 3. In this case, the non-reactive pressure agrees very well with the reactive pressure and no further experiments are necessary; in principle, any number of non-reactive experiments can be conducted and added to the figure for comparison. Since there is good agreement between the non-reactive and reactive pressure traces, the user adds the non-reactive reference file name to `volume-trace.yaml`.

```
reacfile: >
  00_in_02_mm_373K-1282t-100x-19-Jul-15-1633.txt
nonrfile: >
  NR_00_in_02_mm_373K-1278t-100x-19-Jul-15-1652.txt
```

Then, the user specifies the rest of the parameters in `volume-trace.yaml`, including the compression time and the end times for the reactive and non-reactive experiments. The reactive end time (`reacend`) determines the length of the output pressure trace, while the non-reactive end time (`nonrend`) determines the length of the volume trace. The length of the volume trace is also determined by the compression time (`comptime`), which should be set to a time such that the starting point is before the beginning of the compression. All three times should be specified in milliseconds. `comptime` is determined by comparison with the fit to the initial pressure, as shown in Fig. 5. In this case, the compression has started at approximately $t > -28$ ms. The time prior to that where the pressure appears to stabilize around the initial pressure is approximately $t = -33$ ms, giving a compression time of 33 ms. `reacend` is typically chosen to be shortly after the main pressure peak due to ignition, about 80 ms in this case, and `nonrend` is typically chosen to be 400 ms.

```
reacfile: >
  00_in_02_mm_373K-1282t-100x-19-Jul-15-1633.txt
nonrfile: >
  NR_00_in_02_mm_373K-1278t-100x-19-Jul-15-1652.txt
comptime: 33
nonrend: 400
reacend: 80
```

This sample represents a complete, minimal example of the necessary information in the `volume-trace.yaml` file. In addition, two optional parameters can also be specified in `volume-trace.yaml`. These are offset parameters used to control the precise point where the switch from the reactive pressure trace to the non-reactive pressure trace occurs in the volume trace. These parameters may be necessary if the determination of the EOC does not result in aligned compression strokes for the reactive and non-reactive experiments, but they are not generally necessary.

Once the `volume-trace.yaml` file is completed, the `create_volume_trace()` method can be run. Then, the final step is to conduct the simulations to calculate T_C and the simulated ignition delay. This is done by the user by running the `compare_to_sim()` function. This function takes two optional arguments, `run_reactive()` and `run_nonreactive()`, both of which are booleans. These determine which type of simulation should be conducted; by default, `run_reactive()` is `False` and `run_nonreactive()` is `True` because the reactive simulations may take substantial time (~5 min). There is no restriction on combinations of values for the arguments; either or both may be `True` or `False`.

```
cond_00_in_02_mm.create_volume_trace()
cond_00_in_02_mm.compare_to_sim(
    run_reactive=True,
    run_nonreactive=True,
)
```

At this point, the user has completed one experimental condition. Now, further conditions should be studied, either by changing T_0 or the compression ratio of the RCM to reach a different value of T_C for a given P_C.

Modified Interface

It is also possible to replace parts of the processing interface by using the features of Python to overload class methods. Due to the modular nature of UConnRCMPy, small parts of the interface can be replaced without sacrificing consistent analysis for the critical parts of the code, such as computing the ignition delay. For instance, ongoing work involves processing RCM data collected by several operators of the RCM. Each user has their own file naming strategy that must be parsed for information about the experiment. To process this "alternate" data format, two new classes called `AltCondition` and `AltExperiment` are created that inherit from the `Condition` and `Experiment` classes, respectively. The `AltCondition` class only needs to overload the `add_experiment()` method, to create an `AltExperiment`, instead of a regular `Experiment`.

```
class AltCondition(Condition):
    def add_experiment(self, file_name=None):
        exp = AltExperiment(file_name)
        # Omit the plotting code...
```

Then, the `AltExperiment` overloads the `parse_file_name()` method of the `Experiment` class to parse the alternate format. The user must make sure the new `parse_file_name()` method returns the expected values as defined in the docstring for the original `parse_file_name()` method, or else overload other methods that consume the file name information.

```
class AltExperiment(Experiment):
    def parse_file_name(self, file_path):
        # Parse the file name for information...
        return file_name_information
```

In this way, consistent definitions for important research quantities can be used, while providing flexibility in the data format and naming conventions.

Conclusions and Future Work

UConnRCMPy provides a framework to enable consistent analysis of RCM data. Because it is open source and extensible, UConnRCMPy can help to ensure that RCM data in the community can be analyzed in a reproducible manner; in addition, it can be easily modified and used for data in any format. In this sense, UConnRCMPy can be used more generally to process any RCM experiments where the ignition delay is the primary output.

Future plans for UConnRCMPy include the development of a robust test suite to prevent regressions and document correct usage of the framework, as well as the development of a method to determine the optimal cutoff frequency in the filtering algorithm.

Acknowledgements

This paper is based on material supported by the National Science Foundation under Grant No. CBET-1402231.

References

[ADF+02] Maureen D Avakian, Barry Dellinger, Heidelore Fiedler, Brian Gullet, Catherine Koshland, Stellan Marklund, Günter Oberdörster, Stephen Safe, Adel Sarofim, Kirk R Smith, David Schwartz, and William A Suk. The origin, fate, and health effects of combustion by-products: a research framework. *Environmental Health Perspectives*, 110(11):1155–1162, November 2002. URL: http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=1241073&tool=pmcentrez&rendertype=abstract.

[Bra16] Georg Brandl. Overview — Sphinx 1.4.1 documentation, 2016. URL: http://www.sphinx-doc.org/en/stable/.

[BT58] Ralph Beebe Blackman and John Wilder Tukey. *The Measurement of Power Spectra*. Dover, 1958. URL: https://archive.org/details/TheMeasurementOfPowerSpectra.

[CC10] Steven C. Chapra and Raymond P. Canale. *Numerical methods for engineers*. McGraw-Hill Higher Education, Boston, 6th ed edition, 2010.

[DRW+16] Enoch E. Dames, Andrew S. Rosen, Bryan W. Weber, Connie W. Gao, Chih-Jen Sung, and William H. Green. A detailed combined experimental and theoretical study on dimethyl ether/propane blended oxidation. *Combustion and Flame*, 168:310–330, June 2016. doi:10.1016/j.combustflame.2016.02.021.

[DSZM12] Apurba Kumar Das, Chih-Jen Sung, Yu Zhang, and Gaurav Mittal. Ignition delay study of moist hydrogen/oxidizer mixtures using a rapid compression machine. *International Journal of Hydrogen Energy*, 37(8):6901–6911, April 2012. doi:10.1016/j.ijhydene.2012.01.111.

[DUS12] Apurba Kumar Das, Mruthunjaya Uddi, and Chih-Jen Sung. Two-line thermometry and H2O measurement for reactive mixtures in rapid compression machine near 7.6μm. *Combustion and Flame*, 159(12):3493–3501, December 2012. doi:10.1016/j.combustflame.2012.06.020.

[GMS16] David G. Goodwin, Harry K. Moffat, and Raymond L. Speth. Cantera: An Object-oriented Software Toolkit for Chemical Kinetics, Thermodynamics, and Transport Processes, 2016. URL: http://www.cantera.org.

[HBG+05] Alan C. Hindmarsh, Peter N. Brown, Keith E. Grant, Steven L. Lee, Radu Serban, Dan E. Shumaker, and Carol S. Woodward. SUNDIALS: Suite of nonlinear and differential/algebraic equation solvers. *ACM Transactions on Mathematical Software*, 31(3):363–396, September 2005. doi:10.1145/1089014.1089020.

[Hun07] John D. Hunter. Matplotlib: A 2D Graphics Environment. *Computing in Science & Engineering*, 9(3):90–95, 2007. doi:10.1109/MCSE.2007.55.

[JOPosh] Eric Jones, Travis Oliphant, Pearu Peterson, and others. *SciPy: Open source scientific tools for Python*. 2001–. [Online; accessed 2016-05-19]. URL: http://www.scipy.org/.

[LH98] Daeyup Lee and Simone Hochgreb. Rapid Compression Machines: Heat Transfer and Suppression of Corner Vortex. *Combustion and Flame*, 114(3-4):531–545, August 1998. doi:10.1016/S0010-2180(97)00327-1.

[MCSD08] Gaurav Mittal, Marcos Chaos, Chih-Jen Sung, and Frederick L. Dryer. Dimethyl ether autoignition in a rapid compression machine: Experiments and chemical kinetic modeling. *Fuel Processing Technology*, 89(12):1244–1254, December 2008. doi:10.1016/j.fuproc.2008.05.021.

[MS07] Gaurav Mittal and Chih-Jen Sung. A Rapid Compression Machine for Chemical Kinetics Studies at Elevated Pressures and Temperatures. *Combustion Science and Technology*, 179(3):497–530, 2007. doi:10.1080/00102200600671898.

[Nat16] Reality check on reproducibility. *Nature*, 533(7604):437–437, May 2016. doi:10.1038/533437a.

[OSB99] Alan V. Oppenheim, Ronald W. Schafer, and John R. Buck. *Discrete-time signal processing*. Prentice Hall, Upper Saddle River, N.J, 2nd ed edition, 1999.

[PG07] Fernando Pérez and Brian E. Granger. IPython: a System for Interactive Scientific Computing. *Computing in Science and Engineering*, 9(3):21–29, May 2007. doi:10.1109/MCSE.2007.53.

[UDS12] Mruthunjaya Uddi, Apurba Kumar Das, and Chih-Jen Sung. Temperature measurements in a rapid compression machine using mid-infrared H2O absorption spectroscopy near 7.6 μm. *Applied Optics*, 51(22):5464–5476, August 2012. URL: http://www.ncbi.nlm.nih.gov/pubmed/22859037.

[US 16] US Energy Information Administration. EIA Monthly Energy Review. Technical Report DOE/EIA-0035(2016/4), April 2016. URL: http://www.eia.gov/totalenergy/data/monthly/archive/00351604.pdf.

[vCV11] Stéfan van der Walt, S Chris Colbert, and Gaël Varoquaux. The NumPy Array: A Structure for Efficient Numerical Computation. *Computing in Science & Engineering*, 13(2):22–30, March 2011. doi:10.1109/MCSE.2011.37.

[Web15] Bryan William Weber. CanSen, June 2015. URL: https://github.com/bryanwweber/CanSen.

[Web16] Bryan William Weber. UConnRCMPy, May 2016. URL: https://github.com/bryanwweber/UConnRCMPy.

Storing Reproducible Results from Computational Experiments using Scientific Python Packages

Christian Schou Oxvig[‡*], Thomas Arildsen[‡], Torben Larsen[‡]

Abstract—Computational methods have become a prime branch of modern science. Unfortunately, retractions of papers in high-ranked journals due to erroneous computations as well as a general lack of reproducibility of results have led to a so-called credibility crisis. The answer from the scientific community has been an increased focus on implementing reproducible research in the computational sciences. Researchers and scientists have addressed this increasingly important problem by proposing best practices as well as making available tools for aiding in implementing them. We discuss and give an example of how to implement such best practices using scientific Python packages. Our focus is on how to store the relevant metadata along with the results of a computational experiment. We propose the use of JSON and the HDF5 database and detail a reference implementation in the Magni Python package. Further, we discuss the focuses and purposes of the broad range of available tools for making scientific computations reproducible. We pinpoint the particular use cases that we believe are better solved by storing metadata along with results the same HDF5 database. Storing metadata along with results is important in implementing reproducible research and it is readily achievable using scientific Python packages.

Index Terms—Reproducibility, Computational Science, HDF5

Introduction

Exactly how did I produce the computational results stored in this file? Most data scientists and researchers have probably asked this question at some point. For one to be able to answer the question, it is of utmost importance to track the provenance of the computational results by making the computational experiment reproducible, i.e. describing the experiment in such detail that it is possible for others to independently repeat it [LMS12], [Hin14]. Unfortunately, retractions of papers in high-ranked journals due to erroneous computations [Mil06] as well as a general lack of reproducibility of computational results [Mer10], with some studies showing that only around 10% of computational results are reproducible [BE12], [RGPN+11], have led to a so-call *credibility crisis* in the computational sciences.

The answer has been a demand for requiring research to be reproducible [Pen11]. The scientific community has acknowledged that many computational experiments have become so complex that more than a textual presentation in a paper or a technical report is needed to fully detail it. Enough information to make the experiment reproducible must be included with the textual presentation [RGPN+11], [CG12], [SLP14]. Consequently, reproducibility of computational results have become a requirement for submission to many high-ranked journals [Edi11], [LMS12].

But how does one make computational experiments reproducible? Several communities have proposed best practices, rules, and tools to help in making results reproducible, see e.g. [VKV09], [SNTH13], [SM14], [Dav12], [SLP14]. Still, this is an area of active research with methods and tools constantly evolving and maturing. Thus, the adoption of the reproducible research paradigm in most scientific communities is still ongoing - and will be for some time. However, a clear description of how the reproducible research paradigm fits in with customary workflows in a scientific community may help speed up the adoption of it. Furthermore, if tools that aid in making results reproducible for such customary workflows are made available, they may act as an additional catalyst.

In the present study, we focus on giving guidelines for integrating the reproducible research paradigm in the typical scientific Python workflow. In particular, we propose an easy to use scheme for storing metadata along with results in an HDF5 database. We show that it is possible to use Python to adhere to best practices for making computational experiments reproducible by storing metadata as JSON serialized arrays along with the results in an HDF5 database. A reference implementation of our proposed solution is part of the open source Magni Python package.

The remainder of this paper is organized as follows. We first describe our focus and its relation to a more general data management problem. We then outline the desired workflow for making scientific Python experiments reproducible and briefly review the fitness of existing reproducibility aiding tools for this workflow. This is continued by a description of our proposed scheme for storing metadata along with results. Following this specification, we detail a reference implementation of it and give plenty examples of its use. The paper ends with a more general discussion of related reproducibility aiding software packages followed by our conclusions.

The Data Management Problem

Reproducibility of computational results may be considered a part of a more general problem of data management in a computational study. In particular, it is closely related to the data management tasks of documenting and describing data. A typical computational study involves testing several combinations of various elements, e.g. input data, hardware platforms, external software libraries,

* Corresponding author: cso@es.aau.dk
‡ Faculty of Engineering and Science, Department of Electronic Systems, Aalborg University, 9220 Aalborg, Denmark

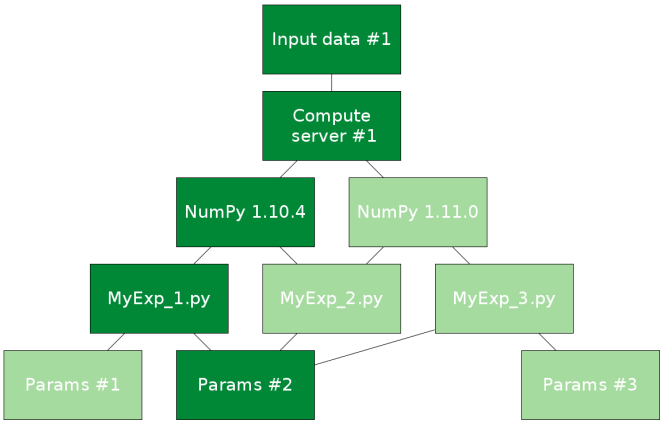

Fig. 1: *Illustration of a typical data management description problem as a layered graph. In this exemplified experiment, several combinations of input data, hardware platforms, software libraries (e.g. NumPy), algorithmic/experimental setup (described in a Python script), and parameter values are tested. The challenging task is to keep track of both the full set of combinations tested (marked by all the edges in the graph) as well as the individual simulations (e.g. the combination of highlighted vertices).*

experiment specific code, and model parameter values. Such a study may be illustrated as a layered graph like the one shown in figure 1. Each layer corresponds to one of the elements, e.g. the version of the NumPy library or the set of parameter values. The edges in the graph mark all the combinations that are tested. An example of a combination that constitutes a single simulation or experiment is the set of connected vertices that are highlighted in the graph in figure 1. In the present study, we focus on the problem of documenting and describing such a single simulation. A closely related problem is that of keeping track of all tested combinations, i.e. the set of all paths through all layers in the graph in figure 1. This is definitely also an interesting and important problem. However, once the "single simulation" problem is solved, it should be straight forward to solve the "all combinations" problem by appropriately combining the information from all the single simulations.

Storing Metadata Along With Results

For our treatment of reproducibility of computational results, we adopt the meaning of reproducibility from [LMS12], [Hin14]. That is, *reproducibility* of a study is the ability of others to repeat the study and obtain the same results using a general description of the original work. The related term *replicability* then means the ability of others to repeat the study and obtain the same results using the exact same setup (code, hardware, etc.) as in the original work[1]. As pointed out in [Hin14], reproducibility generally requires replicability.

The lack of reproducibility of computational results is oftentimes attributed to missing information about critical computational details such as library versions, parameter values, or precise descriptions of the exact code that was run [LMS12], [BPG05], [RGPN+11], [Mer10]. Several studies have given best practices for how to detail such metadata to make computational results reproducible, see e.g. [VKV09], [SNTH13], [SM14], [Dav12]. Here we detail the desired workflow for storing such metadata along with results when using a typical scientific Python workflow in the computational experiments. That is, we detail how to document a single experiment as illustrated by the highlighted vertices in figure 1.

The Scientific Python Workflow

In a typical scientific Python workflow, we define an experiment in a Python script and run that script using the Python interpreter, e.g.

```
$ python my_experiment.py
```

The content of the `my_experiment.py` script would typically have a structure like:

```
import some_library
import some_other_library

def some_func(...):
    ...

def run_my_experiment(...):
    ...

if __name__ == '__main__':
    run_my_experiment(...)
```

This is a particularly generic setup that only requires the availability of the Python interpreter and the libraries imported in the script. We argue that for the best practices for detailing a computational study to see broad adoption by the scientific Python community, three elements are of critical importance: Any method or tool for storing the necessary metadata to make the results reproducible must

1) be very easy to use and integrate well with existing scientific Python workflows.
2) be of high quality to be as trustworthy as the other tools in the scientific Python stack.
3) store the metadata in an open format that is easily inspected using standard viewers as well as programmatically from Python.

These elements are some of the essentials that have made Python so popular in the scientific community[2]. Thus, for storing the necessary metadata, we seek a high quality solution which integrates well with the above exemplified workflow. Furthermore, the metadata must be stored in such a way that is is easy to extract and inspect when needed.

Existing Tools

Several tools for keeping track of provenance and aiding in adhering to best practices for reproducible research already exist, e.g. Sumatra [Dav12], ActivePapers [Hin15], or Madagascar [Fom15]. Tools like Sumatra, ActivePapers, and Madagascar generally function as *reproducibility frameworks*. That is, when used with Python, they wrap the standard Python interpreter with a framework that in addition to running a Python script (using the standard Python interpreter) also captures and stores metadata detailing the setup used to run the experiment. E.g. when using Sumatra, one would replace `python my_experiment.py` with [Dav12]

```
$ smt run -e python -m my_experiment.py
```

1. Some authors (e.g. [SLP14]) swap the meaning of *reproducibility* and *replicability* compared to the convention, we have adopted.

2. See http://cyrille.rossant.net/why-using-python-for-scientific-computing/ for an overview of the main arguments for using Python for scientific computing.

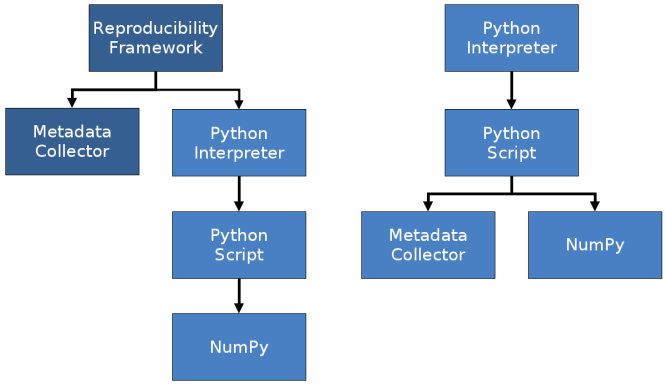

Fig. 2: *Illustration of the difference between a full reproducibility framework (on the left) and an importable Python library (on the right). The reproducibility framework calls the metadata collector as well as the Python interpreter which in turn runs the Python simulation script which e.g. imports NumPy. When using an importable library, the metadata collector is imported in the Python script alongside with e.g. NumPy.*

This idea of wrapping a computational simulation is different from the usual scientific Python workflow which consists of running a Python script that imports other packages and modules as needed, e.g. importing NumPy for numerical computations. This difference is illustrated in figure 2.

We argue that an importable Python library for aiding in making results reproducible has several advantages compared to using a full blown reproducibility framework. A major element in using any tool for computational experiments is being able to trust that the tool does what it is expected do. The scientific community trusts Python and the SciPy stack. For a reproducibility framework to be adopted by the community, it must build trust as the wrapper of the Python interpreter, it effectively is. That is, one must trust that it handles experiment details such as input parameters, library paths, etc. just as accurately as the Python interpreter would have done. Furthermore, such a framework must be able to fully replace the Python interpreter in all existing workflows which uses the Python interpreter. A traditional imported Python library does not have these potentially staggering challenges to overcome in order to see wide adoption. It must only build trust among its users in the same way as any other scientific library. Furthermore, it would be easy to incorporate into any existing workflow. Thus, ideally we seek a solution that allow us to update our `my_experiment.py` to have a structure like:

```
import some_library
import some_other_library
import reproducibility_library

def some_func(...):
    ...

def run_my_experiment(...):
    ...

if __name__ == '__main__':
    reproducibility_library.store_metadata(...)
    run_my_experiment(...)
```

Interestingly, the authors of the Sumatra package has to some degree pursued this idea by offering an API for importing the library as an alternative to using the `smt run` command line tool.

Equally important, to how to obtain the results, is how to inspect the results afterwards. Thus, one may ask: *How are the results and the metadata stored, and how may they be accessed later on?* For example, Sumatra by default stores all metadata in a SQLite database [Dav12] separate from simulation results (which may be stored in any format) whereas ActivePapers stores the metadata along with the results in an HDF5 database [Hin15]. The idea of storing (or "caching") intermediate results and metadata along with the final results has also been pursued in another study [PE09].

We argue that this idea of storing metadata along with results is an excellent solution. Having everything compiled into one standardized and open file format helps keep track of all the individual elements and makes it easy to share the full computational experiment including results and metadata. Preferably, such a file format should be easy to inspect using a standard viewer on any platform; just like the Portable Document Format (PDF) has made it easy to share and inspect textual works across platforms. The HDF5 Hierarchical Data Format [FP10] is a great candidate for such a file format due to the availability of cross-platform viewers like HDFView[3] and HDFCompass[4] as well as its capabilities in terms of storing large datasets. Furthermore, HDF5 is recognized in the scientific Python community[5] with bindings available through e.g. PyTables[6], h5py[7], or Pandas [McK10]. Also, bindings for HDF5 exists in several other major programming languages.

Suggested Library Design

Our above analysis reveals that all elements needed for implementing the reproducible research paradigm in scientific Python are in fact already available in existing reproducibility aiding tools: Sumatra may serve as a Python importable library and the ActivePapers project shows how metadata may be stored along with results in an HDF5 database. However, no single tool offers all of these elements for the scientific Python workflow. Consequently, we propose creating a scientific Python package that may be imported in existing scientific Python scripts and may be used to store all relevant metadata for a computational experiment along with the results of that experiment in an HDF5 database.

Technically, there are various ways to store metadata along with results in an HDF5 database. The probably most obvious way is to store the metadata as attributes to HDF5 tables and arrays containing the results. However, this approach is only recommended for small metadata (generally < 64KB)[8]. For larger metadata it is recommended to use a separate HDF5 array or table for storing the metadata[9]. Thus, for the highest flexibility, we propose to store the metadata as separate HDF5 arrays. This also allows for separation of specific result arrays or tables and general metadata. When using separate metadata arrays, a serialization (a representation) of the metadata must be chosen. For the metadata to be humanly readable using common HDF viewers, it must be stored in an easily readable string representation. We suggest using JSON [ECM13] for serializing the metadata. This makes for a humanly readable representation. Furthermore, JSON is a standard format with bindings for most major programming languages[10].

3. See https://www.hdfgroup.org/products/java/hdfview/
4. See https://github.com/HDFGroup/hdf-compass
5. See https://www.youtube.com/watch?v=nddj5OA8LJo
6. See http://www.pytables.org/
7. See http://www.h5py.org/

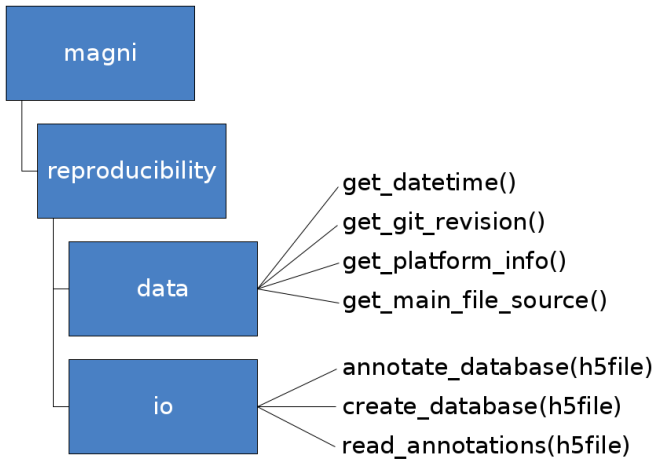

Fig. 3: *Illustration of the structure of the* `magni.reproducibility` *subpackage of Magni. The main modules are the* `data` *module for acquiring metadata and the* `io` *module for interfacing with an HDF5 database when storing as well as reading the metadata. A subset of available functions are listed next to the modules.*

In particular, Python bindings are part of the standard library (introduced in Python 2.6)[11]. This would effectively make Python >=2.6 and an HDF5 Python interface the only dependencies of our proposed reproducibility aiding library. We note, though, that the choice of JSON is not crucial. Other formats similar to JSON (e.g. XML[12] or YAML[13]) may be used as well. We do argue, though, that a humanly readable format should be used such that the metadata may be inspected using any standard HDF5 viewer.

Magni Reference Implementation

A reference implementation of the above suggested library design is available in the open source Magni Python package [OPA+14]. In particular, the subpackage `magni.reproducibility` is based on this suggested design. Figure 3 gives an overview of the `magni.reproducibility` subpackage. Additional resources for `magni` are:

- Official releases: doi:10.5278/VBN/MISC/Magni
- Online documentation: http://magni.readthedocs.io
- GitHub repository: https://github.com/SIP-AAU/Magni

In `magni.reproducibility`, a differentiation is made between *annotations* and *chases*. *Annotations* are metadata that describe the setup used for the computation, e.g. the computational environment, values of input parameters, platform (hardware/OS) details, and when the computation was done. *Chases* on the other hand are metadata describing the specific code that was used in the computation and how it was called, i.e. they *chase* the provenance of the results.

8. See http://docs.h5py.org/en/latest/high/attr.html
9. See https://www.hdfgroup.org/HDF5/doc1.6/UG/13_Attributes.html
10. See http://www.json.org/
11. See https://docs.python.org/2/library/json.html
12. See https://www.w3.org/TR/REC-xml/
13. See http://yaml.org/

Requirements

Magni uses PyTables as its interface to HDF5 databases. Thus, had `magni.reproducibility` been a package of its own, only Python and PyTables would have been requirements for its use. The full requirements for using `magni` (as of version 1.5.0) are[14]

- Python >= 2.7 / 3.3
- Matplotlib [Hun07] (Tested on version >= 1.3)
- NumPy [vdWCV11] (Tested on version >= 1.8)
- PyTables[15] (Tested on version >= 3.1)
- SciPy [Oli07] (Tested on version >= 0.14)
- Setuptools[16] (Tested on version >= 11.3)

When using the Conda[17] package management system for handling the Python environment used in the computation, `magni.reproducibility` may optionally use Conda to capture details about the Python environment. Thus, we have one optional dependency

- Conda (Tested on version >= 3.7.0)

Usage Examples

We now give several smaller examples of how to use `magni.reproducibility` to implement the best practices for reproducibility of computational result described in [VKV09], [SNTH13], [SM14]. An extensive example of the usage of `magni.reproducibility` is available at doi:10.5278/VBN/MISC/MagniRE. This extensive example is based on a Python script used to simulate the Mandelbrot set[18] using the scientific Python workflow described above. An example of a resulting HDF5 database containing both the Mandelbrot simulation result and metadata is also included. Finally, the example includes a Jupyter Notebook showing how to read the metadata using `magni.reproducibility`.

A simple example of how to acquire platform metadata using the `data` module from `magni.reproducibility` is

```
>>> from pprint import pprint
>>> from magni import reproducibility as rep
>>> pprint(rep.data.get_platform_info())
{'libc': '["glibc", "2.2.5"]',
 'linux': '["debian", "jessie/sid", ""]',
 'mac_os': '["", ["", "", ""], ""]',
 'machine': '"x86_64"',
 'node': '"eagle1"',
 'processor': '"x86_64"',
 'python': '"3.5.1"',
 'release': '"3.16.0-46-generic"',
 'status': 'All OK',
 'system': '"Linux"',
 'version': '"#62~14.04.1-Ubuntu SMP ~"',
 'win32': '["", "", "", ""]'}
```

When using the typical scientific Python workflow described above, one may use the functions in the `io` module from `magni.reproducibility` to conveniently store all relevant metadata, e.g. the `create_database(h5file)` to automatically create an HDF5 database with a set of standard annotations

14. More details about Python and the Scientific Python Stack are available at http://python.org and http://scipy.org
15. See http://www.pytables.org/
16. See http://setuptools.readthedocs.io/
17. See http://conda.pydata.org/docs/ as well as https://www.youtube.com/watch?v=UaIvrDWrIWM
18. See https://en.wikipedia.org/wiki/Mandelbrot_set

and chases. The `my_experiment.py` script would then have a structure like

```
import tables
from magni import reproducibility as rep

def run_my_experiment(...):
    ...

def store_result(h5, result):
    ...

if __name__ == '__main__':
    hdf5_db = 'database.hdf5'
    rep.io.create_database(hdf5_db)
    result = run_my_experiment(...)
    with tables.File(hdf5_db, mode='a') as h5:
        store_result(h5, result)
```

This would create an HDF5 database named `database.hdf5` which would hold both the results and all metadata. The HDF5 database may be inspected using any tool capable of reading HDF5 files. As an alternative, the `io` module from `magni.reproducibility` also includes convenience functions for reading the annotations and chases. E.g. to see the set of standard metadata stored in a database with `create_database(h5file)`, one could do

```
>>> from pprint import pprint
>>> import tables
>>> from magni import reproducibility as rep
>>> hdf5_db = 'database.hdf5'
>>> rep.io.create_database(hdf5_db)
>>> with tables.File(hdf5_db) as h5:
...     annotations = rep.io.read_annotations(h5)
...     chases = rep.io.read_chases(h5)
>>> pprint(list(annotations.keys()))
['magni_config',
 'git_revision',
 'datetime',
 'conda_info',
 'magni_info',
 'platform_info']
>>> pprint(list(chases.keys()))
['main_file_source',
 'stack_trace',
 'main_file_name',
 'main_source']
```

Quality Assurance

The Magni Python package is fully documented and comes with an extensive test suite. It has been developed using best practices for developing scientific software [WAB+14] and all code has been reviewed by at least one other person than its author prior to its inclusion in Magni. All code adheres to the PEP8[19] style guide and no function or class has a cyclomatic complexity [McC76], [WM96] exceeding 10. The source code is under version control using Git and a continuous integration system based on Travis CI[20] is in use for the git repository. More details about the quality assurance of `magni` are given in [OPA+14].

Related Software Packages

Independently of the tool or method used, making results from scientific computations reproducible is not only for the benefit of the audience. As pointed out in several studies [Fom15], [CG12], [VKV09], the author of the results gains as least as much in terms increasing one's productivity. Thus, using some method or tool to help make the results reproducible is a win for everyone. In the present work we have attempted to detail the ideal solution for how to do this for the typical scientific Python workflow.

A plethora of related alternative tools exist for aiding in making results reproducible. We have already discussed ActivePapers [Hin15], Sumatra [Dav12], and Madagascar [Fom15] which are general reproducibility frameworks that allow for wrapping most tools - not only Python based computations. Such tools are definitely excellent for some workflows. In particular, they seem fit for large fixed setups which require keeping track of several hundred runs that only differ by the selection of parameters[21] and for which the time cost of initially setting up the tool is insignificant compared to the time cost of the entire study. That is, they are useful in keeping track of the full set of combination in a large computations study as marked by all the edges in the layered graph in figure 1. However, as we have argued, they are less suitable for documenting a single experiment based on the typical scientific Python workflow. Also these tools tend to be designed for use on a single computer. Thus, they do not scale well for big data applications which run on compute clusters.

Another category of related tools are graphical user interface (GUI) based workflow managing tools like Taverna [OAF+04] or Vistrail [SFC07]. Such tools seem to be specifically designed for describing computational workflows in particular fields of research (typically bioinformatics related fields). It is hard, though, to see how they can be effectively integrated with the typical scientific Python workflow. Other much more Python oriented tools are the Jupyter Notebook[22] as well as Dexy[23]. These tools, however, seem to have more of a focus on implementing the concept of literate programming and documentation than reproducibility of results in general.

Conclusions

We have argued that metadata should be stored along with computational results in an easily readable format in order to make the results reproducible. When implementing this in a typical scientific Python workflow, all necessary tools for making the results reproducible should be available as an importable package. We suggest storing the metadata as JSON serialized arrays along with the result in an HDF5 database. A reference implementation of this design is available in the open source Magni Python package which we have detailed with several examples of its use. All of this shows that storing metadata along with results is important in implementing reproducible research and it is readily achievable using scientific Python packages.

Acknowledgements

This work was supported in part by the Danish Council for Independent Research (DFF/FTP) under Project 1335-00278B/12-134971 and in part by the Danish e-Infrastructure Cooperation (DeIC) under Project DeIC2013.12.23.

References

[BE12] C. Glenn Begley and Lee M. Ellis. Drug development: Raise standards for preclinical cancer research. *Nature*, 483(7391):531–533, March 2012. doi:10.1038/483531a.

19. See https://www.python.org/dev/peps/pep-0008/
20. See https://travis-ci.org/
21. See e.g. https://www.youtube.com/watch?v=1YJr9c-zSng
22. See http://jupyter.org/
23. See http://www.dexy.it/ as well as https://www.youtube.com/watch?v=u6_qtDJ6ciA / https://www.youtube.com/watch?v=qFd04rA8lp0

[BPG05] Mauro Barni and Fernando Perez-Gonzalez. Pushing Science into Signal Processing. *IEEE Signal Processing Magazine*, 22(4):120–119, July 2005. doi:10.1109/MSP.2005.1458324.

[CG12] Kingshuk Roy Choudhury and Ray Gibson. Editorial: Reproducible Research in Medical Imaging. *Molecular Imaging and Biology*, 14(4):395–396, June 2012. doi:10.1007/s11307-012-0569-8.

[Dav12] Andrew P. Davison. Automated Capture of Experiment Context for Easier Reproducibility in Computational Research. *Computing in Science & Engineering*, 14(4):48–56, July 2012. doi:10.1109/MCSE.2012.41.

[ECM13] The JSON Data Interchange Format, October 2013.

[Edi11] Editorial. Devil in the details. *Nature*, 470(7334):305–306, February 2011. doi:10.1038/470305b.

[Fom15] Sergey Fomel. Reproducible Research as a Community Effort: Lessons from the Madagascar Project. *Computing in Science & Engineering*, 17(1):20–26, January 2015. doi:10.1109/MCSE.2014.94.

[FP10] Mike Folk and Elena Pourmal. Balancing Performance and Preservation Lessons learned with HDF5. In *Digital Preservation Interoperability Framework (DPIF) Workshop*, Gaithersburg, Maryland, USA, March 29 – 31, 2010. doi:10.1145/2039274.2039285.

[Hin14] Konrad Hinsen. Computational science: shifting the focus from tools to models. *F1000Research*, 3(101):1–16, June 2014. doi:10.12688/f1000research.3978.2.

[Hin15] Konrad Hinsen. ActivePapers: a platform for publishing and archiving computer-aided research. *F1000Research*, 3(289):14, July 2015. doi:10.12688/f1000research.5773.3.

[Hun07] John D. Hunter. Matplotlib: A 2D Graphics Environment. *Computing in Science & Engineering*, 9(3):90–95, May 2007. doi:10.1109/MCSE.2007.55.

[LMS12] Randall J. LeVeque, Ian M. Mitchell, and Victoria Stodden. Reproducible Research for Scientific Computing: Tools and Strategies for Changing the Culture. *Computing in Science & Engineering*, 14(4):13–17, July 2012. doi:10.1109/MCSE.2012.38.

[McC76] Thomas J. McCabe. A Complexity Measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, December 1976. doi:10.1109/TSE.1976.233837.

[McK10] Wes McKinney. Data Structures for Statistical Computing in Python. In *Proceedings of the 9th Python in Science Conference*, pages 51–56, Austin, Texas, USA, June 28 – July 3, 2010.

[Mer10] Zeeya Merali. Computational science: ...Error ...why scientific programming does not compute. *Nature*, 467:775–777, October 2010. doi:10.1038/467775a.

[Mil06] Greg Miller. A Scientist's Nightmare: Software Problem Leads to Five Retractions. *Science*, 314(5807):1856–1857, December 2006. doi:10.1126/science.314.5807.1856.

[OAF+04] Tom Oinn, Matthew Addis, Justin Ferris, Darren Marvin, Martin Senger, Mark Greenwood, Tim Carver, Kevin Glover, Matthew R. Pocock, Anil Wipat, and Peter Li. Taverna: a tool for the composition and enactment of bioinformatics workflows. *Bioinformatics*, 20(17):3045–3054, June 2004. doi:10.1093/bioinformatics/bth361.

[Oli07] Travis E. Oliphant. Python for Scientific Computing. *Computing in Science & Engineering*, 9(3):10–20, May 2007. doi:10.1109/MCSE.2007.58.

[OPA+14] Christian Schou Oxvig, Patrick Steffen Pedersen, Thomas Arildsen, Jan Østergaard, and Torben Larsen. Magni: A Python Package for Compressive Sampling and Reconstruction of Atomic Force Microscopy Images. *Journal of Open Research Software*, 2(1):e29, October 2014. doi:10.5334/jors.bk.

[PE09] Roger D. Peng and Sandrah P. Eckel. Distributed Reproducible Research Using Cached Computations. *Computing in Science & Engineering*, 11(1):28–34, January 2009. doi:10.1109/MCSE.2009.6.

[Pen11] Roger D. Peng. Reproducible Research in Computational Science. *Science*, 334(6060):1226–1227, December 2011. doi:10.1126/science.1213847.

[RGPN+11] Markus Rupp, Fulvio Gini, Ana Pérez-Neira, Beatrice Pesquet-Popescu, Aggelos Pikrakis, Bulent Sankur, Patrick Vandewalle, and Abdelhak Zoubir. Reproducible research in signal processing. *EURASIP Journal on Advances in Signal Processing*, 93(1):1–2, October 2011. doi:10.1186/1687-6180-2011-93.

[SFC07] Claudio T. Silva, Juliana Freire, and Steven P. Callahan. Provenance for Visualizations: Reproducibility and Beyond. *Computing in Science & Engineering*, 9(5):82–89, September 2007. doi:10.1109/MCSE.2007.106.

[SLP14] Victoria Stodden, Friedrich Leisch, and Roger D. Peng, editors. *Implementing Reproducible Research*. Chapman & Hall/CRC The R Series. CRC Press, 2014.

[SM14] Victoria Stodden and Sheila Miguez. Best Practices for Computational Science: Software Infrastructure and Environments for Reproducible and Extensible Research. *Journal of Open Research Software*, 2(1):1–6, July 2014. doi:10.5334/jors.ay.

[SNTH13] Geir Kjetil Sandve, Anton Nekrutenko, James Taylor, and Eivind Hovig. Ten Simple Rules for Reproducible Computational Research. *PLoS Computational Biology*, 9(10):e1003285, October 2013. doi:10.1371/journal.pcbi.1003285.

[vdWCV11] Stéfan van der Walt, S. Chris Colbert, and Gaël Varoquaux. The NumPy Array: A Structure for Efficient Numerical Computation. *Computing in Science & Engineering*, 13(2):22–30, March 2011. doi:10.1109/MCSE.2011.37.

[VKV09] P. Vandewalle, J. Kovačević, and M. Vetterli. Reproducible Research in Signal Processing [What, why, and how]. *IEEE Signal Processing Magazine*, 26(3):37–47, May 2009. doi:10.1109/MSP.2009.932122.

[WAB+14] Greg Wilson, D. A. Aruliah, C. Titus Brown, Neil P. Chue Hong, Matt Davis, Richard T. Guy, Steven H. D. Haddock, Kathryn D. Huff, Ian M. Mitchell, Mark D. Plumbley, Ben Waugh, Ethan P. White, and Paul Wilson. Best Practices for Scientific Computing. *PloS Biology*, 12(1):e1001745, January 2014. doi:10.1371/journal.pbio.1001745.

[WM96] Arthur H. Watson and Thomas J. McCabe. Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric. Special Publication 500-235, National Institute of Standards and Technology (NIST), September 1996.

datreant: persistent, Pythonic trees for heterogeneous data

David L. Dotson[‡†*], Sean L. Seyler[‡], Max Linke[§], Richard J. Gowers[¶‖], Oliver Beckstein[‡]

https://youtu.be/enLHDZoch0U

Abstract—In science the filesystem often serves as a *de facto* database, with directory trees being the zeroth-order scientific data structure. But it can be tedious and error prone to work directly with the filesystem to retrieve and store heterogeneous datasets. **datreant** makes working with directory structures and files Pythonic with **Treants**: specially marked directories with distinguishing characteristics that can be discovered, queried, and filtered. Treants can be manipulated individually and in aggregate, with mechanisms for granular access to the directories and files in their trees. Disparate datasets stored in any format (CSV, HDF5, NetCDF, Feather, etc.) scattered throughout a filesystem can thus be manipulated as meta-datasets of Treants. **datreant** is modular and extensible by design to allow specialized applications to be built on top of it, with MDSynthesis as an example for working with molecular dynamics simulation data. http://datreant.org/

Index Terms—data management, science, filesystems

Introduction

In many scientific fields, especially those analyzing experimental or simulation data, there is an existing ecosystem of specialized tools and file formats which new tools must work around. Consequently, specialized database systems may be unsuitable for data management and storage. In these cases the filesystem ends up serving as a *de facto* database, with directory trees the zeroth-order data structure for scientific data. This is particularly true for fields centered around simulation: simulation systems can vary widely in size, composition, rules, parameters, and starting conditions. And with ever-increasing computational power, it is often necessary to store intermediate results from large amounts of simulation data so that they may be accessed and explored interactively.

These problems make data management difficult, and ultimately serve as a barrier to answering scientific questions. To address this, we present `datreant`, a Pythonic interface to the filesystem. `datreant` deals primarily in **Treants**: specially marked directories with distinguishing characteristics that can be discovered, queried, and filtered. Treants can be manipulated individually and in aggregate, with mechanisms for granular access to the directories and files in their trees. By way of Treants, `datreant` adds a lightweight abstraction layer to the filesystem, allowing researchers to focus more on *what* is stored and less on *where*. This greatly reduces the tedium of storing, retrieving, and operating on datasets of interest, no matter how they are organized.

† *These authors contributed equally.*
∗ *Corresponding author: dldotson@asu.edu*
‡ *Arizona State University, Tempe, Arizona, USA*
§ *Max Planck Institut für Biophysik, Frankfurt, Germany*
¶ *University of Manchester, Manchester, UK*
‖ *University of Edinburgh, Edinburgh, UK*

Treants as filesystem manipulators

The central object of `datreant` is the `Treant`. A Treant is a directory in the filesystem that has been specially marked with a **state file**. A `Treant` is also a Python object. We can create a `Treant` with:

```
>>> import datreant.core as dtr
>>> t = dtr.Treant('maple')
>>> t
<Treant: 'maple'>
```

This creates a directory `maple/` in the filesystem (if it did not already exist), and places a special state file inside which stores the Treant's state. This file also serves as a flagpost indicating that this is more than just a directory:

```
> ls maple
Treant.1dcbb3b1-c396-4bc6-975d-3ae1e4c2983a.json
```

The name of this file includes the type of Treant to which it corresponds, as well as the `uuid` of the Treant, its unique identifier. The state file contains all the information needed to generate an identical instance of this `Treant`, so that we can start a separate Python session and immediately use the same Treant there:

```
# python session 2
>>> import datreant.core as dtr
>>> t = dtr.Treant('maple')
>>> t
<Treant: 'maple'>
```

Making a modification to the `Treant` in one session is immediately reflected by the same `Treant` in any other session. For example, a `Treant` can store any number of descriptive tags to differentiate it from others. We can add tags in the first Python session:

```
# python session 1
>>> t.tags.add('syrup', 'plant')
>>> t.tags
<Tags(['plant', 'syrup'])>
```

And in the other Python session, the same `Treant` with the same tags is visible:

```
# python session 2
>>> t.tags
<Tags(['plant', 'syrup'])>
```

Internally, advisory locking is done to avoid race conditions, making a `Treant` multiprocessing-safe. A `Treant` can also be moved, either locally within the same filesystem or to a remote filesystem, and it will continue to work as expected.

Introspecting a Treant's Tree

A `Treant` can be used to introspect and manipulate its filesystem tree. We can, for example, work with directory structures rather easily:

```
>>> data = t['a/place/for/data/']
>>> data
<Tree: 'maple/a/place/for/data/'>
```

This `Tree` object points to a path in the Treant's own tree, but it need not necessarily exist. We can check this with:

```
>>> data.exists
False
```

This behavior is by design for `Tree` objects (as well as `Leaf` objects; see below). We want to be able to work freely with paths without creating filesystem objects for each, at least until we are ready.

We can make a `Tree` exist in the filesystem easily enough:

```
>>> data.makedirs()
```

and if we also make another directory, too:

```
>>> t['a/place/for/text/'].makedirs()
<Tree: 'maple/a/place/for/text/'>
```

we now have:

```
>>> t.draw()
maple/
 +-- Treant.1dcbb3b1-c396-4bc6-975d-3ae1e4c2983a.json
 +-- a/
     +-- place/
         +-- for/
             +-- data/
             +-- text/
```

Accessing paths in this way returns `Tree` and `Leaf` objects, which refer to directories and files, respectively. These paths need not point to directories or files that actually exist, but they can be used to create and work with these filesystem elements. It should be noted that creating a `Tree` does *not* create a `Treant`. Treants are considered special enough to warrant having a state file with metadata, and making every directory a Treant would make them less useful.

We can, for example, easily store a Pandas [McK10] DataFrame somewhere in the tree for reference later:

```
>>> import pandas as pd
>>> df = pd.DataFrame(pd.np.random.randn(3, 2),
                      columns=['A', 'B'])
>>> data = t['a/place/for/data/']
>>> data
<Tree: 'maple/a/place/for/data/'>
>>> df.to_csv(data['random_dataframe.csv'].abspath)

# take a look at the contents of `data`
>>> data.draw()
data/
+-- random_dataframe.csv
```

and we can introspect the file directly:

```
>>> csv = data['random_dataframe.csv']
>>> csv
<Leaf: 'maple/a/place/for/data/random_dataframe.csv'>

# this should look like a CSV file
>>> print(csv.read())
,A,B
0,-0.573730932177663,-0.08857033924376226
1,0.03157276797041359,-0.10977921690694506
2,-0.2080757315892524,0.6825003213837373
```

Using `Treant`, `Tree`, and `Leaf` objects, we can work with the filesystem Pythonically without giving much attention to precisely *where* these objects live within that filesystem. This becomes especially powerful when we have many directories/files we want to work with, possibly in many different places.

Aggregation and splitting on Treant metadata

What makes a `Treant` distinct from a `Tree` is its **state file**. This file stores metadata that can be used to filter and split `Treant` objects when treated in aggregate. It also serves as a flagpost, making Treant directories discoverable.

If we have many more Treants, perhaps scattered about the filesystem:

```
>>> for path in ('an/elm/', 'the/oldest/oak',
...              'the/oldest/tallest/sequoia'):
...
...     # make a Treant in filesystem at path
...     dtr.Treant(path)
```

we can gather them up with `datreant.core.discover`:

```
>>> b = dtr.discover('.')
>>> b
<Bundle([<Treant: 'oak'>, <Treant: 'sequoia'>,
         <Treant: 'maple'>, <Treant: 'elm'>])>
```

A `Bundle` is an ordered set of `Treant` objects. This collection gives convenient mechanisms for working with Treants as a single logical unit. For example, it exposes a few basic properties for directly accessing its member data:

```
>>> b.relpaths
['the/oldest/oak/',
 'the/oldest/tallest/sequoia/',
 'maple/',
 'an/elm/']

>>> b.names
['oak', 'sequoia', 'maple', 'elm']
```

A `Bundle` can be constructed in a variety of ways, most commonly using existing `Treant` instances or paths to Treants in the filesystem.

We can use a `Bundle` to subselect Treants in typical ways, including integer indexing and slicing, fancy indexing, boolean indexing, and indexing by name. But in addition to these, we can use metadata features such as **tags** and **categories** to filter and group Treants as desired.

Filtering Treants with tags

Tags are individual strings that describe a Treant. Setting the tags for each of our Treants separately:

```
>>> b['maple'].tags = ['syrup', 'furniture', 'plant']
>>> b['sequoia'].tags = ['huge', 'plant']
```

```
>>> b['oak'].tags = ['for building', 'plant', 'building']
>>> b['elm'].tags = ['firewood', 'shady', 'paper',
                     'plant', 'building']
```

we can now work with these tags in aggregate:

```
# will only show tags present in *all* members
>>> b.tags
<AggTags(['plant'])>

# will show tags present among *any* member
>>> b.tags.any
{'building',
 'firewood',
 'for building',
 'furniture',
 'huge',
 'paper',
 'plant',
 'shady',
 'syrup'}
```

and we can filter on them. For example, getting all Treants that are good for construction work:

```
# gives a boolean index for members with this tag
>>> b.tags['building']
[True, False, False, True]

# we can use this to index the Bundle itself
>>> b[b.tags['building']]
<Bundle([<Treant: 'oak'>, <Treant: 'elm'>])>
```

or getting back Treants that are both good for construction *and* used for making furniture by giving tags as a list:

```
# a list of tags serves as an *intersection* query
>>> b[b.tags[['building', 'furniture']]]
<Bundle([])>
```

which in this case none of them are.

Other tag expressions can be constructed using tuples (for *or/union* operations) and sets (for a *negated intersection*), and nesting of any of these works as expected:

```
# we can get a *union* by using a tuple
>>> b[b.tags['building', 'furniture']]
<Bundle([<Treant: 'maple'>, <Treant: 'oak'>,
         <Treant: 'elm'>])>

# we can get a *negated intersection* by using a set
>>> b[b.tags[{'building', 'furniture'}]]
<Bundle([<Treant: 'sequoia'>, <Treant: 'maple'>,
         <Treant: 'oak'>, <Treant: 'elm'>])>
```

Using tag expressions, we can filter to Treants of interest from a `Bundle` counting many, perhaps hundreds, of Treants as members. A common workflow is to use `datreant.core.discover` to gather up many Treants from a section of the filesystem, then use tags to extract only those Treants one actually needs.

Splitting Treants on categories

Categories are key-value pairs that provide another mechanism for distinguishing Treants. We can add categories to each Treant:

```
# add categories to individual members
>>> b['oak'].categories = {'age': 'adult',
                           'type': 'deciduous',
                           'bark': 'mossy'}
>>> b['elm'].categories = {'age': 'young',
                           'type': 'deciduous',
                           'bark': 'smooth'}
>>> b['maple'].categories = {'age': 'young',
                             'type': 'deciduous',
                             'bark': 'mossy'}
>>> b['sequoia'].categories = {'age': 'old',
                               'type': 'evergreen',
                               'bark': 'fibrous',
                               'home': 'california'}

# add value 'tree' to category 'plant'
# for all members
>>> b.categories.add({'plant': 'tree'})
```

and we can access categories for individual Treants:

```
>>> seq = b['sequoia'][0]
>>> seq.categories
<Categories({'home': 'california',
             'age': 'old',
             'type': 'evergreen',
             'bark': 'fibrous',
             'plant': 'tree'})>
```

The aggregated categories for all members in a `Bundle` are accessible via `Bundle.categories`, which gives a view of the categories with keys common to *every* member Treant:

```
>>> b.categories
<AggCategories({'age': ['adult', 'young',
                        'young', 'old'],
                'type': ['deciduous', 'deciduous',
                         'deciduous', 'evergreen'],
                'bark': ['mossy', 'smooth',
                         'mossy', 'fibrous'],
                'plant': ['tree', 'tree',
                          'tree', 'tree']})>
```

Each element of the list associated with a given key corresponds to the value for each member, in member order. Using `Bundle.categories` is equivalent to `Bundle.categories.all`; we can also access categories present among *any* member:

```
>>> b.categories.any
{'age': ['adult', 'young', 'young', 'old'],
 'bark': ['mossy', 'smooth', 'mossy', 'fibrous'],
 'home': [None, None, None, 'california'],
 'type': ['deciduous', 'deciduous',
          'deciduous', 'evergreen']}
```

Members that do not have a given key will have `None` as the corresponding value in the list. Accessing values for a list of keys:

```
>>> b.categories[['age', 'home']]
[['adult', 'young', 'young', 'old'],
 [None, None, None, 'california']]
```

or a set of keys:

```
>>> b.categories[{'age', 'home'}]
{'age': ['adult', 'young', 'young', 'old'],
 'home': [None, None, None, 'california']}
```

returns, respectively, a list or dictionary of lists of values, where the list for a given key is in member order. Perhaps the most powerful feature of categories is the `groupby` method, which, given a key, can be used to group specific members in a `Bundle` by their corresonding category values. If we want to group members by their `'bark'`, we can use `groupby` to obtain a dictionary of members for each value of `'bark'`:

```
>>> b.categories.groupby('bark')
{'fibrous': <Bundle([<Treant: 'sequoia'>])>,
 'mossy': <Bundle([<Treant: 'oak'>,
```

```
                    <Treant: 'maple'>])>,
 'smooth': <Bundle([<Treant: 'elm'>])>}
```

Say we would like to get members grouped by both their `'bark'` and `'home'`:

```
>>> b.categories.groupby({'bark', 'home'})
{('fibrous', 'california'):
    <Bundle([<Treant: 'sequoia'>])>}
```

We get only a single member for the pair of keys `('fibrous', 'california')` since `'sequoia'` is the only Treant having the `'home'` category. Categories are useful as labels to denote the types of data that a Treant may contain or how the data were obtained. By leveraging the `groupby` method, one can extract Treants by selected categories without having to explicitly access each member. This feature can be particularly powerful in cases where many Treants have been created and categorized to handle incoming data over an extended period of time; one can quickly gather any data needed without having to think about low-level details.

Treant modularity with attachable Limbs

`Treant` objects manipulate their tags and categories using `Tags` and `Categories` objects, respectively. These are examples of `Limb` objects: attachable components which serve to extend the capabilities of a `Treant`. While `Tags` and `Categories` are attached by default to all `Treant` objects, custom `Limb` subclasses can be defined for additional functionality.

`datreant` is a namespace package, with the dependency-light core components included in `datreant.core`. The dependencies of `datreant.core` include backports of standard library modules such as `pathlib` and `scandir`, as well as lightweight modules such as `fuzzywuzzy` and `asciitree`.

`datreant.core` remains lightweight because other packages in the `datreant` namespace can have any dependencies they require. One such package is `datreant.data`, which includes a set of convenience `Limb` objects for storing and retrieving Pandas and NumPy [vdW11] datasets in HDF5 using PyTables and h5py internally.

We can attach a `Data` limb to a `Treant` with:

```
>>> import datreant.data
>>> t = dtr.Treant('maple')
>>> t.attach('data')
>>> t.data
<Data([])>
```

and we can immediately start using it to store e.g. a Pandas `Series`:

```
>>> import numpy as np
>>> sn = pd.Series(np.sin(
...     np.linspace(0, 8*np.pi, num=200)))
>>> t.data['sinusoid'] = sn
```

and we can get it back just as easily:

```
>>> t.data['sinusoid'].head()
0    0.000000
1    0.125960
2    0.249913
3    0.369885
4    0.483966
dtype: float64
```

Looking at the directory structure of `"maple"`, we see that the data was stored in an HDF5 file under a directory corresponding to the name we stored it with:

```
>>> t.draw()
maple/
 +-- sinusoid/
 |   +-- pdData.h5
 +-- Treant.1dcbb3b1-c396-4bc6-975d-3ae1e4c2983a.json
```

What's more, `datreant.data` also includes a corresponding `AggLimb` for `Bundle` objects, allowing for automatic aggregation of datasets by name across all member `Treant` objects. If we collect and store similar datasets for each member in our `Bundle`:

```
>>> b = dtr.discover('.')
>>> b
<Bundle([<Treant: 'oak'>, <Treant: 'sequoia'>,
         <Treant: 'maple'>, <Treant: 'elm'>])>

# we want to make each dataset a bit different
>>> b.categories['frequency'] = [1, 2, 3, 4]
>>> for mem in b:
...     freq = mem.categories['frequency']
...     mem.data['sinusoid'] = pd.Series(np.sin(
...         freq * np.linspace(0, 8*np.pi, num=200)))
```

then we can retrieve all of them into a single, multi-index Pandas `Series`:

```
>>> sines = b.data.retrieve('sinusoid', by='name')
>>> sines.groupby(level=0).head()
sequoia  0    0.000000
         1    0.125960
         2    0.249913
         3    0.369885
         4    0.483966
oak      0    0.000000
         1    0.369885
         2    0.687304
         3    0.907232
         4    0.998474
maple    0    0.000000
         1    0.249913
         2    0.483966
         3    0.687304
         4    0.847024
elm      0    0.000000
         1    0.483966
         2    0.847024
         3    0.998474
         4    0.900479
dtype: float64
```

which we can use for aggregated analysis, or perhaps just pretty plots (Figure 1).

```
>>> for name, group in sines.groupby(level=0):
...     s = group.reset_index(level=0, drop=True)
...     s.plot(legend=True, label=name)
```

The `Data` limb stores Pandas and NumPy objects in the HDF5 format within a Treant's own tree. It can also store arbitrary (but pickleable) Python objects as pickles, making it a flexible interface for quick data storage and retrieval. However, it ultimately serves as an example for how `Treant` and `Bundle` objects can be extended to do complex but convenient things.

Using Treants as the basis for dataset access and manipulation with the PyData stack

Although it is possible to extend `datreant` objects with limbs to do complex operations on a Treant's tree, it isn't necessary

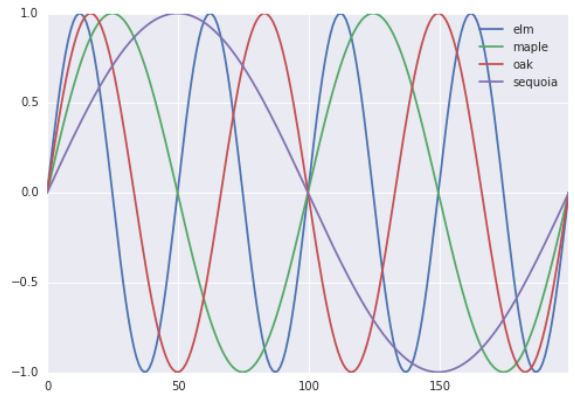

Fig. 1: *Plot of sinusoidal toy datasets aggregated and plotted by source Treant.*

to build specialized interfaces such as these to make use of the extensive PyData stack. `datreant` fundamentally serves as a Pythonic interface to the filesystem, bringing value to datasets and analysis results by making them easily accessible now and later. As data structures and file formats change, `datreant` objects can always be used in the same way to supplement the way these tools are used.

Because each Treant is both a Python object and a filesystem object, they work remarkably well with distributed computation libraries such as dask.distributed [Roc15] and workflow execution frameworks such as Fireworks [Jai15]. Treant metadata features such as tags and categories can be used for automated workflows, including backups and remote copies to external compute resources, making work on datasets less imperative and more declarative when desired.

Building domain-specific applications on datreant

Built-in `datreant.core` objects are general-purpose, while packages like `datreant.data` provide extensions to these objects that are more specific. But it is possible, and very useful, for domain-specific applications to define their own domain-specific `Treant` subclasses, with tightly-coupled limbs for domain-specific needs. Not only do objects such as `Bundle` work just fine with `Treant` subclasses and custom `Limb` classes; they are designed explicitly with this need in mind.

The first example of a domain-specific package built around `datreant` is MDSynthesis, a module that enables high-level management and exploration of molecular dynamics simulation data. MDSynthesis gives a Pythonic interface to molecular dynamics trajectories using MDAnalysis [MiA11], giving the ability to work with the data from many simulations scattered throughout the filesystem with ease. This package makes it possible to write analysis code that can work across many varieties of simulation, but even more importantly, MDSynthesis allows interactive work with the results from hundreds of simulations at once without much effort.

Leveraging molecular dynamics data with MDSynthesis

MDSynthesis defines a `Treant` subclass called a `Sim`. A `Sim` featues special limbs for storing an MDAnalysis `Universe` definition and custom atom selections within its state file, allowing for painless recall of raw simulation data and groups of atoms of interest.

As an example of effectively using `Sims`, say we have 50 biased molecular dynamics simulations that sample the conformational change of the ion transport protein NhaA [Lee14] from the inward-open to outward-open state (Figure 2). Let's also say that we are interested in how many hydrogen bonds exist at any given time between the two domains as they move past each other. These `Sim` objects already exist in the filesystem, each having a `Universe` definition already set to point to its unique trajectory file(s).

We can use the MDAnalysis `HydrogenBondAnalysis` class to collect the data for each `Sim` using `Bundle.map` for process parallelism, storing the results using the `datreant.data` limb:

```
import mdsynthesis as mds
import MDAnalysis.analysis.hbonds as hbonds
import pandas as pd
import seaborn as sns

b = mds.discover('NhaA_i2o_transitions')

def get_hbonds(sim):
    dimerization = sim.atomselections['dimer']
    core = sim.atomselections['core']

    hb = hbonds.HydrogenBondAnalysis(
            sim.universe, dimerization, core)
    hb.run()
    hb.generate_table()

    sim.data['hbonds'] = pd.DataFrame(hb.table)

# process parallelism provided internally
# with `multiprocessing`
b.map(get_hbonds, processes=16)
```

Then we can retrieve the datasets in aggregate using the `Bundle` `datreant.data` limb and visualize the result (Figure 3):

```
df = b.data.retrieve('hbonds', by='name')

counts = df['distance'].groupby(df.index).count()
counts.index = pd.MultiIndex.from_tuples(
                        counts.index)
counts.index = counts.index.droplevel(0)

sns.jointplot(counts.index, counts, kind='hexbin')
```

By making it relatively easy to work with what can often be many terabytes of simulation data spread over tens or hundreds of trajectories, MDSynthesis greatly reduces the time it takes to iterate on new ideas toward answering real biological questions.

Final thoughts

`datreant` is a young project that started as a domain-specific package for working with molecular dynamics data, but has quickly morphed into a powerful, general-purpose tool for managing and manipulating filesystems and the data spread about them. The dependency-light `datreant.core` package is pure Python, BSD-licensed, and openly developed, and the `datreant` namespace is designed to support useful extensions to the core objects. It is the hope of the authors that `datreant` continues to grow in a way that benefits the wider scientific community, smoothing the common pain point of data glut and filesystem management.

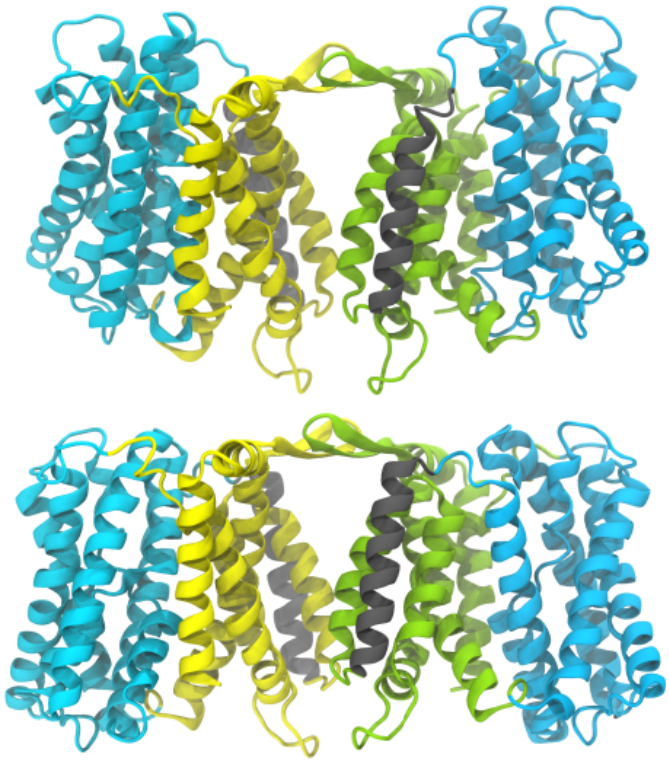

Fig. 2: *A cartoon rendering of an outward-open model (top) and an inward-open crystallographic structure (PDB ID: 4AU5 [Lee14]) (bottom) of* Escherichia coli *NhaA.*

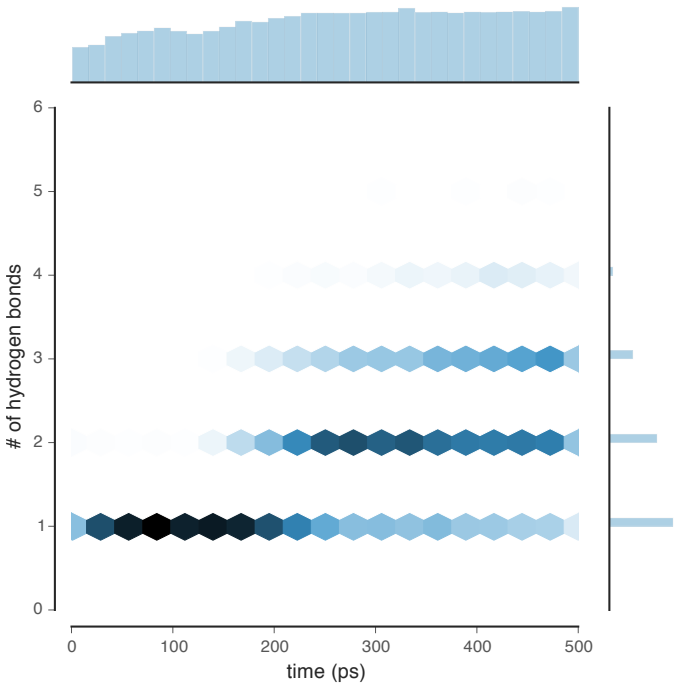

Fig. 3: *The number of hydrogen bonds between the core and dimerization domain during a conformational transition between the inward-open and outward-open state of EcNhaA.*

Acknowledgements

DLD was in part supported by a Molecular Imaging Fellowship from the Department of Physics at Arizona State University. SLS was supported in part by a Wally Stoelzel Fellowship from the Department of Physics at Arizona State University. ML was supported by the Max Planck Society. RG was supported by BBSRC grant BB/J014478/1. OB was supported in part by grant ACI-1443054 from the National Science Foundation; computational resources for OB's work were in part provided by the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation grant number ACI-1053575 (allocation MCB130177 to OB).

References

[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)

[Roc15] Matthew Rocklin. Dask: Parallel Computation with Blocked algorithms and Task Scheduling, Proceedings of the 14th Python in Science Conference, 130-136 (2015)

[Jai15] A. Jain, S. P. Ong, W. Chen, B. Medasani, X. Qu, M. Kocher, M. Brafman, G. Petretto, G.-M. Rignanese, G. Hautier, D. Gunter, and K. A. Persson. FireWorks: a dynamic workflow system designed for high-throughput applications. Concurrency Computat.: Pract. Exper., 27: 5037–5059. doi: 10.1002/cpe.3505 (2015)

[McK10] Wes McKinney. Data Structures for Statistical Computing in Python, Proceedings of the 9th Python in Science Conference, 51-56 (2010)

[MiA11] N. Michaud-Agrawal, E. J. Denning, T. B. Woolf and O. Beckstein. MDAnalysis: A toolkit for the analysis of molecular dynamics simulations, J Comp Chem, 32: 2319-2327. doi: 10.1002/jcc.21787 (2011)

[Lee14] C. Lee, S. Yashiro, D. L. Dotson, P. Uzdavinys, S. Iwata, M. S. P. Sansom, C. von Ballmoos, O. Beckstein, D. Drew, and A. D. Cameron. Crystal structure of the sodium-proton antiporter NhaA dimer and new mechanistic insights, J Gen Physiol, 144:529–544. doi: 10.1085/jgp.201411219 (2014)

Comparison of machine learning methods applied to birdsong element classification

David Nicholson[‡*]

Abstract—Songbirds provide neuroscience with a model system for understanding how the brain learns and produces a motor skill similar to speech. Much like humans, songbirds learn their vocalizations from social interactions during a critical period in development. Each bird's song consists of repeated elements referred to as "syllables". To analyze song, scientists label syllables by hand, but a bird can produce hundreds of songs a day, many more than can be labeled. Several groups have applied machine learning algorithms to automate labeling of syllables, but little work has been done comparing these various algorithms. For example, there are articles that propose using support vector machines (SVM), K-nearest neighbors (k-NN), and even deep learning to automate labeling song of the Bengalese Finch (a species whose behavior has made it the subject of an increasing number of neuroscience studies). This paper compares algorithms for classifying Bengalese Finch syllables (building on previous work [https://youtu.be/ghgniK4X_Js]). Using a standard cross-validation approach, classifiers were trained on syllables from a given bird, and then classifier accuracy was measured with large hand-labeled testing datasets for that bird. The results suggest that both k-NN and SVM with a non-linear kernel achieve higher accuracy than a previously published linear SVM method. Experiments also demonstrate that the accuracy of linear SVM is impaired by "intro syllables", a low-amplitude high-noise syllable found in all Bengalese Finch songs. Testing of machine learning algorithms was carried out using Scikit-learn and Numpy/Scipy via Anaconda. Figures from this paper in Jupyter notebook form, as well as code and links to data, are here: https://github.com/NickleDave/ML-comparison-birdsong

Index Terms—machine learning,birdsong,scikit-learn

Introduction

Songbirds as a model system for the study of learned vocalizations

Songbirds provide an excellent model system through which we can understand how the brain learns and produces motor skills like speech [FEE2010]. Like humans, songbirds learn to vocalize during a critical period in development. During that critical period, they require social interactions, sensory feedback, and practice to learn their vocalizations, just like humans. The songbird brain contains a network of areas specialized for learning and producing song, known as the song system. These brain areas occur only in songbirds, not in birds that do not learn song (e.g., a pigeon). At the same time, all bird brains contains most of the major regions found in the human brain, and the song system sits within these regions that are conserved across evolution. Because of these similarities, we can learn about how our own brains work by studing the songbird brain. For example, studies of songbirds have contributed greatly to our understanding of the basal ganglia [DOUPE2005].

* *Corresponding author: dnicho4@emory.edu*
‡ *Emory University, graduate program in Neuroscience, Biology department*

Machine-learning methods for labeling elements of song

Analysis of birdsong (for neuroscience or the many other fields that study this behavior) typically focuses on "syllables" or "notes", recurring elements in the song. An example song is shown in 1.

Each individual has a unique song that bears some similarity to the song of the bird that tutored it, but is not a direct copy. To analyze song, experimenters label syllables by hand. Typically the experimenter records one bird at a time while carrying out a behavioral experiment. However, each songbird produces thousands of songs a day, more than can be labeled.

In order to deal with this mountain of data, some labs have developed automated analyses. One popular approach scores songs based on similarity of spectrograms, without labeling syllables [TCHER2000]. Another method uses semi-automated clustering to label a birds' syllables, and then measures changes in acoustic and temporal structure of song over days using a distance metric [WU2008]. Other approaches make use of standard supervised learning algorithms to classify syllables, such as Hidden Markov Models [KOGAN2008]. While code for some of these automated analyses is freely available, and there are some repositories of song on-line, to my knowledge almost no work has been done to compare the different algorithms. Note that the studies in this paper are concerned with training a classifier on *syllables* of *one bird's* song to automate labeling of those syllables, **not** with training a classifier to distinguish the song of one bird from another.

The experiments in this paper compare three classifiers applied to one species, the Bengalese Finch. This species is of interest for several reasons. Bengalese Finches depend heavily on auditory feedback throughout life to maintain their vocalizations, much like humans ([SOBER2009] and references therein). In addition, their song tends to have relatively easy-to-quantify acoustic features (e.g., many of the syllables are "low entropy", having a pitchy, whistle-like timbre). Several previously-published studies or open-sourced packages have applied various machine learning techniques to Bengalese Finch song, including support vector machines (SVMs) [TACH2014], and k-Nearest Neighbors (k-NNs) [TROYER2012]. Again, to my knowledge no study has compared these methods with open source code and openly shared data. This study compares the accuracy and amount of training data

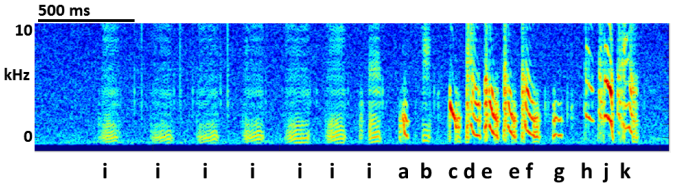

Fig. 1: Spectrogram of Bengalese Finch song. Letters below the time axis, e..g, "i","a","b",..., are labels for syllables, the discrete elements of song separated by brief silent intervals. Frequency (kHz) on the y axis and time on the x axis.

required for SVMs and k-NNs, since at the time of the experiments they were the most recently published methods where code was available. As described in the methods section, for linear SVMs the same C-language library and the same features were used as in [TACH2014], and for k-NN a set of features similar to those used by [TROYER2012] and other songbird researchers was used. The Sci-Kit Learn library [PEDREGOSA2011] provided a convenient API to train both k-NN and support vector machines with non-linear kernels for comparison with the linear SVM results.

Methods

All code used can be found at https://github.com/NickleDave/ML-comparison-birdsong/. Instructions to repeat the experiments are in https://github.com/NickleDave/ML-comparison-birdsong/tree/master/experiment_code. Results and data can be downloaded from http://www.nicholdav.info/data. That page includes files of the features used with the machine learning algorithms to train classifiers, and an example day of raw song files from one bird presented in this paper. Instructions for how to use the feature extraction scripts to reproduce the related file of features from that day of song are in https://github.com/NickleDave/ML-comparison-birdsong/tree/master/feature_extraction_code.

Data acquisition

Song was recorded from four birds, and two to four days worth of songs from each bird were labeled by hand, using custom software written in Labview and Matlab (the Labview program EvTAF for recording, and associated Matlab code for labeling and analysis [TUMER2007]). In some cases more than one person labeled song from a given bird, but everyone that labeled song referred to an agreed-upon rubric for the labels given to syllables. Extra attention was given to the labels because the song was used in behavioral experiments that could have potentially changed syllable acoustics and sequence. All the song used in this study, however, was "baseline" song recorded before the behavioral experiments. Hence I am very confident in this ground truth set.

Raw audio files were bandpass filtered to retain signal between 500 hz and 10 kHz, then smoothed with a Hanning filter. The smoothed signal was segmented into syllables by finding where its amplitude crossed a threshold and where the resulting segments were a minimum duration with a minimum interval between them. The threshold, minimum segment duration, and minimum interval between segments were kept constant for all songs from a given bird except in occassional cases where this method segmented the syllable incorrectly (e.g. because of background noise in the recording).

Feature extraction for use with machine learning algorithms

Once syllables were segmented, features were extracted from them to be used by the machine learning algorithms. Matlab scripts were used for feature extraction. See https://github.com/NickleDave/ML-comparison-birdsong/master/feature_extraction_code/ for this code and for equivalents written in Python using the Matplotlib [HUNTER2007] and Numpy [VANDERWALT2011] packages. The Python versions of the code return slightly different values because of floating point error. I do not expect that using the Python code would qualitatively change the results, but I did not test this. Duration and amplitude features were based on the raw signal; all other features were extracted from spectrograms.

Experiments based on [TACH2014] used the features in that paper, calculated with the code kindly provided by R.O. Tachibana.

For the k-Nearest Neighbor experiments, I used a feature set consisting of: the syllable duration, as well as the duration of the preceding and following syllables, and the preceding and following 'silent gaps' separating the syllables; the Root-Mean-Square amplitude; the spectral entropy; the 'high-low ratio' (power in the 5-10 kHz range / power in the 0-5 kHz range); delta entropy (entropy at 80% of the syllable's duration - entropy at 20% of the syllable's duration); and delta high-low ratio (again the difference at 80% and 20% of the syllable's duration).

Comparison of machine learning algorithms

The goal of comparing algorithms was to determine which could achieve the highest accuracy with the smallest amount of hand-labeled training data. The amount of training data took the form of the number of songs used to train the classifiers. Algorithms were trained by number of songs instead of number of samples because it is most natural for an experimenter to hand-label a set number of songs. This also guaranteed that the frequency of each class of syllable in the training set approximated its frequency in the population. Roughly speaking, less common syllables appeared ~10^3 times in the entire training set while more common syllables appeared ~10^4 times. Preliminary experiments comparing the accuracy of this method to accuracy when the same number of samples for each class was used did not suggest that there was any effect of class imbalance.

Each type of classifier was trained with k songs where k belongs to the set {3,6,9,...27,33,39}. For each k, 5-fold cross validation was used to estimate the accuracy of every classifier. Accuracy was measured as average accuracy across all classes of syllable, because the goal is to achieve the highest accuracy possible for all classes. For every fold, k songs were chosen at random from the training set. This training set consisted of one full day of song, ranging from 100-500 songs depending on the bird. After a classifier was trained with the samples in the k randomly chosen songs, its accuracy was determined on a separate testing set. The testing set consisted of 1-3 additional days of hand-labeled song; no songs from the training data were used in the testing data.

There were three types of models tested: the linear support vector machine as described in [TACH2014], the k-Nearest

Neighbors algorithm, and a support vector machine with a radial basis function as the kernel. Hence, for the 3-song condition, 3 different songs were drawn randomly 5 times, and each time all 3 algorithms were trained with the syllables from those songs, and lastly the accuracy was calculated. All feature sets were z-standardized before training.

Comparison of all machine learning algorithms was greatly facilitated by Scikit-learn [PEDREGOSA2011]. I did use the Liblinear package [FAN2008] directly, instead of the implementation in Scikit-learn, to follow as closely as possible the methods in [TACH2014] (see http://scikit-learn.org/stable/modules/linear_model.html#liblinear-differences). I interacted with Liblinear through the Python API (https://github.com/ninjin/liblinear/tree/master/python) compiled for a 64-bit system. The hyperparameters were those used in [TACH2014]: L2-regularized L2-loss with the cost parameter fixed at 1. Both k-Nearest Neighbors (k-NN) and the support vector machine with radial basis function (SVM-RBF) were implemented via Scikit-learn. For k-NN, I weighted distances by their inverse because I found empirically that this improved classification. I did not test other weightings. For SVM, the RBF hyperparameters 'C' and 'gamma' were found for each set of training samples using grid search.

Results

Both k-NN and SVM with a nonlinear kernel yield higher average accuracy than linear SVM

The main result of this paper is presented in 2. It shows that the average accuracy across classes, i.e. song syllables, was higher for k-NN and for SVM with a non-linear kernel than for linear SVM. (The non-linear kernel is a radial basis function, so the classifier will be abbreviated SVM-RBF). The validation curves for k-NN (blue line) and SVM-RBF (black line) rise more quickly than the curve for linear SVM (red line), indicating they achieve higher accuracy with less training data. Also notice that all the curves reach an asymptote, and that for three of four birds, both k-NN and SVM-RBF achieve higher accuracy at this asymptote than linear SVM. For bird 4 (lower right axis), linear SVM eventually achieved higher accuracy than k-NN, given enough training data, but never reached the accuracy of the SVM-RBF classifier.

As explained in the Methods section, accuracy was estimated with cross validation. Briefly: random samples were drawn from the training data and accuracy was measured on a completely separate set of testing data. Importantly, the number of samples in the testing data set was roughly on the order of the number of syllables that are hand-labeled for a typical songbird behavioral experiment. (Some previous studies have estimated accuracy for large data sets by bootstrapping from a smaller set of hand-labeled testing data.) Note that the comparison uses accuracy averaged across classes as a metric, because the ideal case would be to have each type of syllable classified perfectly. Note also that classifiers were trained with a number of songs instead of number of samples, because it is typical for a songbird reseacher to label complete songs instead of labeling e.g., 100 samples or "sixty seconds" of syllables. Each time a Bengalese Finch sings its song, it may sing a varying number of syllables. Hence one set of three songs drawn at random from the training data might have a different number of samples than another set. This difference in number of training samples accounts for some of the variance in accuracy scores, but k-NN and SVM-RBF clearly achieve higher accuracy than linear SVM in spite of this added variance.

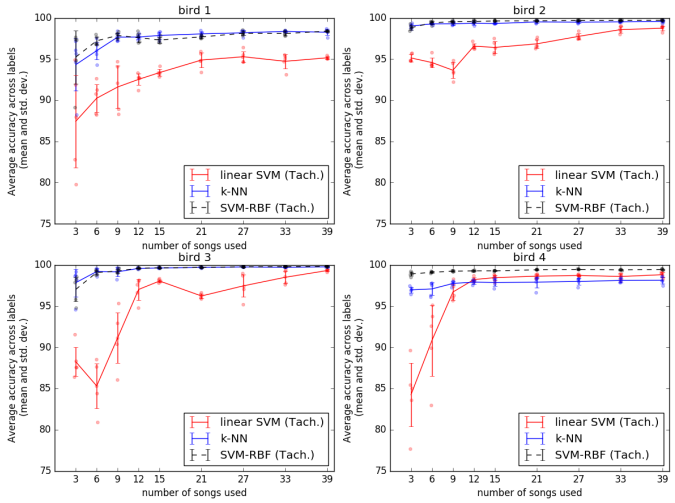

Fig. 2: Validation curves showing accuracy vs. number of songs used to train classifiers. Y axis: average accuracy across labels, x axis: number of songs used to train the classifiers. *Points are accuracy for each fold of 5-fold cross validation. Validation curves are mean, and error bars are standard deviation across five folds. Red line: linear support vector machine (linear SVM); blue line: k-Nearest Neighbors (k-NN); black line: support vector machine with radial basis function as kernel (SVM-RBF). Note that accuracy is average accuracy across classes, i.e., song syllables.*

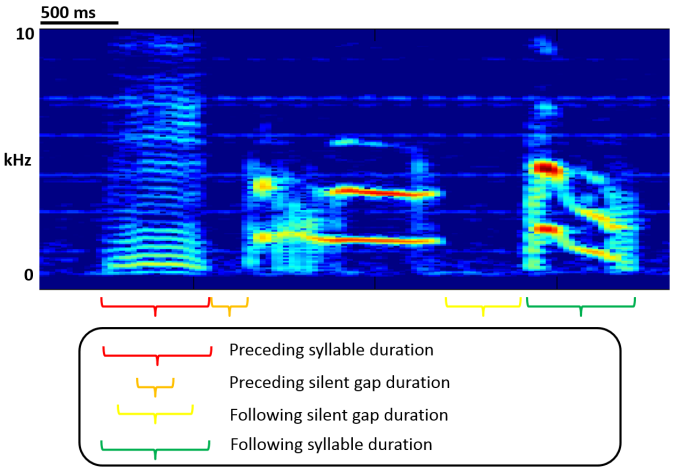

Fig. 3: Features added that improved k-NN accuracy

It is also important to note that the k-NN classifier used a distinct set of features from those used in [TACH2014] because of concerns that the number of dimensions would impair k-NN accuracy. (In high-dimensional spaces, everything is close to everything, so the distances used by k-NN to determine nearest neighbor become uninformative, see [BEYER1999].) Instead, the k-NN algorithm used a small set of acoustic parameters that are commonly measured in songbird research, in addition to features from neighboring syllables that greatly improved the accuracy of the algorithm. These features from neighboring syllables are schematized in 3. The SVM-RBF classifier used the exact same features as the linear SVM. Experiments below address the question of whether the differences between classifiers shown in 2 arise from a difference in features used or a difference in the classifiers themselves.

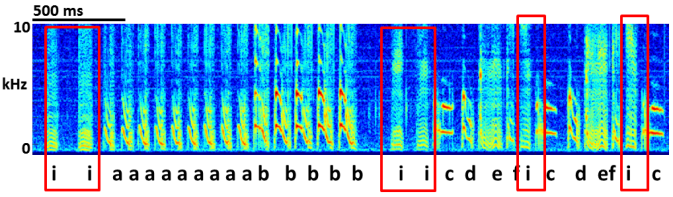

Fig. 4: Introductory notes are low-amplitude high-noise syllables that often occur at the start of song Red boxes indicate introductory notes.

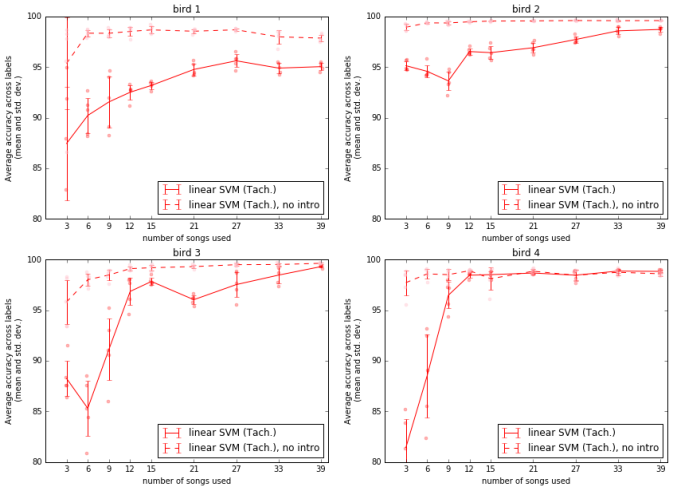

Fig. 5: Accuracy vs. number of songs used to train linear SVM, with intro syllables removed from training and test sets. Y axis: average accuracy across labels, x axis: number of songs used to train the linear SVM. *Removing intro syllables greatly increased accuracy for three of four birds.*

Intro syllables impair the accuracy of linear SVMs

The result in 2 was suprising, given the previously reported accuracy for linear SVMs applied to Bengalese finch song [TACH2014]. One potential cause for the impaired accuracy of the linear SVM method is the presence in song of "introductory notes", low-amplitude, high-entropy syllables that often occur at the start of song, hence their name. Examples are shown in 4. Because these syllables have low amplitude, it can be hard to detect their onset and offset, so the distribution of their duration will have much more variance than other syllabes. Likewise because they are high entropy, any feature derived from the spectrum will also be more variable. For example, measuring the "pitch" of an intro syllable by finding the peak in its power spectrum would yield wildly varying values, because there is no consistent peak to measure across renditions of the syllable. These sources of variability probably make it harder to separate intro syllables from other types.

The next experiment determined whether removing intro syllables from the training and test sets would rescue the accuracy of the linear SVM. For the song of the birds used in this study, removing intro syllables greatly increased accuracy, as shown in 5. Note that this result is consistent with the findings of [TACH2014]. In their final set of experiments they found that the syllables most likely to be misclassified were those at the beginning and end of song, i.e., intro syllables.

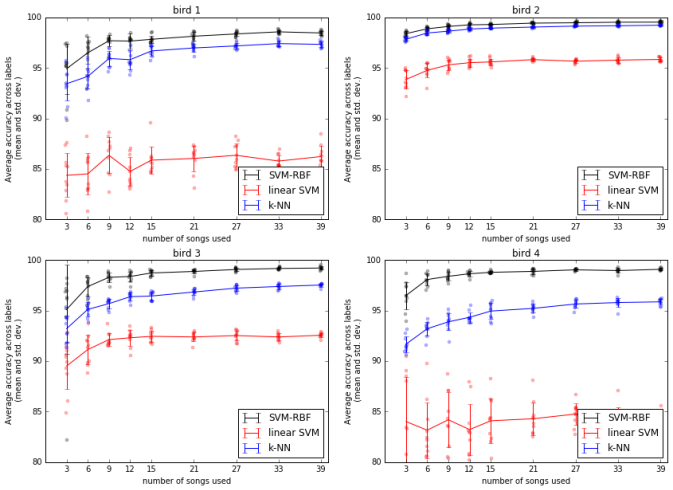

Fig. 6: Accuracy v. number of songs used to train SVM-RBF, k-NN, and linear SVM, all trained with the same acoustic features Y axis: average accuracy across labels, x axis: number of songs used to train.

When using the same features to train all models, SVM-RBF still outperforms k-NN that in turn outperforms linear SVM

The results in 2 showed that k-NN and SVM-RBF can yield higher average accuracy than linear SVM. However, the feature set for training the k-NN differed from the feature set for the SVM classifiers. As described above, a different feature set was used for k-NN because of concerns that the 536-dimensional feature vector would yield poor results (see [BEYER1999] for an in-depth study of how the number of features affects k-NN accuracy).

This leaves unanswered the question of whether differences in accuracy are due to the features used, or due to the ability of the algorithms to fit models to the feature space (or some combination of both). To address this question, the same approach was used to compare all three algorithms, but this time classifiers were trained with a set of 20 acoustic features from [TACH2014]. For all 4 birds tested, SVM-RBF acheived higher average accuracy with less training data than k-NN, and k-NN outperformed linear SVM, as shown in 6.

All three algorithms were also compared with the feature set originally used for training k-NN classifiers. Here, the results were less clear. As shown in 7, for three birds, SVM-RBF performed about as well as k-NN, and both performed better than linear SVM. For bird 4, k-NN on average performed better but the replicates showed high variance in the average accuracy.

Conclusion

There are two clear results from these experiments. First, the linear SVM method proposed in [TACH2014] is impaired by intro syllables in the songs of Bengalese Finches. Second, use of the radial basis function as a kernel can improve SVM performance when applied to the features in [TACH2014].

These results do not answer the question of how often the method of [TACH2014] will be impaired by any given bird's song. What can be said is that for two of the four birds tested, average accuracy for linear SVM did not approach 99% until at least 33 songs were used to train the classifier (birds 2 and 3, 2), and for one bird, average accuracy never went above 97% (bird 1, 2). By comparison, when training SVM-RBF classifiers with the same

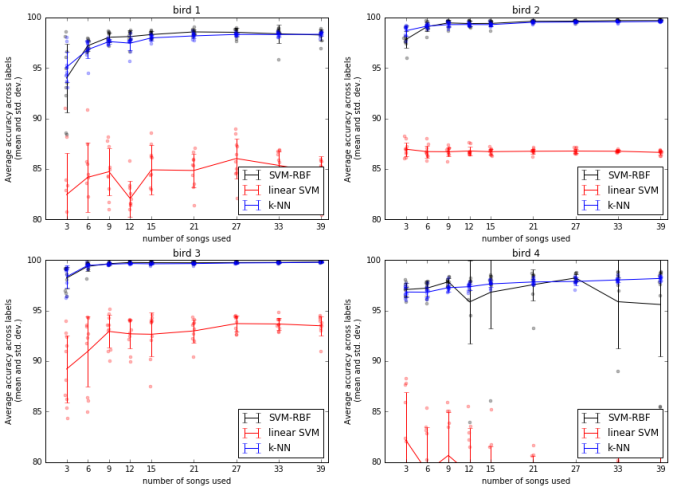

Fig. 7: Accuracy v. number of songs used to train SVM-RBF, k-NN, and linear SVM, all trained with features originally used for k-NN Y axis: average accuracy across labels, x axis: number of songs used to train.

feature set, 6 songs was enough to achieve >99% average accuracy for 3 of the 4 birds (as shown in 2).

When the feature set is held constant, for all four birds, linear SVM is always outpeformed by k-NN and SVM-RBF. Again, it can not be said based on the results how often this would be the case for any given Bengalese finch's song. But the large difference in average accuracy between linear SVM and the other two methods for the four birds tested here (:ref: *fig6* and :ref: *fig7*) certainly suggests that in general the other two methods will outperform linear SVM. Interestingly, the set of twenty acoustic features developed by [TACH2014] yielded what appears to be a large difference in accuracy between the three algorithms. This result shows that instead of using a 536-feature vector with the linear SVM, one can use the 20 features with SVM-RBF, and achieve higher accuracy with less training time and data. (Training time was not measured for each classifier but the experiments in 2 took a week to run while the experiments in 6 took two days to run. This difference was due mainly to the time required for the grid search for SVM-RBF hyperparameters.)

It remains to be tested whether any differences in accuracy translate into meaningful differences in results obtained from analysis of song. That is to say that linear SVMs trained with the original [TACH2014] feature set might yield good enough classifiers to detect some changes in song that experimenters care about. Data sets from songbird behavioral experiments, not just from baseline song, should be used to determine whether this is the case.

There are also other issues to be dealt with to make machine learning methods practical for birdsong researchers. One is how well each method can provide an estimate that a given classification is correct. The libSVM library, for example, can provide probability estimates using a computationally expensive 5-fold cross-validation. But, because the soft margin in support vector machine training algorithms allows some misclassifications, some samples will be misclassified yet still appear to have a high probability of being correct. As [KOGAN2008] recognized in their study, it is also important to determine how well all of these algorithms deal with the presence of sounds that are not part of song, e.g., calls, wing flaps, etc. Such events are rare enough that they may be difficult to detect without changes to the training algorithm, but frequent enough that if misclassified as syllables they could affect analyses of song.

Taken together, the results here demonstrate the importance of comparing how different classifiers perform in a given problem domain. This comparison is an attempt to build upon the previous studies cited, studies that showed that machine learning methods can facilitate much more fine-grained analyses of birdsong. The results here suggest there are still some issues with practical application of machine learning to birdsong, however. Sharing code, results, and raw data will help resolve these issues and lead to better results for the biologists and machine learning scientists studying birdsong.

Acknowledgements

Thank you to Samuel J. Sober for supporting this work in every way. I would also like to acknowledge helpful input from Shamim Nemati, Supreeth Prajwal, Alex Dunlap, and Kyle Srivastava. Thanks also to all members of the Sober lab, my science family, especially to Jonah Queen, undisputed champion and reigning king of syllable labeling.

References

[DOUPE2005] Doupe, Allison J., et al. *Birdbrains could teach basal ganglia research a new song.* Trends in neurosciences 28.7 (2005): 353-363.

[FEE2010] Fee, Michale S., and Constance Scharff. *The songbird as a model for the generation and learning of complex sequential behaviors.* ILAR journal 51.4 (2010): 362-377.

[TCHER2000] Tchernichovski, Ofer, et al. *A procedure for an automated measurement of song similarity.* Animal Behaviour 59.6 (2000): 1167-1176.

[WU2008] Wu, Wei, et al. *A statistical method for quantifying songbird phonology and syntax.* Journal of neuroscience methods 174.1 (2008): 147-154.

[KOGAN2008] Kogan, Joseph A., and Daniel Margoliash. *Automated recognition of bird song elements from continuous recordings using dynamic time warping and hidden Markov models: A comparative study.* The Journal of the Acoustical Society of America 103.4 (1998): 2185-2196.

[SOBER2009] Sober, Samuel J., and Michael S. Brainard. *Adult birdsong is actively maintained by error correction.* Nature neuroscience 12.7 (2009): 927-931.

[TACH2014] Tachibana, Ryosuke O., Naoya Oosugi, and Kazuo Okanoya. *Semi-automatic classification of birdsong elements using a linear support vector machine.* PloS one 9.3 (2014): e92584.

[TROYER2012] http://www.utsa.edu/troyerlab/software.html

[BEYER1999] Beyer, Kevin, et al. *When is "nearest neighbor" meaningful?.* Database theory—ICDT'99. Springer Berlin Heidelberg, 1999. 217-235.

[FAN2008] Fan, Rong-En, et al. *LIBLINEAR: A library for large linear classification.* The Journal of Machine Learning Research 9 (2008): 1871-1874.

[TUMER2007] Tumer, Evren C., and Michael S. Brainard. *Performance variability enables adaptive plasticity of 'crystallized' adult birdsong.* Nature 450.7173 (2007): 1240-1244.

[VANDERWALT2011] Van Der Walt, Stefan, S. Chris Colbert, and Gael Varoquaux. *The NumPy array: a structure for efficient numerical computation.* Computing in Science & Engineering 13.2 (2011): 22-30.

[HUNTER2007] Hunter, John D. *Matplotlib: A 2D graphics environment.* Computing in science and engineering 9.3 (2007): 90-95.

[PEDREGOSA2011] Pedregosa, Fabian, et al. *Scikit-learn: Machine learning in Python.* The Journal of Machine Learning Research 12 (2011): 2825-2830.

MONTE Python for Deep Space Navigation

Jonathon Smith, William Taber, Theodore Drain, Scott Evans, James Evans, Michelle Guevara, William Schulze, Richard Sunseri, Hsi-Cheng Wu[‡*]

https://youtu.be/E3RhKKpm4TM

Abstract—The Mission Analysis, Operations, and Navigation Toolkit Environment (MONTE) is the Jet Propulsion Laboratory's (JPL) signature astrodynamic computing platform. It was built to support JPL's deep space exploration program, and has been used to fly robotic spacecraft to Mars, Jupiter, Saturn, Ceres, and many solar system small bodies. At its core, MONTE consists of low-level astrodynamic libraries that are written in C++ and presented to the end user as an importable Python language module. These libraries form the basis on which Python-language applications are built for specific astrodynamic applications, such as trajectory design and optimization, orbit determination, flight path control, and more. The first half of this paper gives context to the MONTE project by outlining its history, the field of deep space navigation and where MONTE fits into the current Python landscape. The second half gives an overview of the main MONTE libraries and provides a narrative example of how it can be used for astrodynamic analysis. **For information on licensing MONTE and getting a copy visit** montepy.jpl.nasa.gov **or email** mdn_software@jpl.nasa.gov.

Index Terms—astrodynamics, aerospace, orbit, trajectory, JPL, NASA

History

The United States began its reconnaissance of the solar system in the early 1960s. As NASA developed new technologies to build and operate robotic probes in deep space, JPL was working out how to guide those probes to their destinations. In order to fly spacecraft to Mars or Jupiter, engineers needed a way to model their trajectories through interplanetary space. This was partly a problem of **astrodynamics**, a field of study that mathematically describes how man-made objects move through space. It was also a problem of computation because engineers needed a way to solve these complex astrodynamic equations for real spacecraft. Beyond modeling the motion of spacecraft, engineers needed a way to measure the location of spacecraft over time so they could make informed corrections to their models. They also needed a way of designing engine burns, or maneuvers, that would nudge a wayward probe back on course.

These efforts, collectively known as *deep space navigation*, quickly became coupled with software and computing. The first programs JPL wrote to navigate spacecraft were written on punch-cards and processed through an IBM 7090 mainframe. [Eke05] Advances in computing technology were eagerly consumed by navigators, as more storage and faster processing meant the models used to fly spacecraft could be made increasingly detailed and sophisticated.

Starting in 1964, a group of engineers, led by Ted Moyer, began developing the astrodynamic algorithms and software that would eventually become the Double Precision Trajectory and Orbit Determination Program, or DPTRAJ/ODP ([Moy71], [Moy03]). Over its forty-plus years of active life, JPL engineers used the DPTRAJ/ODP to navigate the "Golden Age" of deep space exploration. This included the later Mariner and Pioneer missions, Viking, Voyager, Magellan, Galileo, Cassini and more. Also over this time, its base language moved through Fortran IV, Fortran V, Fortran 77 and Fortran 95 as the computational appetites of navigators grew ever larger.

By 1998 it was clear that the aging DPTRAJ/ODP needed to be updated once again. Rather than initiate another refactor, JPL's navigation section commissioned a new effort that would depart from its predecessor in two important ways. First, the new software would be an object-oriented library, written in C++ and exposed to the user as a Python-language library. Second, it would be a general-purpose astrodynamic computing platform, not a dedicated navigation program like the DPTRAJ/ODP. The goal was to create a single library that could be used for astrodynamic research, space mission design, planetary science, etc., in addition to deep space navigation. This new project was affectionately named the Mission Analysis, Operations, and Navigation Toolkit Environment, or MONTE-Python for short.

Throughout the first half of the 2000s, MONTE was carefully constructed by reshaping the algorithms under-pinning the DPTRAJ/ODP into a rigorously tested and well documented object-oriented software package. In 2007, MONTE had its first operational assignment navigating NASA's Phoenix lander to a successful encoutner with Mars. Since 2012, MONTE has powered all flight navigation services at JPL, including the Cassini extended mission, Mars Science Laboratory, MAVEN, GRAIL, Dawn, Mars Reconnaissance Orbiter, Juno, and more. [Eva16]

Deep Space Navigation

At JPL, the practice of navigating robotic probes in deep space is broken down into three interrelated disciplines: (1) designing a reference trajectory which describes the planned flight path of the spacecraft (*mission design*), (2) keeping track of the spacecraft position while the mission is in flight (*orbit determination*), and (3) designing maneuvers to bring the spacecraft back to the reference trajectory when it has strayed (*flight path control*, Figure 1).

The process of designing a spacecraft reference trajectory begins at the earliest stages of mission planning. Navigators work

* *Corresponding author: jonathon.j.smith@jpl.nasa.gov*
‡ *Jet Propulsion Laboratory, California Institute of Technology / NASA*

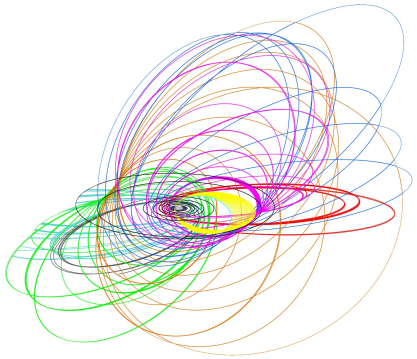

Fig. 1: *Illustration of Cassini's reference trajectory at Saturn. The mission designers built this trajectory, and the orbit determination and maneuver design teams keep the spacecraft flying on these orbits during the mission.*

closely with mission science teams to put together a reference orbit that allows the spacecraft to take all the desired science measurements. They also work with mission planners and spacecraft system engineers to make sure that the spacecraft is able to withstand the rigors of its planned trajectory. Through a process of increasingly detailed iterations, a process which often takes years, the mission reference trajectory is produced. This reference trajectory serves as the flight plan for the spacecraft. It will be up to the orbit determination and flight path control teams to make sure the spacecraft follows this flight plan when the spacecraft finally launches.

The job of the orbit determination team is to keep track of where the spacecraft has been (orbit reconstruction), where it is currently (orbit determination), and where it will go in the future (orbit prediction). The spacecraft is always drifting away from its planned flight path because of small disturbances it encounters in space. Even the slight pressure of sunlight on the spacecraft can add up over time and push a mission off course. The trajectory designers do their best to account for these disturbances when creating the reference orbit, but there is no accounting for the randomness and unpredictability of the real world. To further complicate matters, once the spacecraft leaves the launch-pad, it can no longer be directly observed. Orbit determination analysts must process various forms of tracking data that are tied mathematically to the evolution of the spacecraft orbit to determine its position at any given time.

Once the orbit determination team has a good estimate for the current location of the spacecraft, the flight path control team is responsible for evaluating how far the spacecraft has drifted from the reference trajectory and designing a maneuver to get the spacecraft back on course. The result of this maneuver design is a ΔV vector, which stands for delta-velocity or change in velocity. This ΔV vector represents the direction and magnitude of the required change in the spacecraft velocity which must be accomplished to get the spacecraft back on course. Once in hand, this ΔV vector will be sent to the spacecraft propulsion team, who will decompose it into thruster firings on the spacecraft. These will be uplinked to the spacecraft, which will then perform the maneuver.

After a maneuver has been performed, the cycle repeats. Perhaps the thrusters were slightly misaligned or the engine cutoff was a second too late. The orbit determination team must examine more tracking data to find out. This iterative relationship between orbit determination and flight path control continues without pause through the lifetime of a flight mission. The spacecraft is constantly wandering off, and must be patiently brought back on course.

MONTE as a Platform

As previously mentioned, MONTE was built to be a general purpose astrodynamic computing platform, not a dedicated navigation application. It supplies the models and computational algorithms needed for trajectory design, orbit determination and flight path control but doesn't force the end-user into any specific workflow or interface. As a result, before MONTE can be used on a flight mission, it must be *deployed* for that mission. This entails using MONTE in cooperation with other applications and libraries to assemble a custom navigation framework.

The process of deploying MONTE for a flight mission can be quite involved. The effort to build a navigation system for the Cassini Extended Mission took over two years, and required the use of many other Python libraries in addition to MONTE. The resulting navigation framework can not be properly characterized as MONTE itself. Rather, it is a custom application built using the MONTE library to perform navigation for that specific mission.

This is important to note because it illustrates the way in which MONTE is likely to be useful to those outside JPL. Deep space navigation is (not yet at least) a high-demand field. The majority of astrodynamic computing occurs in other contexts such as Earth-centered navigation (weather and communication satellites, etc), collision avoidance analysis (making sure two spacecraft don't collide), cooperative rendezvous (docking a cargo-ship to the International Space Station) and non-cooperative rendezvous (capturing a malfunctioning satellite), etc. Much the same way that MONTE can be configured and deployed for deep space navigation, it can also be brought to bear on these and other problems across the aerospace industry.

MONTE provides a solid foundation of core systems that make it attractive as a general purpose astrodynamic platform. These include models for trajectories and trajectory queries, coordinate frames and rotations, high-precision time, astrodynamic event searches, numerical integrators, configurable optimizers, and many more. By starting with MONTE, a user can focus on solving the problem at hand, and leave the important-but-incidental infrastructure to MONTE.

MONTE and the Python Ecosystem

MONTE has a decidedly friendly stance when it comes to working with other libraries in the Python scientific computing stack. It makes heavy use of many open-source Python libraries such as matplotlib and IPython (Jupyter), and reciprocally tries to make it easy for users of these systems to interface with MONTE. Many of MONTE's classes can transform themselves into NumPy data types --- a common pattern is for MONTE classes to have a `.toArray` method which returns a `numpy.ndarray`. Additionally, the MONTE team has a history of collaboration with matplotlib dating all the way back to the early 2000s. They have contributed code that makes matplotlib able to natively plot MONTE's unit and time systems, and have also open-sourced a custom matplotlib styling-system (github.com/nasa/mplStyle) developed in house.

The MONTE project started in 1998 at a time when the Python language was still relatively new. As a result, MONTE has several

custom systems that are redundant in the current Python landscape. For instance, MONTE developed an interactive shell similar to IPython and has several numerical computing classes that would generally be dispatched to NumPy in a brand new project.

Historical quirks aside, MONTE considers itself a member of the Python scientific programming community and aims to integrate as seamlessly as possible with other Python libraries. It can be embedded in custom GUI applications, run on a back-end server, executed in parallel across a cluster of nodes, and pretty much anything else you would expect of a dynamic, well constructed Python library.

Library Overview

Most of the functionality of MONTE is encapsulated in the `Monte` and `mpy` libraries. `Monte` is written in C++ and wrapped in Python. It is presented to the end user as a normal, importable Python-language module. The `mpy` module is written entirely in Python and contains higher level applications built using `Monte` and other Python libraries.

Convention is to import the main `Monte` library as `M`. Throughout this paper, if a class is referred to with the prefix `M.`, it means this class belongs to the main MONTE library (e.g. `M.TrajLeg`, `M.Gm`, etc). The following example shows a simple script using the `Monte` and `mpy` libraries to get the state of the Cassini spacecraft with respect to Saturn at the time of its Saturn Orbit Insertion (SOI) burn.[1][2]

```
import Monte as M
import mpy.io.data as defaultData

# Set up a project BOA database, and populate it
# with astrodynamic data from default data
boa = defaultData.load([ "time", "body", "frame",
  "ephem/planet/de405"])

# Load the Saturn satellite ephemeris and Cassini
# trajectory into our BOA database
boa.load("saturn_satellites.boa")
boa.load("cassini_trajectory.boa")

# Define time of SOI
soiTime = M.Epoch("01-JUL-2004 02:48:00 UTC")

# Get the trajectory manager from the BOA database
traj = M.TrajSetBoa.read(boa)

# Request the state of Cassini at SOI from the
# trajectory manager in a Saturn-centered Earth
# Mean Orbit of 2000 coordinate frame
casAtSoi = traj.state(soiTime, "Cassini", "Saturn",
  "EMO2000")
```

Several of MONTE's core systems --- the basic astrodynamic scaffolding that supports its more advanced functionality --- are used in the above example. These are explained in a short tour of MONTE below.

BOA

The Binary Object Archive (BOA) is MONTE's primary data management system. Most MONTE classes that define concrete objects (for instance, `M.Gm` which defines the standard gravitational parameter for a natural body or `M.FiniteBurn` which defines a spacecraft burn) are stored in BOA, and accessed by MONTE's astrodynamic functions from BOA.

BOA is based on the binary XDR data format, which allows data to be written-to and read-from binary on different operating systems and using different transport layers (e.g. you can read and write locally to your hard disk, or over a network connection).

The role that BOA plays in MONTE can perhaps be best understood as "defining the universe" on which MONTE's astrodynamic tools operate. In our example, we populated our "model universe" (e.g. our BOA database) with time systems, natural body data, a planetary ephemeris, the Cassini spacecraft trajectory, etc. We then asked MONTE's trajectory manager (an astrodynamic tool) to examine this particular universe and return the state of Cassini with respect to Saturn.

Default Data

A standard MONTE installation comes with a collection of predefined, publicly available astrodynamic datasets (the "default data depot"). These can be accessed and loaded into a BOA database via MONTE's default data loader (`mpy.io.data`) and serve to help an analyst get a "model universe" up and running quickly.

Time and Units

In the astrodynamic community there are multiple time systems used to describe the dynamics of a spacecraft and to specify the time of an observation. While necessary, multiple systems for specifying time can add considerable complexity to software. In MONTE, time is encapsulated in the `M.Epoch` class, which supports time definition in the TDB, TT, TAI, GPS, UTC, and UT1 systems. This class handles the problem of transforming times between different frames thereby allowing the user to specify times in the most convenient form for their application.

MONTE's unit system supports the notions of time, length, mass, and angle. It has implemented operator overloading to allow unit arithmetic, e.g. dividing a unit length by a unit time results in unit velocity. Most functions that accept unit-quantities also check their inputs for correctness, so supplying a unit length to a function that expects unit time will raise an exception.

Trajectories

MONTE models spacecraft and natural body trajectories in a number of underlying formats; most of the differences involve how many data points along the trajectory are stored, and how to interpolate between these points. In addition, MONTE provides conversion routines which allow some external trajectory formats to be read and written (including NAIF "bsp" files and international "oem" files).

The `M.TrajSet` class is MONTE's trajectory manager, and is responsible for coordinating state requests between all of the trajectories loaded into a given BOA database. It has access to the coordinate frame system (described in the next section) allowing it to make coordinate frame rotations when doing state queries. In fact, most coordinate frame rotations in MONTE are accomplished by simply requesting a state from `M.TrajSet` in the desired frame.

The general steps for building and using trajectories in MONTE are illustrated in Figure 2.

Coordinate Frames

The MONTE trajectory and coordinate frame systems are very analogous and have a tight integration that enables powerful state

1. All MONTE code in this paper is current as of the v121 delivery.

2. Saturn Orbit Insertion was a spacecraft maneuver that occurred as Cassini approached Saturn. It changed the course of the spacecraft so that instead of flying past Saturn, it captured into orbit around the planet.

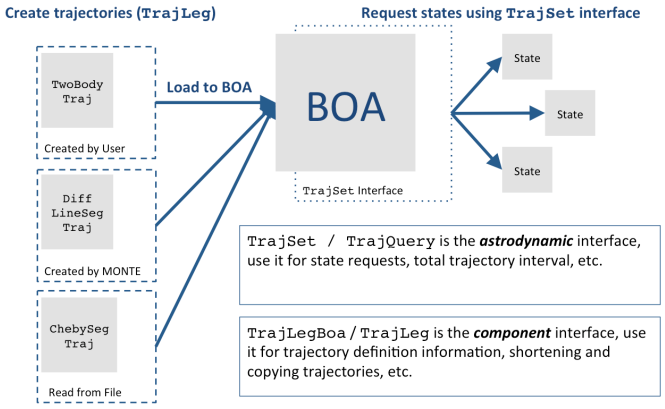

Fig. 2: *Dataflow through MONTE's trajectory system*

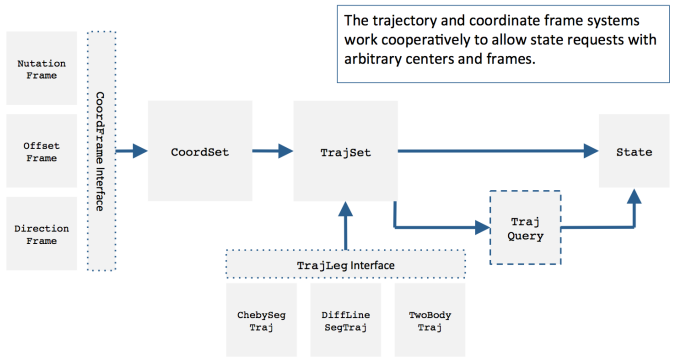

Fig. 3: *Cooperation between MONTE's trajectory and coordinate frame systems*

requests. Figure 3 illustrates these similarities and how the two systems are integrated.

MONTE models coordinate frames in a number of underlying formats and provides conversion routines which allow some external coordinate frame formats to be read and written (including NAIF "ck" files).

Event Finding

MONTE allows a user to search through astrodynamic relationships in a given BOA database in pursuit of particular events. For instance, the `M.AltitudeEvent` class allows a user to search for when a spacecraft is within a certain altitude range from another body.

Numerical Integration

MONTE provides a framework for numerically integrating spacecraft and natural body trajectories, subject to a set of force models such as gravity, solar radiation pressure, atmospheric drag, etc. The resulting trajectory has the Cartesian position and velocity of the body over time, and optionally the partial derivatives of state parameters with respect to parameters in the force models. A walkthrough of setting up MONTE's numerical integration system for a simple gravitational propagation is shown in Figure 4.

In addition to trajectories, MONTE also allows numerical integration of mass (for instance due to burning of propellant), coordinate frames (rigid body dynamics), time (relativistic time transformations) and user-defined ordinary differential equations.

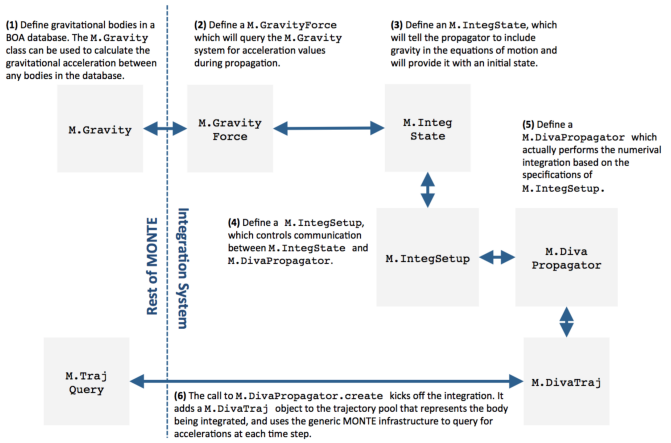

Fig. 4: *Overview of MONTE's numerical integration system.*

Parameters and Partial Derivatives

MONTE's parameter system supports the calculation of partial derivatives for astrodynamic variables, which can then be used in optimization and estimation. Every variable that belongs to the parameter system is responsible for not only calculating its value, but also its partial derivative with respect to any other parameters. These partial derivatives are contained in a special set of classes that employ operator overloading to correctly combine partial derivatives under various mathematical operations. [Smi16]

Example: Exploring bodies in motion

Generally, MONTE is scripted or assembled into custom applications that solve complex end-user problems. However, it is also useful as an off-the-cuff tool to explore astrodynamic relationships as we will see in the narrated example below.

For this example, we will explore the Voyager 2 trajectory. We will identify the time and distance of the Uranus planetary encounter, and also find the time periods where Voyager 2 was in line with the sun. Along the way we will highlight various aspects of MONTE's core systems. Also, if our exploration happens to turn up anything interesting (it will), we will take some time to investigate what we find.

Voyager 2 Trajectory

We begin by specifying the model of the solar system during Voyager's mission. This is done by creating a BOA database and loading the default data sets for planetary ephemerides (the trajectories of all the planets in the solar system), coordinate frames, and body parameters like mass and shape. We will also load in our Voyager 2 trajectory.[3]

3. JPL hosts two excellent websites for accessing trajectory data for natural solar system bodies and deep-space probes. The Horizons website (http://ssd.jpl.nasa.gov/horizons.cgi) is maintained by JPL's Solar System Dynamics group and has an expansive and powerful webapp for getting ephemerides in a variety of formats. The Navigation and Ancillary Data Facility (NAIF) at JPL hosts the navigation section of NASA's Planetary Database System. At its website (http://naif.jpl.nasa.gov/naif/data.html), you will find a host of downloadable binary navigation files, which can be used with the SPICE toolkit, and of course, with MONTE.

For the following examples, we will be using the Voyager 2 spacecraft trajectory, which can be downloaded at http://naif.jpl.nasa.gov/pub/naif/VOYAGER/kernels/spk/. The file name at the time of this writing is "voyager_2.ST+1992_m05208u.merged.bsp", which we will shorten to just "voyager2.bsp" for ease of use.

```
In [1]: import Monte as M
In [2]: import mpy.io.data as defaultData
In [3]: boa = M.BoaLoad()
In [4]: defaultData.loadInto( boa,
   ...:   ["ephem/planet/de405", "frame", "body"] )
In [5]: boa.load( "voyager2.bsp" )
```

The trajectories of Voyager and the natural bodies of the solar system are coordinated by the trajectory manager (`M.TrajSet`) that is supplied by BOA we just created. We can retrieve the trajectory manager using its BOA accessor `M.TrajSetBoa`. Every object that resides in BOA has an accessor (often named `M.ClassNameBoa`) that allows it to be read to and from the database. Once in hand, we can list all the trajectories that are on the BOA using the `M.TrajSet.getAll` method.

```
In [6]: traj = M.TrajSetBoa.read( boa )
In [7]: traj.getAll()
Out[7]: ['Mercury', 'Mercury Barycenter',
         'Venus', 'Venus Barycenter',
         'Earth', 'Earth Barycenter', 'Moon',
         'Mars', 'Mars Barycenter',
         'Jupiter Barycenter', 'Saturn Barycenter',
         'Uranus Barycenter', 'Neptune Barycenter',
         'Pluto Barycenter', 'Sun'
         'Solar System Barycenter', 'Voyager 2']
```

The list of bodies returned by `M.TrajSet.getAll` confirms that we have successfully loaded our solar system and spacecraft. We continue our analysis by checking the span of the Voyager 2 trajectory, e.g. the interval over which we have data, using the `M.TrajSet.totalInterval` method. *Note that if the trajectory has been updated at the NAIF PDS website, the exact span you get may be different than what is listed below.*

```
In [8]: traj.totalInterval( "Voyager 2" )
Out[8]:
TimeInterval(
   [ '20-AUG-1977 15:32:32.1830 ET',
     '05-JAN-2021 00:00:00.0000 ET' ],
)
```

The Voyager 2 trajectory starts just after launch in 1977, extends through the present, and has predictions out into the future. We can use the trajectory manager to request states at any time in this window. For instance, we can find the distance of Voyager 2 from Earth right now. The `M.Epoch.now` static method returns the current time and this can be passed to the trajectory manager to request the state of Voyager 2 with respect to Earth.

```
In [11]: currentTime = M.Epoch.now()
In [12]: vygrTwoNow = traj.state(currentTime,
    ...:   "Voyager 2", "Earth", "EME2000" )
In [13]: vygrTwoNow
Out[13]:
State (km, km/sec)
'Earth' -> 'Voyager 2' in 'EME2000'
at '06-JUN-2014 19:58:35.1356 TAI'
Pos:  4.358633010242671e+09 -7.411125552099214e+09
     -1.302731854689579e+10
Vel: -2.415141211951430e+01  2.640692963340520e+00
     -1.128801136174438e+01
```

We used the `M.TrajSet.state` method to perform our query, which required us to specify the time, target body, reference body, and coordinate frame for the return state. Because `M.TrajSet` has a global view of all the trajectories in our BOA, we can request states with respect to any body for which we have a trajectory, for instance Venus or Neptune.

```
In [14]: vygrTwoNowVenus = traj.state( currentTime,
    ...:   "Voyager 2", "Venus", "EME2000" )
In [15]: vygrTwoNowVenus
Out[15]:
State (km, km/sec)
'Venus' -> 'Voyager 2' in 'EME2000'
at '06-JUN-2014 19:58:35.1356 TAI'
Pos:  4.216416788778397e+09 -7.523453172910529e+09
     -1.306899257275581e+10
Vel: -4.457126033807687e+00 -3.509301445530399e+01
     -2.760459587874612e+01

In [17]: vygrTwoNowNeptune = traj.state(currentTime,
    ...:   "Voyager 2", "Neptune Barycenter", "EME2000" )
In [18]: vygrTwoNowNeptune
Out[18]:
State (km, km/sec)
'Neptune Barycenter' -> 'Voyager 2' in 'EME2000'
at '06-JUN-2014 19:58:35.1356 TAI'
Pos:  2.423407540346480e+08 -5.860459060720786e+09
     -1.229435420991246e+10
Vel:  2.036299646730726e+00 -8.760646249684767e+00
     -1.606470435709401e+01
```

The `M.TrajSet.state` method returns an `M.State` object. `M.State` captures the relative position, velocity and acceleration (or some subset) of one body with respect to another at a given time. It has a number of methods that help with extracting and transforming the information it contains. For instance, we can find the distance from Earth to Voyager 2 like this.

```
In [26]: vygrTwoPoskm = vygrTwoNow.posMag()
In [27]: vygrTwoPoskm
Out[27]:  1.560876331389678e+10 * km

In [28]: vygrTwoPoskm.convert( 'AU' )
Out[28]: 104.33813824888766
```

When reading states from a trajectory you are often interested in making repeated calls for the same body and center but at different times. `M.TrajSet` works fine for this application, but if the target and center bodies don't change on repeated calls, some optimizations can be made for better performance. The `M.TrajQuery` class is provided for this use case, and can be thought of as simply a special case of `M.TrajSet` where the body and center are fixed for every call.

```
In [29]: vygrTwoQuery =  M.TrajQuery( boa,
    ...:   "Voyager 2", "Earth", "EME2000" )
In [31]: vygrTwoQuery.state( currentTime )
Out[31]:
State (km, km/sec)
'Earth' -> 'Voyager 2' in 'EME2000'
at '06-JUN-2014 19:58:35.1356 TAI'
Pos:  4.358633010242671e+09 -7.411125552099214e+09
     -1.302731854689579e+10
Vel: -2.415141211951430e+01  2.640692963340520e+00
     -1.128801136174438e+01
```

This can be useful when you are sampling states from a trajectory, for instance, to create a plot of an orbit.

Uranus Encounter

We said earlier that `M.TrajSet` and `M.CoordSet`, in their roles as manager classes, have a global view of the trajectory and coordinate systems. This high-level perspective allows them to work with the *relationships* between different bodies and frames, a capability we have so far used to get relative states between bodies. However, there are certain specific relationships between bodies and frames that can be of particular interest to an analyst. For instance, identifying the time at which two bodies achieve their closest approach (periapse) and the magnitude of that minimum distance can be an important astrodynamic metric. MONTE provides tools for searching through various relationship-spaces and identifying some of these key events. The `M.EventSpec`

set of classes allow us to define a particular event type then search through the requisite relationships to identify specific occurrences. The `M.Event` class is used to report the relevant data associated with an occurrence.

Continuing the example, we will use `M.ApsisEvent` (which is a specific type of `M.EventSpec`) to find the precise time and distance of Voyager 2's closest approach with Uranus.

```
In [6]: vygrTwoUranusQuery = M.TrajQuery( boa,
   ...:   "Voyager 2", "Uranus Barycenter", "EME2000" )
In [7]: apsisSearch = M.ApsisEvent( vygrTwoUranusQuery,
   ...:   "PERIAPSIS" )
```

`M.ApsisEvent` takes as its first argument an `M.TrajQuery` object that is configured to return the state of our target body with respect to the desired center (in this case, Voyager 2 with respect to Uranus). The second argument specifies what type of apsis we are looking for; this can be "PERIAPSIS", "APOAPSIS", or the catch-all "ANY". Once the event type is defined, the `M.ApsisEvent.search` method can be called to perform the search and locate the apses. To call this method we need to provide a time interval to search over and a search step size.

```
In [14]: searchInterval = M.TimeInterval(
    ...:   "01-JAN-1986 ET", "01-JAN-1987 ET" )
In [15]: stepSize = 60 * sec
In [16]: foundEvents = apsisSearch.search(
    ...:   searchInterval, stepSize )
```

The result of the search, which we have saved in the variable `foundEvents`, is an `M.EventSet` container class. This container has all the events found matching our specification in the search window. `M.EventSet` has a number of useful methods for sorting, filtering and returning events. In this case there should only be one event returned since there was only one closest approach of Voyager 2 to Uranus. We can read out this event by indexing into the `M.EventSet`.

```
In [17]: foundEvents.size()
Out[17]: 1

In [18]: uranusPeriapse = foundEvents[0]
In [19]: uranusPeriapse
Out[19]:
Event:
Spec : Periapsis Uranus Barycenter to Voyager 2
Type : Periapsis
Epoch: 24-JAN-1986 17:59:45.6473 ET
Value:  1.071300446056250e+05 * km
```

Another relationship which can play a significant role in deep space missions is the angular offset between the Earth-Sun line and Earth-Spacecraft line (often referred to as the Sun-Earth-Probe (SEP) angle). At low SEP values, the spacecraft appears very close to the Sun from the vantage of Earth, requiring radio transmissions from Earth to pass through the near-solar environment before reaching the spacecraft. Flight projects avoid critical mission operations during these times because the highly-charged solar atmosphere can interfere with radio signals.

We can set up an event search to find periods of low-SEP for Voyager 2, from mission start through the end of our trajectory data, using the `M.AngleEvent` event specification class.

```
In [20]: sepSearch = M.AngleEvent(boa, "Sun", "Earth"
    ...:   "Voyager 2", 12 *deg, "BELOW")
In [23]: searchWindow = traj.totalInterval("Voyager 2")
In [25]: foundEvents = sepSearch.search(searchWindow,
    ...:   1 *hour)
```

We constructed our `M.AngleEvent` by defining the Sun-Earth-Probe angle using the Sun for body one, the Earth as the vertex, and Voyager 2 as body two. Twelve degrees was set as the threshhold defining conjunction, and the "BELOW" qualifier was used to instruct the search to return times when the SEP angle was below this threshold.

The search again returned an `M.EventSet`, which we can use to get information about the number of events found and the maximum / minumum times Voyager 2 spent in conjunction.

```
In [26]: foundEvents.size()
Out[26]: 15

In [52]: foundEvents.maxInterval()
Out[52]:
Event:
. . .
Type : Angle below  1.200000000000000e+01 * deg
Begin: 28-JUN-1978 07:34:09.7021 ET
End  : 03-AUG-1978 05:22:28.3997 ET
Value:  1.199999999999977e+01 * deg

In [53]: foundEvents.minInterval()
Out[53]:
Event:
. . .
Type : Angle below  1.200000000000000e+01 * deg
Begin: 31-DEC-1992 09:35:21.3322 ET
End  : 07-JAN-1993 21:30:07.6066 ET
Value:  1.199999999999999e+01 * deg
```

We can loop through all the events found in our search using Python iterator syntax, and print out the time periods of each found low-SEP region.

```
In [56]: for event in foundEvents:
    ...:     print event.interval()
    ...:
TimeInterval(
   [ '28-JUN-1978 07:34:09.7021 ET',
     '03-AUG-1978 05:22:28.3997 ET' ],
)
TimeInterval(
   [ '29-JUL-1979 03:25:57.3664 ET',
     '31-AUG-1979 14:35:53.2033 ET' ],
)

. . .

TimeInterval(
   [ '26-DEC-1991 13:45:23.6951 ET',
     '12-JAN-1992 23:46:40.4029 ET' ],
)
TimeInterval(
   [ '31-DEC-1992 09:35:21.3322 ET',
     '07-JAN-1993 21:30:07.6066 ET' ],
)
```

As we can see, low-SEP periods occur on a near-yearly basis. This makes sense because as the Earth makes a complete revolution around the Sun, there is bound to be a period of time when the Sun falls in the line-of-sight of Voyager 2. Curiously though, the last low-SEP region found was in the winter of 1992. After this time, the Sun no longer obscures the Earth's view of Voyager 2 at all! Evidently, Voyager 2s trajectory changed in a way that disrupted this the annual low-SEP viewing geometry dynamic.

If Voyager 2 were to somehow leave the plane of the solar-system, the Earth would have a constant unobstructed view of the spacecraft permanently. We can investigate this theory by looking at the distance of Voyager 2 from the solar system ecliptic plane. We do this by setting up a trajectory query to return the state of Voyager 2 with respect to the Sun in EMO2000 coordinates (the EMO2000 coordinate frame measures Z with respect to the solar system plane). The Z-component of the position vector will then

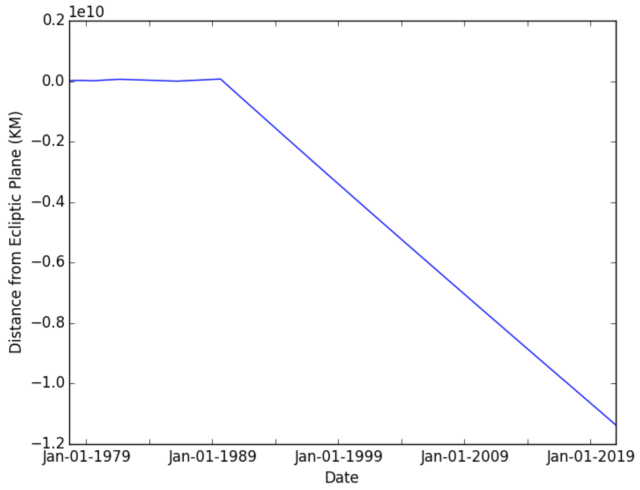

Fig. 5: *Distance in kilometers of Voyager 2 from the solar system ecliptic plane.*

yield the offset from the ecliptic plane. We will plot this distance over the course of the Voyager 2 mission to see how this distance evolves.

```
In [63]: eclipticQuery = M.TrajQuery(boa,
    ...:   "Voyager 2", "Sun", "EMO2000")
In [64]: searchWindow
Out[64]:
TimeInterval(
   [ '20-AUG-1977 15:32:32.1830 ET',
     '05-JAN-2021 00:00:00.0000 ET' ],
)

In [65]: sampleTimes = M.Epoch.range(
    ...:   '21-AUG-1977 ET', '04-JAN-2021 ET', 1 *day)
In [66]: z = []
In [67]: for time in sampleTimes:
    ...:   state = eclipticQuery.state( time )
    ...:   z.append( state.pos()[2] )
    ...:
In [68]: import mpylab
In [69]: fig, ax = mpylab.subplots()
In [70]: ax.plot( sampleTimes, z )
In [71]: ax.set_xlabel( "Date" )
In [72]: ax.set_ylabel(
    ...:   "Distance from Ecliptic Plane (Km)" )
```

The generated plot is shown in Figure 5.

It appears that something happened in 1989 to cause Voyager 2 to depart from the ecliptic plane. A quick glance at the Wikipedia page for Voyager 2 confirms this, and reveals the cause of this departure.

> *Voyager 2's closest approach to Neptune occurred on August 25, 1989 ... Since the plane of the orbit of Triton is tilted significantly with respect to the plane of the ecliptic, through mid-course corrections, Voyager 2 was directed into a path several thousand miles over the north pole of Neptune ... The net and final effect on the trajectory of Voyager 2 was to bend its trajectory south below the plane of the ecliptic by about 30 degrees.*

Conclusion

MONTE is one of the most powerful astrodynamic computing libraries in the world. It has been extensively tested and verified by flying actual spacecraft to destinations in the solar system. It is a compelling platform for anyone doing aerospace related computation, especially for those who love working with the Python language.

Acknowledgements

This work was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

References

[Moy71] T. Moyer, *Mathematical Formulation of the Double-Precision Orbit Determination Program (DPODP)*, TR 32-1527 Jet Propulsion Laboratory, Pasadena 1971.

[Moy03] T. Moyer, *Formulation for Observed and Computed Values of Deep Space Network Data Types for Navigation*, John-Wiley & Sons, Inc. Hoboken, Jew Jersey, 2003.

[Eke05] J. Ekelund, *History of the ODP at JPL*, Internal Document, Jet Propulsion Laboratory, Pasadena 2005.

[Smi16] J. Smith, *Distributed Parameter System for Optimization and Filtering in Astrodynamic Software*, 26th AAS/AIAA Spaceflight Mechanics Meeting 2016 proceedings, Napa, CA.

[Eva16] S. Evans, *MONTE: The Next Generation of Mission Design & Navigation Software*, The 6th International Conference on Astrodynamics Tools and Techniques (ICATT) proceedings 2016, Darmstadt, Germany.

The Climate Modelling Toolkit

Joy Merwin Monteiro‡*, Rodrigo Caballero‡

Abstract—The Climate Modelling Toolkit (CliMT) is a Python-based software component toolkit providing a flexible problem-solving environment for climate science problems. It aims to simplify the development of models of complexity 'appropriate' to the scientific question at hand. This aim is achieved by providing Python-level access to components commonly used in climate models (such as radiative transfer models and dynamical cores) and using the expressive data structures available in Python to access and combine these components. This paper describes the motivation behind developing CliMT, and serves as an introduction to interested users and developers.

Index Terms—Climate Modelling, Hierarchical Models

Introduction

Climate models are numerical representations of the climate system consisting of ocean, land and atmosphere. They have become an important aspect of climate science as they provide a virtual laboratory in which to perform experiments and gain a deeper understanding of the climate system. Climate models can be conceived as a combination of two distinct parts: One, called the "dynamics", is code which numerically integrates the equations of motions of a fluid. The other, called the "physics" is code which approximates various processes considered important for the evolution of the atmospheric/oceanic fluid, including radiation, moist convection and turbulence. Some of these processes, such as convection and turbulence should ideally simulated by the dynamics, but the coarse resolution of typical climate models and limitations of computational resources lead to their being approximated as physics components.

In an influential essay, Isaac Held made the case for studying the climate system using a hierarchy of models, in a manner similar to the hierarchy of model organisms used by evolutionary biologists [Hel05]. The essay argued that such a hierarchy would not only help in our understanding of the climate system, but would also help in interpreting results obtained from more complex models and even aid in improving them. A qualitative description of the climate model ecosystem is shown in Fig. 1. On the dynamics axis, they range from models which represent the atmosphere as a single vertical column to a full turbulent, three dimensional flow. On the physics axis, they range from models which represent radiation or turbulence using ten lines of code to those whose representation of the physics run into thousands of lines.

* *Corresponding author: joy.merwin@gmail.com*
‡ *MISU, Stockholm University*

Over the past few decades, efforts have been made to develop such a hierarchy of models (a significant fraction of which has been, not surprisingly, by Isaac's students and collaborators) which have had a positive impact on our understanding of the climate system and the general circulation of the atmosphere [HK81], [HS94], [NZ00], [Sch04], [FHZG06], [MFCE07], [CPM08], [MPFC09]. Note that we restrict our focus here only on numerical models of the climate system, excluding many influential theoretical models such as [HH80] and models of phenomena such as the Madden-Julian Oscillation (MJO) or tropical cyclones.

However, the scale of these efforts has not kept pace with the increasing complexity of full scale general circulation models (GCMs) which are on the threshold of cloud-scale (~1 km horizontal resolution) simulations of the entire atmosphere. One of the primary reasons, we believe, is that significant effort is required to build models which represent even the basic features of the atmospheric general circulation. Existing frameworks to develop such models like the Flexible Modelling System (FMS, `http://www.gfdl.noaa.gov/fms`) and the MITgcm (`http://mitgcm.org/`) are typically written in Fortran and the effort to set up a model beyond those already provided as examples can, in our experience, be quite discouraging for new users who lack a strong background in Fortran and programming.

In this paper we introduce the Climate Modelling Toolkit (CliMT, pronounced "Klimt"), which attempts to reduce this barrier to developing simplified models of the atmosphere. It is similar in spirit to the above mentioned frameworks, with the following distinctions:

- Configuration and execution of models is much simpler and done in the same script, making repeated simulations less error prone
- New components can be added with minimal infrastructure code requirements, and does not require recompilation of the entire codebase
- Object-oriented design makes program flow both intuitive and less prone to error
- Allows for incremental development: proof of concept development in pure Python and production code in another language (Cython, C, Fortran)

CliMT is currently not capable of parallel execution, and thus is mainly useful for 1 and 2-dimensional climate models. Despite these limitations, CliMT is used by around 10 research groups around the world (based on user queries/feedback) for research [CPM08], [CH13], [RBSB10] and pedagogy [Pie10]. In the following sections, we describe the basic building blocks of CliMT, their usage, and how new components can be added. We end with a roadmap towards version 1.0 of CliMT, which should

Physical Process	Components
Convection	Zhang-McFarlane
	Emanuel
	Emanuel hard adjustment
	Simplified Betts-Miller
Dynamics	Axisymmetric dynamics
	Two column dynamics
Ocean	Slab Ocean
Radiation	Community Atmosphere Model (CAM) 3
	CCM 3
	Chou
	Grey Gas
	Rapid Radiative Transfer Model (RRTM)
	Insolation
	Absorption by ozone
Turbulence	CCM3
	Simple (diffusive)
Thermodynamics	routines for calculating thermodynamic quantities

TABLE 1: *Components available currently in CliMT.*

see CliMT working as a fully parallel, moist GCM capable of simulating a realistic climate.

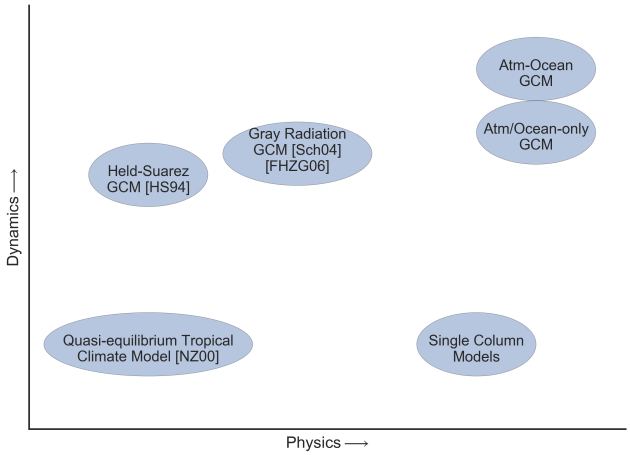

Fig. 1: *A qualitative depiction of the climate model hierarchy. The complexity of the dynamics and the physics models increases along the direction of the arrows. This is merely an indicative representation, and is not meant to be exhaustive.*

CliMT: best of both worlds

CliMT combines the elegance and clarity of the Python language with the efficiency of Fortran libraries. Users interact with CliMT in a pythonic way, using high-level data structures like dictionaries and lists, and the numerical computations are done by optimised (and tested) Fortran code extracted from state-of-the-art climate models. Currently, `f2py` is used to convert Fortran code to a library that can be imported into Python. Table 1 lists the physical processes that can be currently simulated using CliMT and the options available to represent each physical process.

The initialization of the components and the execution of the resulting model is handled in the same script, which makes the parameters and assumptions underlying the model explicit. This makes interpreting the results of the simulation easier. Given that model initialization, execution and data analysis can be performed from within a single IPython notebook, this makes model results and the resulting scientific results reproducible as well. CliMT also enables users to study the effects of changing physical parameterizations and dynamical cores on the simulated climate, something that is difficult to do in other idealised modelling frameworks.

Architecture

CliMT, in a broad sense, is a library which enables numerical representations of different processes in the climate system to be linked together in an intuitive manner. While it provides a leapfrog integrator (a second order method for numerical integration common to many climate models) to step the model forward in time, it does not provide routines to calculate gradients or spectral coefficients. All components in CliMT are either written from scratch or extracted from larger climate models (especially radiative transfer models). There is no facility to update the underlying Fortran/C code itself if the original code is updated. It is assumed that each component will implement any numerical methods that it requires. While this may lead to some code duplication, it allows for a loose coupling between the various components. This allows development of new components without recompilation of the entire codebase.

When a component is instantiated, CliMT queries the component to find out which variables the component affects. For instance, a convection component will affect the specific humidity and the temperature variables. It creates a Numpy array of the appropriate dimensions for each such variable. If multiple components affect the same variable, only one such array is created. During execution, it collects the time tendency* terms from each component (in the form of a Numpy array), sums them together and uses the resulting cumulative tendency to step the model forward in time. Currently, it is assumed that all components share a common grid, i.e, all arrays representing tendency terms have the same shape, and represent the same location in three dimensional space. As is commonly the case in climate models, the spatial coordinates are in latitude-longitude-pressure space, and CliMT does a sanity check to ensure that all components have the same spatial representation (i.e, tendency arrays expected from each component has the same shape).

To summarize, each component (encapsulated in the `Component` class) provides time tendency terms to the main execution loop, and the model is stepped forward in time by integrating these tendencies using the leapfrog integrator. Optionally, the model state is displayed using a wrapper over `matplotlib` and written to disk using the `netCDF4` library. Since the model state variables are Numpy arrays, they can be easily accessed by external Python libraries for online processing or any other purpose.

Combining multiple `Component` objects is made possible using the `Federation` class. Combining two or more desired `Component` objects in a `Federation` results in a climate model of appropriate complexity.

The `Component` and `Federation` classes are the interface between the end-user and CliMT, with all other classes being used internally by these two classes.

*. A time tendency term at time t_1 is the incremental value of a variable to be added to obtain that variable's value at time t_2 , where t_2 is the time instant succeeding t_1.

Component

A Component class is the fundamental abstraction in CliMT. It encapsulates the behavior of a component that takes certain inputs and provides certain tendencies as output. Each Component object has (among others) the following members which are specified by the developer:

- Prognostic
- Diagnostic
- Fixed
- FromExtension
- ToExtension

These members are lists whose elements are one of many predefined field names (available in the State class) relevant to climate science applications. For example, if Component.Prognostic = ['U', 'V', 'theta'], then the component represents a model which can forecast the future state of the wind along longitude, wind along latitude and the potential temperature, respectively. The Diagnostic list contains those fields which the component calculates using the prognostic fields, and the Fixed list contains those fields which are left unchanged by the component. The ToExtension list indicates which fields are required by the component to forecast the future state, and the FromExtension list indicates which fields are returned by the component. Typically, the FromExtension list contains the name of fields with an Inc suffix, indicating that the component returns increments only, which are to be stepped forward in time. The term Extension refers to the compiled Fortran/C library which does the actual computation. Each Component also keeps track of the time step dt taken during each integration (normally decided by stability constraints), and the time elapsed from the beginning of the integration.

Component has two main methods: compute and step. The compute method calls the compiled Fortran/C code and retrieves the increments and diagnostic fields and stores them internally. compute takes an optional boolean argument ForcedCompute. If ForcedCompute is true, then the tendency terms are always calculated. If it is false (the default), then the tendencies are calculated only if the elapsed time is at least dt greater than the previous time at which the tendencies were calculated. Such behavior is required when combining two components which operate on very different time scales, such as convection (time scale of hours) and radiation (time scale of days). compute is also invoked by simply calling the object.

The step method steps the component forward in time by taking the increments calculated in compute and passing them on to the leapfrog integrator (available in the infrastructure code, not in each individual component) to get future values of the fields. step internally calls compute, so the user needs only to call step. step accepts two optional arguments Inc and RunLength. Inc which is a dictionary whose keys are some or all of the elements in ToExtension, and the corresponding values are additional tendency terms calculated outside the component. These increments are added to the internally computed tendency terms before calling the integrator. Runlength decides how many seconds forward in time the component is stepped forward. If RunLength is a positive integer, then the component is stepped forward in time RunLength * dt seconds. If it is a positive floating point number, then the component is stepped foward in time RunLength seconds.

All parameters required by any Component are passed as a dictionary during object instantiation. This includes initial values of the fields integrated by the Component. If no initial values are supplied, the fields are initialized as zeroed NumPy arrays of the appropriate shape. An example which uses the CAM radiative transfer model to compute the radiative tendencies is shown below (also available in the source code itself):

```
import numpy as np
import climt

#--- instantiate radiation module
r = climt.radiation(scheme='cam3')

#--- initialise T,q
# Surface temperature
Ts = 273.15 + 30.
# Stratospheric temp
Tst = 273.15 - 80.
# Surface pressure
ps = 1000.
# Equispaced pressure levels
p = ( np.arange(r.nlev)+ 0.5 )/r.nlev * ps
# Return moist adiabat with 70% rel hum
(T,q) = climt.thermodyn.moistadiabat(p, Ts, Tst, 1.)

# Set values for cloud fraction and
#cloud liquid water path
cldf = q*0.
clwp = q*0.
cldf[len(cldf)/3] = 0.5
clwp[len(cldf)/3] = 100.

#--- compute radiative fluxes and heating rates
r(p=p, ps=ps, T=T, Ts=Ts, q=q, cldf=cldf, clwp=clwp)
```

In the above code, the computed outputs can be accessed by treating r as a dictionary: the shortwave flux at the top of the atmosphere is available at r['SwToa'], for example.

Federation

Federation is a subclass of Component which is instantiated by providing two or more Component objects as arguments. It provides the same interface as Component, and is the abstraction of a climate model with multiple interacting components. On instantiation, Federation does a few sanity checks to ensure consistency of dimensions between its member Components. As in Component, integrating the Federation forward in time is simply achieved by calling step. An example which computes the radiative convective equilibrium in a column of the atmosphere is given below:

```
import climt
import numpy as np

# Some code initialising kwargs
...

# -- Instantiate components and federation

#Radiation is called only once every
#50 timesteps, since it is a slow process.
rad = climt.radiation(
        UpdateFreq=kwargs['dt']*50,
        scheme='cam3')

#Convection consumes the instability
#produced by radiation
con = climt.convection(
        scheme='emanuel')

# turbulence facilitates the exchange
# of water vapour and momentum between
```

```
# the ocean and the atmospheric column
dif = climt.turbulence()

#Ocean provides a source of water vapour
oce = climt.ocean()

#Instantiate the federation
fed = climt.federation(dif, rad, oce,
            con, **kwargs)

# Main timestepping loop
for i in range(1000):
    # The following code adds a uniform
    # 1 K/day cooling rate to
    # the internally-computed tendencies
    dT= np.array([[-1./86400.*kwargs['dt']*
            2.*np.ones(rad.nlev)]]).transpose()

    fed.step(Inc={'T':dT})
```

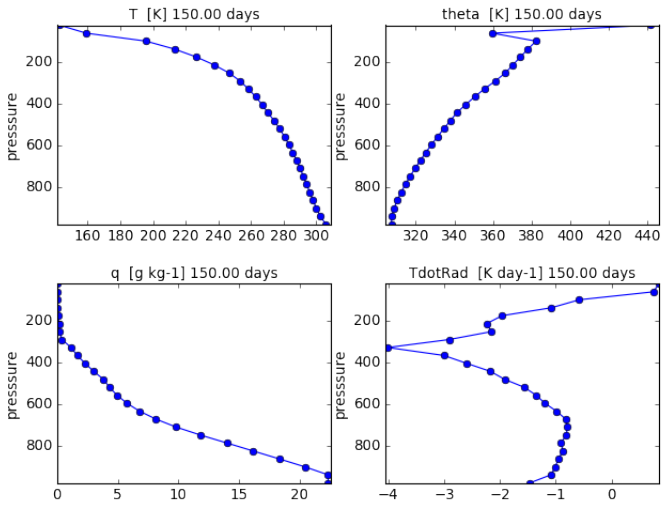

Fig. 2: *The displayed output from a one dimensional (vertical) radiative-convective simulation on day 150. The fields are updated in real time during the simulation. The panels display (clockwise from top left): Temperature, Potential Temperature, radiative heating and specific humidity respectively. The y axis is height measured in pressure and has units of millibar (100 Pascals = 1 millibar). As expected from theory and observations, the temperature decreases almost linearly in the lower levels of the column.*

Here, the radiative code has an `UpdateFreq` value that is 50 times the actual timestep of the federation. As mentioned before, this feature facilitates coupling of components whose characteristic time scales are very different from each other without increasing the computational load during the simulation. Notice also the external tendency term `dT` passed on to `fed` in the `step` method. The output fields are again accessed by treating `fed` as a dictionary. Figure 2 shows the typical output from a CliMT radiative-convective simulation; Display and I/O is discussed in the next section.

Software Layout and Documentation

CliMT maintains the infrastructure code and the actual component code in separate folders. The `src` directory contains the component code whereas `lib/climt` contains the infrastructure code. The main infrastructure code resides in `{component,federation,state,grid}.py`. The various physical processes are accessible from appropriately named files in `lib/climt` (e.g, `convection.py`). These files implement the `Component` class and act as an interface to the underlying Fortran code. Note that there is no restriction on the language in which extensions are written. All the physical variables that CliMT recognises are listed in `state.py`. While all files themselves have detailed inline documentation, there is currently no automated system in place to build a module reference based on these comments. Querying an object in an IPython environment is currently the best way of accessing the documentation, as demonstrated in Fig. 3. Addition of a new module would require copying the extension code to `src/`, adding a reference to it in the appropriate physical process file (e.g, a new dynamical core would be included in `dynamics.py`), and adding a reference in `setup.py` to enable building and installation.

```
In [1]: from climt.dynamics import dynamics
Using netCDF4 interface for IO

In [2]: ?dynamics
Init signature: dynamics(self, scheme='axisymmetric', **kwargs)
Docstring:
Interface to atmospheric dynamical cores.

* Instantiation:

  x=climt.dynamics( <args> )

  where <args> are the following OPTIONAL arguments:
  Name            Dims  Meaning                  Units     Default         Notes
  scheme             0  Dynamical core          (string) 'axisymmetric'  Choices are: 'axisymmetric'
  T                1-3  Temperature                K          283.15
  q                1-3  Specific humidity         g/kg       1.e-5
  U                1-3  Zonal wind               m/s        0.
  V                1-3  Meridional wind           m/s        0.

* Usage:
  Call instance directly to compute dynamical tendencies.

  x( <args> )

  where <args> are as above.

* Output (accessible as x.swflx etc.):
  Name       Meaning                          Units   Notes
  Tdot       Turbulent heating rate           K s-1
  qdot       Turbulent humidification rate    g/kg s-1
  Udot       Turbulent drag                   m s-2
  Vdot       Turbulent drag                   m s-2
  SrfSenFlx  Surface sens. heat flux          W m-2
  SrfLatFlx  Surface lat. heat flux           W m-2
  taux       Surface stress                   Pa
  tauy       Surface stress                   Pa
File:           /usr/local/lib/python2.7/dist-packages/climt/dynamics.py
Type:           classobj

In [3]:
```

Fig. 3: *Accessing documentation for the dynamics class in an IPython prompt.*

Monitoring fields and I/O

CliMT also provides for real time display (monitoring) of the simulated fields. Currently, up to four fields can be monitored. Monitoring is activated by providing an additional argument during component instantiation called `MonitorFields`. `MonitorFields` is a list of up to four fields that are part of the simulation. If the field is three dimensional, the zonal average (average along longitude) is displayed. The frequency at which the display is refreshed is decided by the `MonitorFreq` argument.

CliMT can read intial conditions from the file whose name is specified in the `RestartFile` argument. The output is written to the file whose name is specified in the `OutputFile` argument. If `RestartFile` and `OutputFile` are the same, then the data is appended to `OutputFile`. The last time slice stored in `RestartFile` is used to initialize the model. If some fields are missing in `RestartFile`, they are initialized to default (zero) values.

The fields written to the output file are specified in the `OutputFields` argument. If `OutputFields` is not specified, all fields are written to file. `OutputFreq` is an optional argument

which specifies the time between writing data to file. If it is not specified, the output is stored once every model day.

Developing new Components

CliMT requires a single point of entry into the Fortran/C code to be provided by each `Component`: the `driver` method. The `driver` method takes as input NumPy arrays representing the fields required to calculate the tendency terms. The order in which the fields are input is represented by the `ToExtension` list in the Component. The output of the driver is a list of NumPy arrays ordered in the same way as the `FromExtension` list. The translation between NumPy arrays and the Fortran code is currently done automatically by `f2py` generated code. The Fortran/C extension module itself is stored in `Component.Extension` and an optional name is provided in `Component.Name`. `Component.Required` is a list of those fields which are essential for the component to calculate tendencies. These variables along with `Prognostic`, `Diagnostic` and `Fixed` lists (which were previously discussed) enable CliMT to interface with a new component.

We note that CliMT expects the tendency terms to be premultiplied by `dt`, i.e, the units of the fields returned by `driver` is expected to be the same as the units of the prognostic fields. The integrator does not multiply the tendency terms by `dt`, as is normally the case.

Current Development: towards CliMT 1.0

The space occupied by CliMT in the climate model hierarchy is shown in Fig. 4. It is currently capable of simulating relatively simple (1 and 2 dimensional) dynamics and quite sophisticated physical processes. Moving forward, we hope fulfill the vision of using CliMT as a full fledged moist idealized GCM. As a first step, we have integrated a dynamical core adapted from the Global Forecast System (GFS). Together with this, we have added a new Held-Suarez module which provides the Held-Suarez forcing terms for a 3-d atmosphere. A working example of the benchmark is now available from a development fork (available at `https://github.com/JoyMonteiro/CliMT/lib/examples`). Figure 5 shows the mean wind along longitudes ("zonal" wind) simulated by the model. It shows most of the important aspects of the mean circulation in the earth's atmosphere: strong westerly jet streams around 30 degrees N/S and easterly winds near the surface and the top of the atmosphere in the tropics.

Many changes were incorporated enroute this integration. The dynamical core is the first component of CliMT that interfaces with the Fortran library using Cython and the ISO_C_Binding module introduced in Fortran 2003. This will be used as a template to eventually move all components to a Cython interface: `f2py` does not seem to be actively developed anymore, and currently cannot interface with code that includes compound data structures, like the FMS dynamical cores. Therefore, we expect the Cython-ISO_C_Binding combination to enable CliMT to use a wider range of libraries.

A new feature in CliMT 1.0 will be to allow components to use an internal integrator and not the default leapfrog available in CliMT. This is useful since components such as the 3-D dynamical core already include non-trivial implementations of numerical integrators which will have to be reimplemented in CliMT to ensure stable integrations. Moreover, it is unlikely that atmosphere

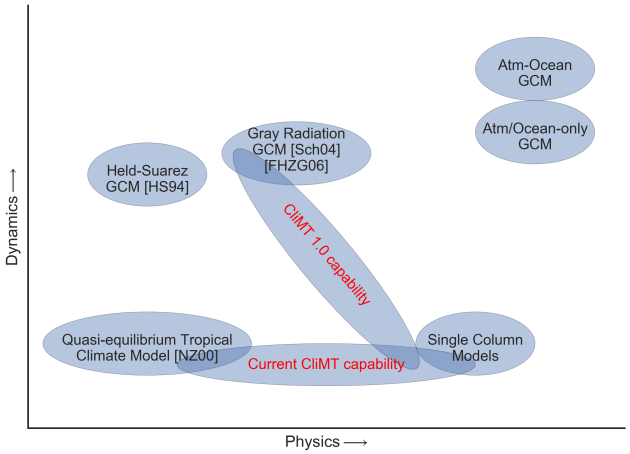

Fig. 4: *A look at the current capability and future directions for CliMT development in context of the model hierarchy*

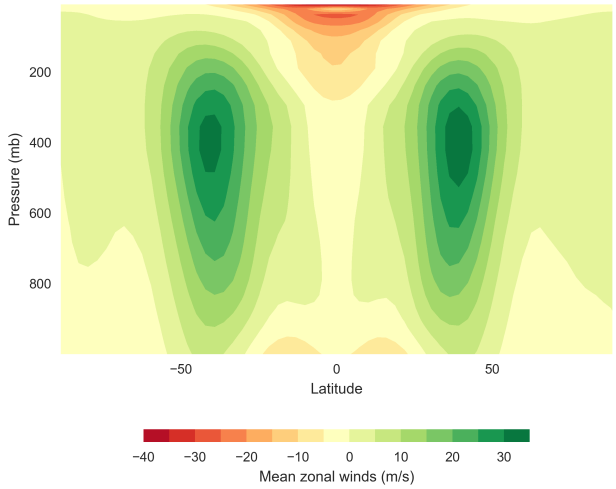

Fig. 5: *The mean wind along longitudes in the Held-Suarez simulation. The mean is over 1000 days and over all longitudes (a "zonal" mean). The y-axis has units of millibar (= 100 Pa). It compares well with the simulated winds in [HS94] (see Fig. 2 in their paper)*

and ocean models use similar numerical algorithms. Since the focus of CliMT is on the infrastructure and not the numerics, this feature enables rapid addition of new components into CliMT without substantially changing its basic structure. This feature and other enhancements will be described in detail in a forthcoming paper accompanying the release of CliMT 1.0.

Next, we intend to interface the dynamical core with the grey gas radiation module to enable CliMT to generate a realistic general circulation without using the Held-Suarez forcing. Issues we intend to address in the future include:

- scalability by making CliMT MPI and/or OpenMP-aware
- More systematic testing architecture
- A full user manual and IPython notebook examples

With these additions, we hope CliMT will be the framework of choice for a wide audience, from undergraduates to scientists to explore questions in climate science.

Acknowledgements

This work is supported by funding from the Swedish e-Science Research Centre (`http://www.e-science.se/`).

REFERENCES

[CH13] R. Caballero and M. Huber. State-dependent climate sensitivity in past warm climates and its implications for future climate projections. *Proceedings of the National Academy of Sciences*, 110(35):14162–14167, August 2013.

[CPM08] R. Caballero, R. T. Pierrehumbert, and J. L. Mitchell. Axisymmetric, nearly inviscid circulations in non-condensing radiative-convective atmospheres. *Quarterly Journal of the Royal Meteorological Society*, 134(634):1269–1285, July 2008.

[FHZG06] D. M. W. Frierson, I. M. Held, and P. Zurita-Gotor. A Gray-Radiation Aquaplanet Moist GCM. Part I: Static Stability and Eddy Scale. *Journal of the Atmospheric Sciences*, 63(10):2548–2566, October 2006.

[Hel05] I. M. Held. The Gap between Simulation and Understanding in Climate Modeling. *Bulletin of the American Meteorological Society*, 86(11):1609–1614, November 2005.

[HH80] I. M. Held and A. Y. Hou. Nonlinear Axially Symmetric Circulations in a Nearly Inviscid Atmosphere. *Journal of the Atmospheric Sciences*, 37(3):515–533, March 1980.

[HK81] B. J. Hoskins and D. J. Karoly. The steady linear response of a spherical atmosphere to thermal and orographic forcing. *Journal of the Atmospheric Sciences*, 38(6):1179–1196, 1981.

[HS94] I. M. Held and M. J. Suarez. A Proposal for the Intercomparison of the Dynamical Cores of Atmospheric General Circulation Models. 75(10):1825–1830, October 1994.

[MFCE07] J. Marshall, D. Ferreira, J-M. Campin, and D. Enderton. Mean Climate and Variability of the Atmosphere and Ocean on an Aquaplanet. *Journal of the Atmospheric Sciences*, 64(12):4270–4286, December 2007.

[MPFC09] J. L. Mitchell, R. T. Pierrehumbert, D. M.W. Frierson, and R. Caballero. The impact of methane thermodynamics on seasonal convection and circulation in a model Titan atmosphere. *Icarus*, 203(1):250–264, September 2009.

[NZ00] J. D. Neelin and N. Zeng. A Quasi-Equilibrium Tropical Circulation Model-Formulation. *Journal of the atmospheric sciences*, 57(11):1741–1766, 2000.

[Pie10] R. T. Pierrehumbert. *Principles of planetary climate*. Cambridge University Press, 2010.

[RBSB10] M. T. Rosing, D. K. Bird, N. H. Sleep, and C. J. Bjerrum. No climate paradox under the faint early Sun. *Nature*, 464(7289):744–747, April 2010.

[Sch04] T. Schneider. The Tropopause and the Thermal Stratification in the Extratropics of a Dry Atmosphere. *Journal of the Atmospheric Sciences*, 61(12):1317–1340, June 2004.

Tell Me Something I Don't Know: Analyzing OkCupid Profiles

Juan Shishido[‡*], Jaya Narasimhan[¶†], Matar Haller[§†]

https://youtu.be/dtgmMj8W298

Abstract—In this paper, we present an analysis of 59,000 OkCupid user profiles that examines online self-presentation by combining natural language processing (NLP) with machine learning. We analyze word usage patterns by self-reported sex and drug usage status. In doing so, we review standard NLP techniques, cover several ways to represent text data, and explain topic modeling. We find that individuals in particular demographic groups self-present in consistent ways. Our results also suggest that users may unintentionally reveal demographic attributes in their online profiles.

Index Terms—natural language processing, machine learning, supervised learning, unsupervised learning, topic modeling, okcupid, online dating

Introduction

Online dating has become a common and acceptable way of finding mates. In the United States, 41 percent of adults know someone who uses online dating, 29 percent know someone who has met a partner this way, and 59 percent believe online dating is a good way to meet people [Pew16]. In 2015, online dating sites or mobile dating apps were used by 27 percent of 18-24 year olds, 22 percent of 25-34 year olds, 21 percent of 35-44 year olds, 13 percent of 45-54 year olds, and 12 percent of 55-64 year olds [Pew16]. Relative to 2013, usage across every age group, except 25-34 year olds, increased. Given the popularity of online dating, the way that people self-present online has broad implications for the relationships they pursue.

Previous studies suggest that the free-text portion of online dating profiles is an important factor (after photographs) for assessing attractiveness [Fio08]. The principle of homophily posits that people tend to associate and bond with individuals who are similar to themselves and that this strongly structures social networks, most prominently by race and ethnicity [McP01]. Perhaps not surprisingly, research suggests that homophily extends to online dating, with people seeking mates similar to themselves [Fio05]. However, it remains unclear whether people within particular demographic groups, such as sex or ethnicity, self-present in similar ways when searching for a mate online.

* *Corresponding author: juanshishido@berkeley.edu*
‡ *School of Information, University of California, Berkeley*
† *These authors contributed equally.*
¶ *Department of Electrical Engineering and Computer Science, University of California, Berkeley*
§ *Helen Wills Neuroscience Institute, University of California, Berkeley*

In this paper, we analyze demographic trends in online self-presentation. Specifically, we focus on whether people signal demographic characteristics through the way they present themselves online. We extend previous natural language processing analyses of online dating [Nag09] by using a much larger sample[1] and by combining NLP with supervised and unsupervised machine learning. We leverage multiple approaches including clustering and topic modeling as well as feature selection and modeling strategies. By exploring the relationships between free-text self-descriptions and demographics, we discover that we can predict users' demographic makeup and also find some unexpected insights into unintentional signaling of demographic characteristics.

Code and data for this work are available in our `okcupid` GitHub repository[2]. A Jupyter notebook with the analysis results is also available[3].

Data

Description

Profile information[4] was available for 59,946 OkCupid users that were members as of 06/26/2012, lived within 25 miles of San Francisco, had been active in the previous year, and had at least one photo in their profile [Wet15]. The data set contained free-text responses to 10 essay prompts as well as the following user characteristics: age, body type, diet, drinking status, drug usage status, education level, ethnicity, height, income, job type, location, number of children, sexual orientation, attitude toward pets, religion, sex, astrological sign, smoking status, number of language spoken, and relationship status.

This public[5] data set was selected for its diverse set of essay prompts and availability of detailed user characteristics, which enabled us to examine the connection between online self-presentation and demographics. This data set has previously been used to demonstrate the basics of text analysis as well as how to fit a simple logistic regression model to predict sex using only height

1. [Nag09]'s uses a sample of 1,000 individuals.
2. https://github.com/juanshishido/okcupid.
3. https://github.com/juanshishido/okcupid/blob/master/OkNLP-paper.ipynb
4. https://github.com/rudeboybert/JSE_OkCupid. Our original data source was Everett Wetchler's `okcupid` repository (https://github.com/everett-wetchler/okcupid). However, after commit `0d62e62`, in which the data was "fully anonimized" to exclude essays, we switched to Kim's repository. Kim uses the original Wetchler data.

[Kim15]. The present study extends previous work by analyzing additional features and by introducing novel analyses.

Preprocessing

Line break characters, URLs, and HTML tags were removed from the essay text. Multiple periods, dashes, and white spaces were replaced by single instances, and all text was converted to lowercase. Essays were segmented, first into sentences and then into individual terms, using spaCy's [Hon16][6] default tokenizer, which is well suited for online communication as it maintains emoticons as discrete tokens. This allowed us to differentiate between the syntactic way that special characters are traditionally used and the meaning that's conveyed when they are used in particular combinations. Punctuation was removed *after* the text was tokenized[7]. Finally, users who wrote less than five words for a given essay were removed from the analysis.

In order to reduce the number of categories, we combined drug usage status levels. Specifically, users who responded "sometimes" or "often" were grouped into a "yes" category. Individuals who answered "never" were assigned to the "no" group and we created an "unknown" category for users who did not answer.

Methods

Term Frequency-Inverse Document Frequency

Machine learning tasks require numerical inputs. There are several ways to represent text as numerical feature vectors. Features typically correspond to distinct tokens or to sequences of adjacent tokens. A token is a series of characters, such as a word, that is treated as a distinct unit [Bir10].

One way to represent a corpus, or collection of text documents, is as a matrix of token counts. This weights terms by their absolute frequencies. Often, highly-weighted terms, such as "a" or "the," are not informative, so token counts are weighted using term frequency-inverse document frequency (tf-idf).

Tf-idf is the product of the term frequency and the inverse document frequency. The term frequency refers to the *relative* frequency of term t in document d. The inverse document frequency is the log of the total number of documents N to the number of documents that contain term t.

Log-Odds-Ratio

One metric for comparing word usage across groups is to calculate the log-odds-ratio. The odds for word w in the usage of group g are defined as $O_{iw} = \frac{f_{iw}}{(1-f_{iw})}$ where f_{iw} is the frequency count of word w normalized by total count of words used by group i. If a word is used only by one group, its log-odds-ratio is infinite. Therefore, a constant is added to each frequency when calculating the odds. The log of the ratio of the adjusted odds between groups can then be used to compare word usage across groups.

Non-negative Matrix Factorization

For document clustering, the document corpus is projected onto a k-dimensional semantic space, with each axis corresponding to a particular topic and each document being represented as a linear combination of those topics [Xu_03]. Methods such as latent semantic indexing require the derived latent semantic space to be orthogonal, so this class of methods does not work well when corpus topics overlap, as is often the case. Conversely, non-negative matrix factorization (NMF) does not require the latent semantic space to be orthogonal, and therefore is able to find directions for related or overlapping topics.

NMF was applied to each essay of interest using scikit-learn [Ped11][8], which uses the coordinate descent solver. NMF utilizes document frequency counts, so the tf-idf matrix for unigrams, bigrams, and trigrams was calculated, while limiting tokens to those appearing in at least 0.5 percent of the documents. NMF was calculated with k dimensions, which factorized the tf-idf matrix into two matrices, W and H. The dimensions were `n_samples x k` and `k x n_features` for W and H, respectively. Group descriptions were given by top-ranked terms in the columns of H. Document membership weights were given by the rows of W. The maximum value in each row of W determined essay group membership.

Permutation Testing

Permutation tests provide an exact sampling distribution of a test statistic under the null hypothesis [Ger12] by computing the test statistic for every manner by which labels can be associated with the observed data. In practice, permutations are rarely ever completely enumerated. Instead, the sampling distribution is approximated by randomly shuffling the labels P times.

The likelihood of the observed test statistic is determined as the proportion of times that the absolute value of the permuted test statistics are greater than or equal to the absolute value of the observed test statistic. This is the p-value for a two-tailed hypothesis. Permutation-based methods can be used to compare two samples or to assess the performance of classifiers [Oja10].

There are several advantages to using randomization to make inferences as opposed to parametric methods. Permutation tests do not assume normality, do not require large samples, and "can be applied to all sorts of outcomes, including counts, durations, or ranks" [Ger12].

Approach

Our analyses focused on two demographic dimensions — sex and drug usage — and on two essays — "My self summary" and "Favorite books, movies, shows, music, food." These essays were selected because they were answered by most users. "The most private thing I am willing to admit" prompt, for example, was ignored by 32 percent of users.

We began by exploring the lexical features of the text as a way to determine whether there were differences in writing styles by demographic group. We considered essay length, the use of profanity and slang terms, and part-of-speech usage.

Essay length was determined based on the tokenized essays. A list of profane words was obtained from the "Comprehensive Perl Archive Network" website. Slang terms include words such as "dough," which refers to money, and acronyms like "LOL." These terms come from the Wiktionary Category:Slang page[9]. Note that there is overlap between the profane and slang lists.

5. As authorized by OkCupid president and co-founder Christian Rudder [Kim15].

6. We used version 0.101.0. GitHub, 10 May 2016. https://github.com/spacy-io/spaCy/releases/tag/0.101.0.

7. Punctuation is needed for the sentence tokenizer and sentences are important for the part-of-speech tagging.

8. We used version 0.17.1. GitHub, 18 Feb 2016. https://github.com/scikit-learn/scikit-learn/releases/tag/0.17.1-1. This is particularly important for NMF as the coordinate descent solver is the default as of 0.17.0. Using the deprecated projected gradient solver will lead to different results.

9. https://simple.wiktionary.org/wiki/Category:Slang.

Each token in the corpus was associated with a lexical category using spaCy's part-of-speech tagger. spaCy supports 19 coarse-grained tags[10] that expand upon Petrov, Das, and McDonald's universal part-of-speech tagset [Pet11].

Differences in lexical features by demographic were analyzed using permutation testing. We first compared average essay length by sex. Next, we examined whether the proportion of females using profanity was different than the proportion of males using such terms. The same was done for slang words. Finally, we compared the average proportion of adjectives, nouns, and verbs and identified the most distinctive terms in each lexical category by sex using the smoothed log-odds-ratio, which accounts for variance.

We also analyzed text semantics by transforming the corpus into a tf-idf matrix using spaCy's default tokenizer. We chose to include unigrams, bigrams, and trigrams[11]. Stop words[12] and terms that appeared in less than 0.5 percent of documents were removed. Stemming, the process of removing word affixes, was not performed. This resulted in a vocabulary size of 2,058 for the self-summaries essay and 2,898 for the favorites essay.

Non-negative matrix factorization was used to identify latent structure in the text. This structure represented "topics" or "clusters" which were described by particular tokens. In order to determine whether particular demographic groups were more likely to write about certain topics, the relative distribution of users over topics was plotted. In cases where we were able to create superordinate groupings from NMF topics — for example, by combining semantically similar clusters — we used the log-odds-ratio to find their distinctive tokens.

Based on our findings, we decided to fit a logistic regression model to predict drug usage status.

Results

In this section, we describe our lexical- and semantic-based findings.

We first compared lexical-based characteristics on the self-summary text by sex. Our sample included 21,321 females and 31,637 males[13]. On average, females wrote significantly longer essays than males (150 terms compared to 139, $p < 0.001$).

Next, we compared the proportion of users who utilized profanity and slang. Profanity was rarely used in the self-summary essay. Overall, only 6 percent of users included profane terms in their self-descriptions. The difference by sex was not statistically significant (5.8% of females versus 6.1% of males, $p = 0.14$).

Not surprisingly, slang was much more prevalent than profanity. 56 percent of users used some form of slang in their self-summary essays and females used slang at a significantly lower rate than males (54% versus 57%, $p < 0.001$).

To compare part-of-speech usage, we first associated part-of-speech tags with every token in the self-summary corpus. This resulted in counts by user and part-of-speech. Each user's counts were then normalized by the user's essay length to account for essay length differences between users. Of the 19 possible part-of-speech tags, we focused on adjectives, nouns, and verbs. The proportions of part-of-speech terms used is shown in Table 1.

Part-of-Speech	Female	Male
Adjectives ⋆⋆	10.61%	10.16%
Nouns ⋆⋆	18.65%	18.86%
Verbs	18.28%	18.27%

TABLE 1: *Proportion of part-of-speech terms used, by sex. Asterisks (⋆⋆) denote statistically significant differences at the 0.001 level.*

Part-of-Speech	Female	Male
Adjectives	independent sweet my sassy silly happy warm favorite girly fabulous	nice cool its that few interesting martial most masculine more
Nouns	girl family who yoga men gal heels love dancing friends	guy computer engineer guitar sports software women video technology geek
Verbs	love am laugh laughing dancing adore loving dance appreciate being	m was play playing laid 'll working hit moved been

TABLE 2: *The 10 most-distinctive adjective, noun, and verb tokens , by sex.*

Females used significantly more adjectives than males, while males used significantly more nouns than females ($p < 0.001$ for both). There was no difference in verb usage between the sexes ($p = 0.91$).

In addition to part-of-speech usage, we explored specific terms associated with parts-of-speech that were distinctive to a particular sex. We did this using the log-odds-ratio. Table 2 summarizes this, below.

Distinctly-female adjectives are mostly descriptive. Males, on the other hand, use more quantity-based and demonstrative adjectives. For nouns, females focus on relationship- and experience-based terms while males write about work, sports, and technology. (Note that `m` corresponds to the contracted form of "am" when "Im" (no apostrophe) is tokenized and that `'ll` is the contracted form of "will" in terms such as "I'll.")

NMF was then used to provide insight into the underlying topics that users chose to use to describe themselves. Selecting the number of NMF components (topics to which users are clustered) is an arbitrary and iterative process. For the self-summary essay, we chose 25 components, which resulted in a diverse, but manageable, set of topics.

Several expected themes emerged. Many users chose to highlight personality traits, for example "humor" or "easy-going," while others focused on describing the types of activities they enjoyed. Hiking, traveling, and cooking were popular choices. Others chose to mention what kind of interaction they were seeking, whether that was a long-term relationship, a friendship, or sex. Topics and the highest weighted tokens for each are summarized in Table 3. Note that topic names were hand-labeled.

In order to determine whether there were differences in the topics that OkCupid users chose to write about in their self-summaries, we plotted the distribution over topics by demographic split. This allowed us to identify if specific topics were distinct to

10. https://spacy.io/docs#token-postags.

11. Unigrams are single tokens. Bigrams refer to two adjacent and trigrams to three adjacent tokens.

12. Stop words are words that appear with very high frequency, such as "the" or "to."

13. The difference between the number of users in the data set and the number of users in the analysis is due to the fact that we drop users that write less than five tokens for a particular essay.

Topic	Tokens
meet & greet	meet new people, looking meet new, love meeting new, new friends, enjoy meeting, interesting people, want meet, 'm new, people love, experiences
the city	san francisco, moved san francisco, city, living san francisco, just moved san, native, san diego, grew, originally, recently
enthusiastic	love travel, love laugh, love outdoors, love love, laugh, dance, love cook, especially, life love, love life
straight talk	know, just, want, ask, message, just ask, really, talk, write, questions
about me	'm pretty, 'm really, 'm looking, 'm just, say 'm, think 'm, 'm good, 'm trying, nerd, 'm working
novelty	new things, trying new, trying new things, new places, learning new things, exploring, restaurants, things love, love trying, different
seeking	'm looking, guy, relationship, looking meet, share, woman, nice, just looking, man, partner
carefree	easy going, 'm easy going, easy going guy, pretty easy going, laid, love going, enjoy going, simple, friendly, likes
casual	guy, lol, chill, nice, old, pretty, alot, laid, kinda, wanna
enjoy	like, 'd like, things like, really like, n't like, feel like, stuff, like people, like going, watch
transplant	moved, sf, years ago, school, east coast, city, just moved, college, went, california
nots	n't, ca n't, does n't, really, wo n't, n't like, n't know, n't really, did n't, probably
moments	spend time, good time, lot, free time, spending time, lot time, spend lot, time friends, time 'm, working
personality	humor, good sense humor, good time, good conversation, sarcastic, love good, dry, good company, appreciate, listener
amusing	fun loving, 'm fun, having fun, outgoing, guy, girl, adventurous, like fun, looking fun, spontaneous
review	let 's, think, way, self, right, thing, say, little, profile, summary
region	bay area, moved bay area, bay area native, grew, living, 'm bay area, east bay, raised bay area, east, originally
career-focused	work hard, play hard, hard working, progress, harder, job, try, love work, company, busy
locals	born, raised, born raised, california, raised bay area, college, school, sf, berkeley, oakland
unconstrained	open minded, creative, honest, relationship, adventurous, curious, passionate, intelligent, heart, independent
active	enjoy, friends, family, hiking, watching, outdoors, traveling, hanging, cooking, sports
creative	music, art, live, movies, live music, play, food, games, dancing, books
carpe diem	live, world, fullest, enjoy life, experiences, passionate, love life, moment, living life, life short
cheerful	person, people, make, laugh, think, funny, kind, happy, honest, smile
jet setter	've, lived, years, world, traveled, year, spent, countries, different, europe

TABLE 3: *Self-summary topics and associated terms.*

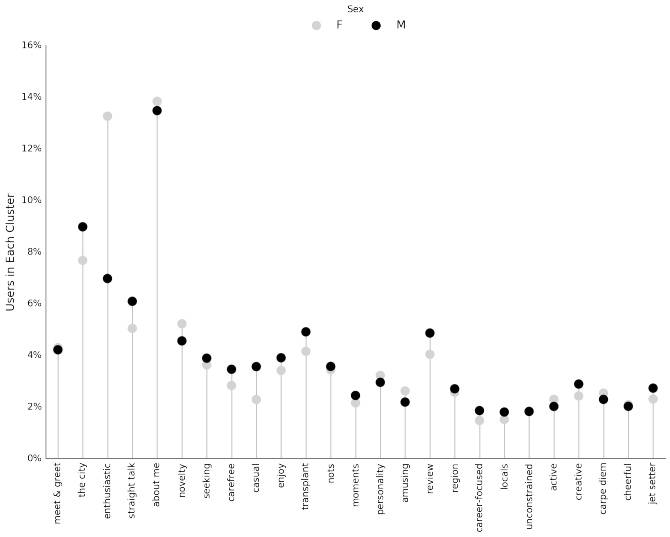

Fig. 1: *Self-summary distribution over topics*

particular demographic groups.

Figure 1 shows the distribution over topics by sex for the self-summary essay. The highest proportion of users, of either sex, were in the "about me" topic. This is not surprising given the essay prompt. For most topics, females and males were mostly evenly distributed. For example, the proportion of females who emphasized their careers or travel or other topics was similar to the proportion of males who did the same. One exception was with the "enthusiastic" topic, to which females belonged at almost twice the rate of males. Users in this group used modifiers such as, "love," "really," and "absolutely" regardless of the activities they were describing.

We further examined online self-presentation by considering the other available essays in the OkCupid data set. Previous psychology research suggests that a person's preferred music styles are tied to their personalities [Col15], and it is possible that this extends to other media, such as books or movies. We next analyzed the "Favorite books, movies, shows, music, food" essay.

As with the self-summaries, we removed users who wrote less than 5 tokens for this essay (11,836 such cases). Note that because the favorites text is less expository and more list-like, we did not perform a lexical-based analysis. Instead, we used NMF to identify topics (or genres). Like with the self-summaries, we chose 25 topics. Table 4 lists the topics and a selection of their highest weighted tokens.

The topics for this essay were less distinctive than the topics for the self-summaries. In some cases, genres (or media) overlapped. For example, the "TV-comedies-0" group included "The Walking Dead," which is a drama. There was also overlap between groups. Still, we decided to keep 25 components. The granularity these topics provided was used for further analyses. We created superordinate groupings from the topics from which we extracted distinctive tokens for particular demographic groups, showing the approach's flexibility. Figure 2 shows the distribution over topics, by sex.

The most popular topics, for both females and males, were "TV-hits" and "music-rock," with about 16 percent of each sex writing about shows or artists in those groups. We found more separation between the sexes in the favorites essay than we did with the self-summaries. As with the self-summary essay,

Topic	Tokens
like	like, music like, movies like, really like, stuff, food like, things, like music, books like, like movies
TV-hits	mad men, arrested development, breaking bad, 30 rock, tv, parks, sunny, wire, dexter, office
enthusiastic	love food, love music, love movies, love love, cook, love good, eat, food, love read, books love
favorite-0	favorite, favorite food, favorite movies, favorite books, favorite music, favorite movie, favorite book, favorite shows, favorite tv, time favorite
genres-movies	sci fi, action, comedy, horror, fantasy, movies, drama, romantic, classic, adventure
genres-music	hip hop, rock, r&b, jazz, reggae, rap, pop, country, classic, old
misc-0	fan, reading, food 'm, right, 'm big, really, currently, music 'm, just, open
TV-comedies-0	big bang theory, met mother, big lebowski, friends, house, office, community, walking dead, new girl, bones
genres-food	italian, thai, mexican, food, indian, chinese, japanese, sushi, french, vietnamese
nots	ca n't, watch, n't really, does, n't like, does n't, think, eat, n't watch tv, n't read
teen	harry potter, hunger games, twilight, dragon tattoo, pride prejudice, harry met sally, disney, vampire, trilogy, lady gaga
everything	books, movies, food, music, shows, country, dance, action, lots, horror
movies-drama-0	eternal sunshine, spotless mind, litte miss sunshine, amelie, garden state, lost, life, beautiful, lost translation, beauty
time periods	80, let, good, 90, life, just, 70, world, time, man
avid	read lot, time, watch, listen, recently, lately, love read, watch lot, favorites, just read
misc-1	list, just, long, ask, way, goes, things, try, favorites, far
music-rock	david, black, john, tom, radiohead, bob, brothers, beatles, black keys, bowie
movies-sci-fi	star, lord, wars, rings, star trek, trilogy, series, matrix, princess, bride
TV-comedies-1	modern family, family guy, office, south park, met mother, glee, simpsons, american dad, 30 rock, colbert
movies-drama-1	fight club, shawshank redemption, pulp fiction, fear loathing, peppers, red hot, vegas, american, catcher rye, big lebowski
kinds	kinds music, love kinds, kinds food, kinds movies, listen, different, country, foods, comedy, action
favorite-1	favorite book, favorite movie, food, music, good, fav, book read, reading, great, best
novelty	enjoy, new, types, trying, reading, things, foods, types music, films, different
TV-drama	game thrones, ender 's game, walking dead, true blood, series, currently, hunger games, dexter, song ice, boardwalk empire
genres-books	fiction, non fiction, science fiction, fiction books, read non fiction, historical fiction, films, books, documentaries, biographies

TABLE 4: *Favorites topics and associated terms.*

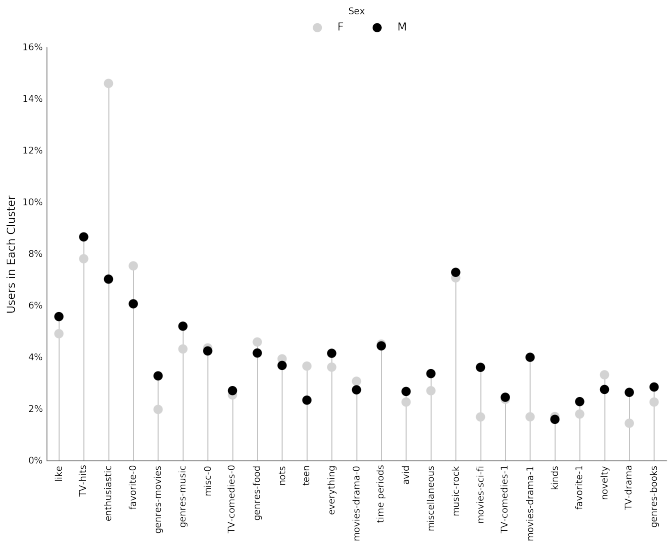

Fig. 2: *Favorites distribution over topics, by sex*

the enthusiastic group was distinctly female. A distinctly male category included films such as "Fight Club" and "The Shawshank Redemption" and musicians such as the Red Hot Chili Peppers.

We created superordinate groupings by combining clusters. There were four groups related to movies. In order to extract demographic-distinctive tokens, we used the smoothed log-odds-ratio which accounts for variance as described by Monroe, Colaresi, and Quinn [Mon09]. The top movies for females were Harry Potter, Pride & Prejudice, and Hunger Games while males favored Star Wars, The Matrix, and Fight Club. The "movies-sci-fi" and "movies-drama-1" groups, whose highest weighted tokens referred to the male-favored movies, had a higher proportion of males than females. Similarly, the "teen" group, which which corresponded to female-favored movies, had a higher proportion of females. This reflects the terms found by the log-odds-ratio.

Figure 3 shows the distribution over topics by drug usage. In this demographic category, users self-identified as drug users or non-drug users. To this, we added a third level for users who declined the state their drug usage status. There were 6,859 drug users, 29,402 non-drug users, and 11,849 users who did not state their drug usage status ("unknown").

There was more intra-cluster variation in the distribution of users across topics than for the demographic split by sex. Interestingly, the distribution across topics of users for whom we had no drug usage information — those in the "unknown" category — tended to track the distribution of self-identified drug users. In other words, the proportion of drugs users and unknown users in most topics was similar. This was especially true in cases where difference in proportions of drug users and non-drug users was large. This unexpected finding may suggest that individuals who did not respond to the drug usage question abstained in order to avoid admitting they did use drugs.

Although we were unable to test this hypothesis directly due to lack of the true drug-usage status for these users, the manner by which free-text writing styles may unintentionally disclose demographic attributes is an intriguing avenue for research. We used a predictive modeling approach to attempt to gain insights into this question. Specifically, we trained a logistic regression model on a binary outcome, using only drug users and non-

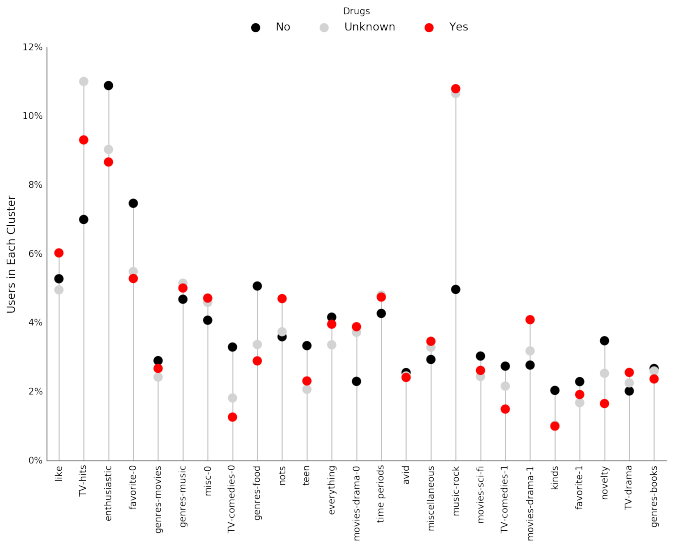

Fig. 3: *Favorites distribution over topics, by drug usage status*

drug users. We used tf-idf weights on unigrams, bigrams, and trigrams as in the previous analyses. We also balanced the classes by randomly sampling 6,859 accounts from the non-drug user population. The data was split into training (80%) and test (20%) sets in order to assess model accuracy. We then predicted class labels on the group of unknown drug usage status.

Our initial model, which used only the "Favorites" essay text, accurately predicted 68.0 percent of drug users. When applied to the unknown users upon which the model was not trained, the model predicted that 55 percent of the unknown users were drug users and that 45 percent were not. When we examined the proportion of predicted user by NMF cluster, however, we found intriguing patterns. In the "music-rock" group — the group with the largest disparity between users and non-users — 84 percent of unknowns were classified as drug users. In contrast, only 25 percent of the unknowns in the "TV-comedies-0" group were classified as such. While this cluster included "The Big Lebowski," which is identified as a "stoner film" [She13], it also features "The Big Bang Theory," "How I Met Your Mother," "NCIS," "New Girl," and "Seinfeld," which we would argue are decidedly not drug-related.

These results prompted us examine if we could predict drug usage status based on text alone. For this, we combined the text of all 10 essays and dropped the 2,496 users who used less than five tokens in the full-text. As before, we randomly sampled from the non-users in order to balance the classes and split the data into training and test sets.

The full-text model accuracy increased to 72.7 percent. We used the feature weights to find the 25 most-predictive drug-usage terms. These are listed below, with the odds ratio[14] shown in parentheses.

```
sex (68.96), shit (45.51), music (20.95),
weed (18.46), party (15.54), beer (14.18),
dubstep (13.86), fuck (12.28), drinking (11.48),
smoking (11.39), partying (10.59), chill (9.45),
hair (8.84), park (8.09), fucking (7.93), dj (7.9),
burning (7.78), electronic (7.05), drunk (6.67),
ass (6.36), reggae (6.18), robbins (5.81),
dude (5.74), smoke (5.68), cat (5.5)
```

Drug users in this data set reference drinking, smoking, partying, and music more than non-users and also use particular profane terms.

14. Logistic regression coefficient estimates are given as log-odds-ratios. The odds-ratios, which say how much a one unit increase affects the odds of being a drug user, are calculated by exponentiating.

Conclusion and Future Work

The current study extended previous NLP analyses of online dating profiles. The scope of this work was larger than previous studies, both because of the size of the data set and because of the novel combination of NLP with both supervised and unsupervised machine learning techniques, such as logistic regression and NMF. To our knowledge, there is currently no study that combines these techniques to identify unintentional cues in online self-presentation or uses them to predict demographics from free-text self descriptions. The idea that people may unintentionally be providing information about themselves in the way that they answer questions online is an intriguing avenue for future research and can also be extended to deception online.

This work serves as an initial exploration for analyzing self-presentation in the context of online dating. Given the availability of other demographic characteristics, such as ethnicity and education level, future work will focus on describing the ways in which other demographic groups tend to describe themselves. We would also like to explore recent advancements in language modeling techniques, such as word embeddings. Most importantly, future work will involve exploring methods to help us better identify deception. If the data ever becomes available, we would like to explore how the *way* that people choose to self-present affects the interactions they have.

Acknowledgements

This work began as a final project for the Applied Natural Language Processing course at the School of Information at the University of California, Berkeley. We would like to thank Marti Hearst for her guidance in the "right" way to do NLP and in pushing us to explore new and exciting data sets. We would also like to thank David Bamman for fruitful discussions on NLP and ideas for permutation testing. We would especially like to thank our reviewers, in particular David Lippa. His comments were invaluable for helping us organize our thoughts and analyses.

References

[Bir10] Bird, S., Klein, E., & Loper, E. (2009). Natural language processing with Python. "O'Reilly Media, Inc.".

[Col15] Collingwood, J. (2015). Preferred Music Style Is Tied to Personality. Psych Central. Retrieved on June 22, 2016, from http://psychcentral.com/lib/preferred-music-style-is-tied-to-personality/

[Fio05] Fiore, A. T., & Donath, J. S. (2005, April). Homophily in online dating: when do you like someone like yourself?. In CHI'05 Extended Abstracts on Human Factors in Computing Systems (pp. 1371-1374). ACM.

[Fio08] Fiore, A. T., Taylor, L. S., Mendelsohn, G. A., & Hearst, M. (2008, April). Assessing attractiveness in online dating profiles. In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (pp. 797-806). ACM.

[Ger12] Gerber, A. S., & Green, D. P. (2012). Field experiments: Design, analysis, and interpretation. WW Norton.

[Hon16] Honnibal, M (2016). spaCy. [Computer software]. https://spacy.io/.

[Kim15] Kim, A. Y., & Escobedo-Land, A. (2015). OkCupid Data for Introductory Statistics and Data Science Courses. Journal of Statistics Education, 23(2), n2.

[McP01] McPherson, M., Smith-Lovin, L., & Cook, J. M. (2001). Birds of a feather: Homophily in social networks. Annual review of sociology, 415-444.

[Mon09] Monroe, B. L., Colaresi, M. P., & Quinn, K. M. (2008). Fightin'words: Lexical feature selection and evaluation for identifying the content of political conflict. Political Analysis, 16(4), 372-403.

[Nag09] Nagarajan, M., & Hearst, M. A. (2009, March). An Examination of Language Use in Online Dating Profiles. In ICWSM.

[Oja10] Ojala, M., & Garriga, G. C. (2010). Permutation tests for studying classifier performance. Journal of Machine Learning Research, 11(Jun), 1833-1863.

[Ped11] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., & Vanderplas, J. (2011). Scikit-learn: Machine learning in Python. Journal of Machine Learning Research, 12(Oct), 2825-2830.

[Pet11] Petrov, S., Das, D., & McDonald, R. (2011). A universal part-of-speech tagset. arXiv preprint arXiv:1104.2086.

[Pew16] Smith, Aaron, & Anderson, Monica (2016). 5 Facts About Online Dating. Retrieved from http://www.pewresearch.org/fact-tank/2016/02/29/5-facts-about-online-dating/.

[She13] Sheffield, Rob (2013). 10 Best Stoner Movies of All Time. Rolling Stones. Retrieved on June 23, 2016, from http://www.rollingstone.com/movies/lists/the-greatest-stoner-movies-of-all-time-20130606

[Wet15] Everett Wetchler, okcupid, (2015), GitHub repository, https://github.com/everett-wetchler/okcupid.git

[Xu_03] Xu, W., Liu, X., & Gong, Y. (2003, July). Document clustering based on non-negative matrix factorization. In Proceedings of the 26th annual international ACM SIGIR conference on Research and development in informaion retrieval (pp. 267-273). ACM.

PyTeCK: a Python-based automatic testing package for chemical kinetic models

Kyle E. Niemeyer‡*

https://youtu.be/Ke3Ip25C4pY

Abstract—Combustion simulations require detailed chemical kinetic models to predict fuel oxidation, heat release, and pollutant emissions. These models are typically validated using qualitative rather than quantitative comparisons with limited sets of experimental data. This work introduces PyTeCK, an open-source Python-based package for automatic testing of chemical kinetic models. Given a model of interest, PyTeCK automatically parses experimental datasets encoded in a YAML format, validates the self-consistency of each dataset, and performs simulations for each experimental data point. It then reports a quantitative metric of the model's performance, based on the discrepancy between experimental and simulated values and weighted by experimental variance. The initial version of PyTeCK supports shock tube and rapid compression machine experiments that measure autoignition delay.

Index Terms—combustion, chemical kinetics, model validation

Introduction

Combustion simulations require chemical kinetic models to predict fuel oxidation, heat release, and pollutant emissions. These models are typically validated using qualitative, rather than quantitative, comparisons with limited sets of experimental data. Furthermore, while a plethora of published data exist for quantities of interest such as autoignition delay and laminar flame speed, most are not available in a standardized machine-readable format. Such data is commonly offered in poorly documented, nonstandard CSV files and Excel spreadsheets, or even contained in PDF tables or figures.

This work aims to support quantitative validation of kinetic models by:

1. Encouraging the use of a human- and machine-readable format to encode experimental data for combustion.
2. Offering an efficient, automated software package, PyTeCK, that quantitatively evaluates the performance of chemical kinetics models based on available experimental data.

Fundamental combustion experiments typically study the behavior of fuels in idealized configurations at conditions relevant to applications in transporation, aerospace, or power generation. These produce useful data for validating chemical kinetic models, which in turn can support simulations of more complex applications such as internal combustion or gas turbine engines. The autoignition delay of a fuel/oxidizer mixture represents the time required for the mixture to ignite (i.e., experience a rapid increase in temperature and corresponding consumption of fuel and oxidizer) after arriving at a specified initial state. Autoignition occurs in practical applications such as knock in spark-ignition engines or ignition in compression-ignition and gas turbine engines, and so ignition delay measurements provide useful validation measures for models aimed at capturing such phenomena. Other combustion experimental measurements—such as extinction in perfectly stirred reactors, species profiles in jet-stirred reactors, and laminar flame speeds—also provide useful information about fuel combustion characteristics, but these are not considered in this paper.

Ignition delay times are typically measured with two categories of experiments: shock tubes and rapid compression machines. In shock tubes, a diaphragm separates high-pressure gases from a lower-pressure mixture of fuel and oxidizer. Rupturing the diaphragm propagates a (compressive) shock wave into the fuel/oxidizer mixture, quickly increasing the temperature and pressure and leading to autoignition after a time delay. Chaos and Dryer [Chaos2010], and more recently Hanson and Davidson [Hanson2014], discuss shock tubes in more detail. In contrast, rapid compression machines, reviewed by Sung and Curran [Sung2014], emulate a single compression stroke in an internal combustion engine; the compression of a piston raises the temperature and pressure of a fuel/oxidizer mixture in a short period of time, after which ignition occurs. Shock tubes and rapid compression machines offer complementary approaches to measuring ignition delay times. Shock tubes can investigate a wide range of temperatures (600–2500 K) [Hanson2014], although problems with pre-ignition pressure rise occur at higher pressures and temperatures below around 1100 K [Petersen2009], [Chaos2010], while rapid compression machines can reach low-to-intermediate temperatures (600–1100 K) [Sung2014].

In this paper, I propose a data format for capturing results from experimental measurements of autoignition delay times. This paper also describes the components of PyTeCK (**Py**thon-based **te**sting of **c**hemical **k**inetic models), a software package that quantifies the performance of a chemical kinetic model in reproducing experimental ignition delays. This includes discussion of the experimental data parser, simulation framework, and solution post-processing. The paper also explains the theoretical basis for

* *Corresponding author: Kyle.Niemeyer@oregonstate.edu*
‡ *School of Mechanical, Industrial, and Manufacturing Engineering, Oregon State University*

the models of shock tubes and rapid compression machines.

Implementation of PyTeCK

PyTeCK is packaged as a standard Python package using setuptools, and consists of three primary modules:

1. `parse_files` contains functions to read the YAML-encoded experimental data file using the `PyYAML` module. Smaller functions comprise this process to enable easier unit testing.
2. `simulation` contains the `Simulation` class and relevant functions for initiating, setting up, and running cases, and then processing the results.
3. `eval_model` uses the previous two modules to set up simulations based on experimental data, and then runs simulations in parallel using the `multiprocessing` module.

The next three sections explain the implementation of each primary module. PyTeCK also includes the module `detect_peaks`, based on the work of Duarte [Duarte2015], for detecting peaks in targeted quantities (e.g., pressure, temperature) to determine the ignition delay time. Supporting modules in PyTeCK include `exceptions` for raising exceptions while reading YAML files, `utils` that initializes a single Pint-based unit registry [Grecco2016], and `validation` that provides quantity validation functions.

PyTeCK relies on well-established scientific Python software tools. These include NumPy [vanderWalt2011] for large array manipulation, SciPy [Jones2001] for interpolation, Pint [Grecco2016] for interpreting and converting between units, PyTables [Alted2002] for HDF5 file manipulation, Cantera [Goodwin2016] for chemical kinetics, and `pytest` [Krekel2016] for unit testing. Travis-CI [Travis2016] also provides continuous integration testing.

PyTeCK is available under an open-source MIT license via a GitHub repository [Niemeyer2016b]. It can be installed using setuptools by downloading the source code files and executing `python setup.py install`. More mature versions of PyTeCK will be distributed on PyPI (Python Package Index).

Parsing ChemKED files

The PyTeCK module `parse_files` parses experimental data encoded in the ChemKED (**chem**ical **k**inetics **e**xperimental **d**ata) format proposed by this paper. ChemKED builds on XML-based `ReSpecTh` of Varga et al. [Varga2015a], [Varga2015b]—which in turn builds on the PrIMe data format [Frenklach2007], [You2012], [PrIMe2016]—but is written in YAML instead of XML. While XML is a powerful markup language, YAML offers a number of advantages: parsers and libraries exist for most programming langauges, it supports multiple data types and arrays. YAML files are also intended for data and more readable by humans, which allows easier composition and could encourage adoption.

The code block below shows a complete example of an autoignition dataset for an hydrogen/oxygen/argon (`H2/O2/Ar`) mixture, taken from Figure 12 (right) of Chaumeix et al. [Chaumeix2007]:

```
---
file-author:
    name: Kyle E Niemeyer
    ORCID: 0000-0003-4425-7097
file-version: (1, 0)
reference:
    doi: 10.1016/j.ijhydene.2007.04.008
    authors:
        - name: N. Chaumeix
          ORCID:
        - name: S. Pichon
          ORCID:
        - name: F. Lafosse
          ORCID:
        - name: C.-E. Paillard
          ORCID:
    journal: International Journal of Hydrogen Energy
    year: 2007
    volume: 32
    pages: 2216-2226
    detail: Fig. 12., right, open diamond
experiment-type: Ignition delay
apparatus:
    kind: shock tube
    institution: CNRS-ICARE
    facility: stainless steel shock tube
common-properties:
    pressure: &pres
        value: 220
        units: kilopascal
    composition: &comp
        - species: H2
          InChI: 1S/H2/h1H
          mole-fraction: 0.00444
        - species: O2
          InChI: 1S/O2/c1-2
          mole-fraction: 0.00566
        - species: Ar
          InChI: 1S/Ar
          mole-fraction: 0.9899
    ignition-type: &ign
        target: pressure
        type: d/dt max
datapoints:
    - temperature:
        value: 1164.48
        units: kelvin
      ignition-delay:
          value: 471.54
          units: us
      pressure: *pres
      composition: *comp
      ignition-type: *ign
    - temperature:
        value: 1164.97
        units: kelvin
      ignition-delay:
          value: 448.03
          units: us
      pressure: *pres
      composition: *comp
      ignition-type: *ign
    - temperature:
        value: 1264.2
        units: kelvin
      ignition-delay:
          value: 291.57
          units: us
      pressure: *pres
      composition: *comp
      ignition-type: *ign
    - temperature:
        value: 1332.57
        units: kelvin
      ignition-delay:
          value: 205.93
          units: us
      pressure: *pres
      composition: *comp
      ignition-type: *ign
    - temperature:
        value: 1519.18
```

```
      units: kelvin
    ignition-delay:
        value: 88.11
        units: us
    pressure: *pres
    composition: *comp
    ignition-type: *ign
```

This example contains all the information needed to evaluate the performance of a chemical kinetic model with five data points. The file also includes metadata about the file itself, as well as reference information. While these elements, including `file-author`, `file-version`, and the entries in `reference`, are not required by PyTeCK, a valid ChemKED file should include this information for completeness. The elements necessary for PyTeCK include the type of experiment given by `experiment-type` (currently limited to `Ignition delay`), the `kind` of apparatus used to measure ignition delay (`shock tube` or `rapid compression machine`), and then a list of experimental `datapoints` given as associative arrays with necessary information. Mandatory elements of each entry in"datapoints" include the initial `temperature`, `pressure`, and mixture `composition`, as well as the experimental `ignition-delay` and `ignition-type` (means by which PyTeCk detects ignition, discussed in more detail later). All quantities provided include a magnitude and units, which Pint [Grecco2016] interprets. Since many experimental datasets hold certain properties constant (e.g., composition, pressure) while varying a single quantity (e.g., temperature), a `common-properties` element can describe properties common to all `datapoints`, using an arbitrary anchor label (e.g., `&pres` above for the constant pressure). Each data point then refers to the common property with a reference (`*pres`). However, every data point should still contain the complete information needed to reproduce its conditions; the `common-properties` element is used for convenience.

Modeling ignition in shock tubes or RCMs may require more elements to capture effects not accounted for by the simplest models. Under certain conditions that lead to longer ignition delay times, shock tubes can exhibit pressure rise before ignition. This is typically expressed in the literature with a constant pressure rise rate at a fraction of the initial pressure (with units of inverse time), and ChemKED files encode this as an item in the associative array describing an experimental data point:

```
pressure-rise:
    value: 0.10
    units: 1/ms
```

Later versions of PyTeCK will support specifying a pressure-time history directly, although these are not commonly published in the shock tube literature.

Simulations of RCM experiments commonly provide a volume-time history to capture nonideal pre- and post-ignition heat losses, as well as effects due to the compression stroke. This data can be provided with experimental `datapoints` in ChemKED as a list of lists, with the `column` index and units identified:

```
volume-history:
    time:
        units: s
        column: 0
    volume:
        units: cm3
        column: 1
    values:
        - [0.00E+000, 5.47669375000E+002]
        - [1.00E-003, 5.46608789894E+002]
```

The PyTeCK `tests` directory [Niemeyer2016b] contains more examples of ChemKED files for shock tube and RCM experiments.

The function `parse_files.read_experiment()` takes a ChemKED-format file as input, and returns a dictionary with the necessary information to perform simulations of the experimental data points. The `parse_files.get_experiment_kind()` and `parse_files.get_datapoints()` functions perform important checking of input information for consistency and validity of quantities via the `validation` module. For example, after detecting the specified initial temperature, `get_datapoints()` checks the correct dimensionality of units and range of magnitude (in this case, that the units are consistent with Kelvin and that the magnitude is greater than zero),

```
validation.validate_gt('temperature',
                       case['temperature'],
                       0. * units.kelvin
                       )
```

where the `validation.validate_gt()` function—borrowed heavily from Huff and Wang's PyRK [Huff2015], [Huff2015b]—is

```
def validate_gt(value_name, value, low_lim):
"""Raise error if value not greater than lower
limit or wrong type.

Parameters
----------
value_name : str
    Name of value being tested
value : int, float, numpy.ndarray, pint.Quantity
    Value to be tested
low_lim : type(value)
    ``value`` must be greater than this limit

Returns
-------
value : type(value)
    The original value

"""
try:
    if not validate_num(value_name, value) > low_lim:
        msg = (value_name + ' must be greater than ' +
               str(low_lim) + '.\n'
               'Value provided was: ' + str(value)
               )
        # RuntimeError used to avoid being caught by
        # Pint comparison error. Pint should really
        # raise TypeError (or something) rather than
        # ValueError.
        raise RuntimeError(msg)
    else:
        return value
except ValueError:
    if isinstance(value, units.Quantity):
        msg = ('\n' + value_name +
               ' given with units, when variable '
               'should be dimensionless.'
               )
        raise pint.DimensionalityError(value.units,
                                       None,
                                       extra_msg=msg
                                       )
    else:
        msg = ('\n' + value_name +
               ' not given in units. Correct '
```

```
                'units share dimensionality with: ' +
                str(low_lim.units)
                )
            raise pint.DimensionalityError(None,
                                           low_lim.units,
                                           extra_msg=msg
                                           )
except pint.DimensionalityError:
    msg = ('\n' + value_name +
           ' given in incompatible units. Correct '
           'units share dimensionality with: ' +
           str(low_lim.units)
           )
    raise pint.DimensionalityError(value.units,
                                   low_lim.units,
                                   extra_msg=msg
                                   )
except:
    raise
```

The `read_experiment()` function also checks that necessary parameters are present, and also for consistency between input parameters based on the particular experiment type being modeled. For example, an input ChemKED file describing a shock tube experiment cannot include `compression-time` or `volume-history` elements.

After parsing and checking the simulation parameters, the `parse_files.create_simulations()` function creates a list of `Simulation` objects.

Autoignition simulation procedure

Once `parse_files.create_simulations()` initializes a list of `Simulation` objects, the member function `setup_case()` prepares each object to perform a simulation by initiating the governing equations that model shock tubes and rapid compression machines. These equations are briefly described next.

A composition state vector Φ defines the thermochemical state of a general chemical kinetic system:

$$\Phi = \left\{T, Y_1, Y_2, \ldots, Y_{N_{\text{sp}}}\right\} ,$$

where T is the temperature, Y_i is the mass fraction of the ith species, and N_{sp} is the number of species represented by the chemical kinetic model. A system of ordinary differential equations advances this thermochemical state when modeling both experimental types, derived from conservation of mass and energy:

$$\frac{d\Phi}{dt} = \left\{\frac{dT}{dt}, \frac{dY_1}{dt}, \frac{dY_2}{dt}, \ldots, \frac{dY_{N_{\text{sp}}}}{dt}\right\} . \tag{1}$$

The derivative terms in Equation (1) come from the conservation of energy

$$\frac{dT}{dt} = \frac{-1}{c_v}\left(\sum_{i=1}^{N_{\text{sp}}} e_i \frac{dY_i}{dt} + p\frac{dv}{dt}\right) \tag{2}$$

and conservation of mass

$$\frac{dY_i}{dt} = \frac{1}{\rho} W_i \dot{\omega}_i \quad i = 1, \ldots, N_{\text{sp}} , \tag{3}$$

where c_v is the mass-averaged constant-volume specific heat of the mixture, e_i is the internal energy of the jth species in mass units, v is the specific volume of the mixture, and $\dot{\omega}_i$ is the overall molar production rate of the ith species.

PyTeCK relies on Cantera [Goodwin2016] for handling most chemical kinetics calculations. Cantera is an open-source software library that provides tools for solving problems related to chemical kinetics, thermodynamics, and transport processes. The core of Cantera is written in C++, but it provides interfaces for Python and Matlab. PyTeCK uses a Cantera [Goodwin2016] `ReactorNet` object to solve the system given by Equation (1), by connecting `IdealGasReactor` and `Reservoir` objects separated by a `Wall`. The `Wall` may or may not be moving, depending on whether the modeled system has varying or constant volume, respectively.

The simplest way to model both shock tubes and RCM experiments is by assuming an adiabatic, constant-volume process. In this case, I simplify Equation (2) by assuming $\frac{dv}{dt} = 0$, and the `Wall` is initialized with `velocity=0`:

```
self.wall = ct.Wall(self.reac, env, A=1.0, velocity=0)
```

This approach does not account for either preignition pressure rise observed in some shock tube experiments [Chaos2010], [Hanson2014] or heat loss in RCMs [Sung2014]. RCM volume histories are typically provided directly, but publications describing shock tube experiments with observed preignition pressure rise usually instead give a constant pressure-rise rate $\frac{dP}{dt}$. This is incorporated into Equation (2) by determining an associated preignition pressure history $p(t)$:

$$p(t) = p_0 + \int_0^{t_{\text{end}}} \frac{dp}{dt} dt , \tag{4}$$

where p_0 is the initial pressure and t_{end} the time interval of interest (typically the ignition delay time). The function `simulation.sample_rising_pressure()` actually constructs this pressure history, which is then used to construct a volume history $v(t)$ assuming isentropic compression:

$$v(t) = v_0 \frac{\rho_0}{\rho(t)}\bigg|_{s_0} , \tag{5}$$

where v_0 is the initial volume, ρ is the density, ρ_0 is the initial density, and s_0 is the specific entropy of the initial mixture.

The varying volume of the system is handled by assigning the `velocity` attribute of the `ReactorNet`'s '`Wall` to one of two classes: `VolumeProfile` when volume history is provided

```
self.wall = ct.Wall(
    self.reac, env, A=1.0,
    velocity=VolumeProfile(self.properties)
    )
```

and `PressureRiseProfile` when pressure-rise value is specified

```
self.wall = ct.Wall(
    self.reac, env, A=1.0,
    velocity=PressureRiseProfile(
        mechanism_filename, initial_temp,
        initial_pres, reactants,
        self.properties['pressure-rise'].magnitude,
        self.time_end
        )
    )
```

PyTeCK needs more details about the chemical kinetic model and initial conditions to initialize the `PressureRiseProfile` object, and specifically to construct the discrete volume-time history via Equations (4) and (5) using the `simulation.create_volume_history()` function. Objects of both classes contain the derivative of volume dv/dt, which PyTeCK obtains by numerically differentiating the volume history via `simulation.first_derivative()`. This

function uses `numpy.gradient()` to calculate second-order central differences at interior points and second-order one-sided differences (either forward or backward) at the edge points. When called, the `VolumeProfile` or `PressureRiseProfile` object returns the derivative of volume at the specified time (i.e., the velocity of the `Wall`), using `numpy.interp()` to interpolate as needed.

After each `setup_case()` prepares a `Simulation` object, the `run_case()` member function actually runs each simulation. PyTeCK prepares and runs each simulation independently to allow the use of `multiprocessing` workers to perform these steps in parallel (if desired), as described in the next section. When running a simulation, PyTeCK creates an HDF5 file and opens it as a PyTables [Alted2002] table, then performs integration steps until it reaches the desired end time (set as 100 times the experimental ignition delay). At every timestep, `run_case()` saves the time and information about the current thermochemical state (temperature, pressure, volume, and species mass fractions) to the HDF5 table. The Cantera `ReactorNet.step()` function performs a single integration step, selecting an appropriate time-step size based on estimated integration error. Internally, `step()` uses the CVODE implicit integrator [Cohen1996], part of the SUNDIALS suite [Hindmarsh2005], to advance the state of the `IdealGasReactor` contained by the `ReactorNet`.

Finally, a call to the `process_results()` member function determines the autoignition delay by opening the saved simulation results. The method by which it detects ignition depends on the target and type specified in the input ChemKED file. Target quantities include pressure, temperature, and mass fractions of commonly used species such as the OH and CH radicals (as well as their excited equivalents OH^* and CH^*). `process_results()` detects ignition by finding the location of either the maximum value of the target quantity (e.g., `type: max`) or the maximum value of the derivative of the quantity (e.g., `type: d/dt max`):

```
# Analysis for ignition depends on type specified
if self.ignition_type == 'd/dt max':
# Evaluate derivative
target = first_derivative(time, target)

# Get indices of peaks
ind = detect_peaks(target)

# Fall back on derivative if max value doesn't work.
    if len(ind) == 0 and self.ignition_type == 'max':
        target = first_derivative(time, target)
        ind = detect_peaks(target)

    # Get index of largest peak
    # (overall ignition delay)
    max_ind = ind[np.argmax(target[ind])]

    # add units to time
    time *= units.second

    # Will need to subtract compression time for RCM
    time_comp = 0.0
    if 'compression-time' in self.properties:
        time_comp = self.properties['compression-time']

    ign_delays = time[
        ind[np.where((time[ind[ind <= max_ind]] -
        time_comp) > 0)]
        ] - time_comp

    # Overall ignition delay

    if len(ign_delays) > 0:
        ign_delay = ign_delays[-1]
    else:
        ign_delay = 0.0 * units.second
    self.properties[
        'simulated ignition delay'
        ] = ign_delay
```

using the `detect_peaks.detect_peaks()` function [Duarte2015].

Evaluation of model performance

The approach used by PyTeCK to report performance of a chemical kinetic model is adapted from the work of Olm et al. [Olm2014], [Olm2015], and briefly discussed by Niemeyer [Niemeyer2016].

The function `eval_model.evaluate_model()` controls the overall evaluation procedure, given the required and optional parameters:

- `model_name`: a string with the name of the Cantera-format chemical kinetic model file (e.g., CTI file)
- `spec_keys_file`: a string with the name of a YAML file identifying important species
- `dataset_file`: a string with the name of a file listing the ChemKED files to be used, which gives the filenames in a newline delimited list
- `model_path`: a string with the directory containing `model_name`. This is optional; the default is `'models'`
- `results_path`: a string with the directory for placing results files. This is optional; the default is `'results'`
- `model_variant_file`: a string with the name of a YAML file identifying ranges of conditions for variants of the kinetic model. This is optional; the default is `None`
- `num_threads`: an integer with the number of CPU threads to use to perform simulations in parallel. This is optional; the default is the maximum number of available threads minus one

A few of these parameters require greater explanation. The chemical kinetic model, also referred to as "chemical reaction mechanism", needs to be provided in Cantera's CTI file (CanTera Input file) format [Goodwin2016]. This file contains a description of the elements, species (including names, molecular composition, and thermodynamic property data), and reactions (including reversibility, stoichiometry, Arrhenius rate parameters, third-body species efficiencies, and pressure dependence). Although the use of the CTI format in the literature has increased recently, often models are instead available in the older Chemkin format [Kee1996]. Such files can be converted using the Cantera-provided utility `ck2cti`.

PyTeCK needs the species key YAML file `spec_keys_file` because different chemical kinetic models internally use different names for species. PyTeCK interprets these names to set the initial mixture composition, and potentially identify a species target to detect ignition. This file contains entries (for multiple model files, if desired) of the form:

```
---
model_name:
    H2: "H2"
    O2: "O2"
    Ar: "AR"
```

where the key indicates the internal PyTeCK species name and the value is the name used by the model. In this case, the

necessary species names are consistent with the names used internally by PyTeCK, other than the capitalization of argon (`AR`). Names will likely differ for other kinetic models; for example, internally `nC7H16` represents the species *n*-heptane, while other models may use `C7H16`, `C7H16-1`, or `NXC7H16`, for example. PyTeCK's internal naming convention for key species is given by the `SPEC_KEY` and `SPEC_KEY_REV` dictionaries in the `utils` module, and can be obtained by calling `utils.print_species_names()`. For correct results the species name keys given in the `spec_keys_file` file only need to match names of species in the ChemKED files.

The `model_variant_file` YAML file is needed in certain (uncommon) cases where the chemical kinetic model needs manual changes to apply to different ranges of conditions (such as pressure or bath gas). In other words, different versions of the CTI file need to be created for accurate performance under different conditions. This file may contain entries of the form:

```
---
model_name:
    bath gases:
        N2: "_N2"
        Ar: "_Ar"
    pressures:
        1: "_1atm.cti"
        9: "_9atm.cti"
        15: "_15atm.cti"
        50: "_50atm.cti"
        100: "_100atm.cti"
```

where the keys are extensions added to `model_name`, in order of `bath gases` and then `pressures`, and the values represent the extensions to the base filename given by `model_name`. For models that need such variants, all combinations need to be present in the `model_path` directory. As an example, the kinetic model of Haas et al. [Haas2009] for mixtures of *n*-heptane, isooctane, and toluene, which I term `Princeton-2009`, has certain reactions that require rate parameters to be changed manually for different bath gases and pressure ranges. For a case with nitrogen as the bath gas and at pressures around 9 atm, the resulting file name would be `Princeton-2009_N2_9atm.cti`.

To determine the performance of a given model, `evaluate_model()` parses the ChemKED file(s), then sets up and runs simulations as described. A `multiprocessing.Pool` can perform simulations in parallel if multiple CPU threads are available, creating `simulation_worker` objects for each case. Then, `process_results()` calculates the simulated ignition delays.

PyTeCK reports the overall performance of a model by the average error function over all the experimental datasets:

$$E = \frac{1}{N}\sum_{i=1}^{N} E_i \tag{6}$$

where N is the number of datasets and E_i is the error function for a particular dataset. A lower E value indicates that the model better matches the experimental data. The error function for a dataset E_i is the average squared difference of the ignition delay times divided by the variance of the experimental data:

$$E_i = \frac{1}{N_i}\sum_{j=1}^{N_i}\left(\frac{\log\tau_{ij}^{\text{exp}} - \log\tau_{ij}^{\text{sim}}}{\sigma(\log\tau_{ij}^{\text{exp}})}\right)^2, \tag{7}$$

where N_i is the number of data points in dataset i, τ_{ij} is the jth ignition delay value in the ith dataset, σ is the experimental variance, log indicates the natural logarithm (rather than base-10), and the superscripts "exp" and "sim" represent experimental and simulated results, respectively.

The experimental variance σ serves as a weighting factor for datasets based on the estimated uncertainty of results. This term reduces the contribution to E of a dataset with high variance, from discrepancies between model predictions and experimental data, compared to datasets with lower variance. Ideally, publications describing experimental results would provide uncertainty values for ignition delay results, but these are difficult to estimate for shock tube and rapid compression machines and thus not commonly reported. Thus, for now, PyTeCK estimates all variance values.

PyTeCK estimates the variance with the `eval_model.estimate_std_dev()` function, by first fitting a `scipy.interpolate.UnivariateSpline()` of order three (or less, if the fit fails) to the natural logarithm of ignition delay values for a given dataset (where results mainly vary with a single variable, such as temperature), and then calculating the standard deviation of the differences between the fit and experimental data via `numpy.std()`. PyTeCK sets 0.1 as a lower bound for the uncertainty in ignition delay time, based on the precedent set by Olm et al. [Olm2014], [Olm2015].

After calculating the error associated with a dataset using Equation (7) and the overall error metric for a model using Equation (6), `evaluate_model()` saves the performance results to a YAML file and returns the associated dictionary if `evaluate_model()` was called programmatically. If the `--print` command line option was given, or the `print_results` option set to `True` when calling `evaluate_model()`, then the results are also printed to screen.

Example Usage

This section provides an example of using PyTeCK to compare the performance of 12 chemical kinetic models for hydrogen oxidation [Niemeyer2016c] using a collection of experimental shock tube ignition delay data [Niemeyer2016d]. 54 data sets from 14 publications comprise this collection, with a total of 786 ignition data points. Both the set of models and ChemKED experimental data set are available openly via the respective references.

After installing PyTeCK [Niemeyer2016b], and placing the model and experimental data files in appropriate locations (`h2-models` and `h2-files`, in this example), each model can be evaluated by executing a command similar to `PyTeCK -m GRI30-1999.cti -k h2-model-species-keys.yaml -d h2-data-list.txt -dp h2-files -mp h2-models`, with the appropriate model name inserted in place of `GRI30-1999.cti`.

Figure 1 compares the performances of the 12 hydrogen models, showing both the average error function E as well as the standard deviation of E_i values across data sets. Lower error function values indicate better agreement with experimental data. While the actual values are not important for the current example, generally both the average and variation of error function decrease with publication year of the models---indicating an overall improvment of model fidelity with time. Although this example only considers subsets of both the models and experimental data of Olm et al.'s study [Olm2014], the results generally agree.

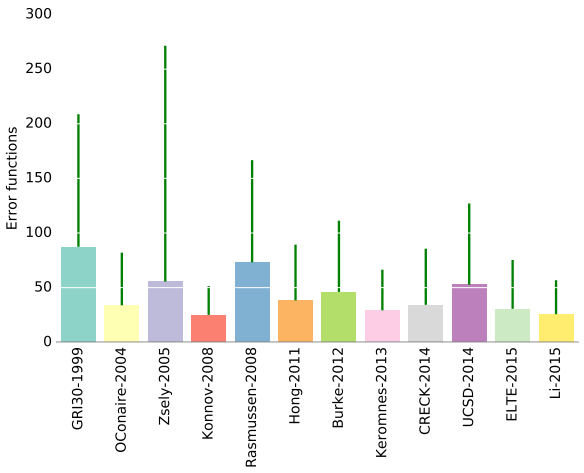

Fig. 1: *Average error functions, with standard deviations, for the 12 models of hydrogen oxidation. Models are arranged in order of publication, going from the oldest to the newest.*

Conclusions and Future Work

PyTeCK provides an open, Python-based framework for rigorously quantifying the performance of chemical kinetic models using experimental autoignition data generated from shock tube and rapid compression machine experiments. It can be used to compare models for describing the combustion of a given fuel and identify areas for improvement. Along with the software framework, this paper describes a new YAML-based data standard, ChemKED, that encodes experimental results in a human- and machine-readable manner.

Immediate plans for PyTeCK include better documentation generated by Sphinx [Brandl2016] and hosted on Read The Docs. Longer term plans for PyTeCK include extending support for other experimental types, including laminar flames and flow reactors, building in visualization of results, and creating an open database of ChemKED files for experimental data.

Acknowledgments

I thank Bryan Weber of the University of Connecticut for helpful discussions on the ChemKED format and an early review of this paper. I also thank Matt McCormick, Erik Tollerud, and Katy Huff for their helpful review comments.

Appendix

The following code snippet can be used to reproduce Fig. 1 using the produced by PyTeCK following the instructions given in the Example Usage section.

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.ticker import ScalarFormatter
from matplotlib.backends.backend_pdf import PdfPages
import brewer2mpl
import yaml

names = ['GRI30-1999', 'OConaire-2004', 'Zsely-2005',
         'Konnov-2008', 'Rasmussen-2008', 'Hong-2011',
         'Burke-2012', 'Keromnes-2013', 'CRECK-2014',
         'UCSD-2014', 'ELTE-2015', 'Li-2015'
         ]

ind = np.arange(len(names))

error_funcs = []
error_stds = []
for name in names:
with open(name + '-results.yaml', 'r') as f:
    results = yaml.load(f)
    error_func = results['average error function']
    std_dev = results['error function '
                      'standard deviation']
    error_funcs.append(error_func)
    error_stds.append(std_dev)

# colors for boxes
box_colors = brewer2mpl.get_map('Set3',
                                'qualitative',
                                len(names)
                                ).mpl_colors

fig, ax = plt.subplots()
yerr = [np.zeros(len(names)), error_stds]
ax.bar(ind, error_funcs, align='center',
       color=box_colors, linewidth=0,
       yerr=yerr, error_kw=dict(ecolor='g',
       lw=2, capsize=0)
       )

fmt = ScalarFormatter(useOffset=False)
ax.xaxis.set_major_formatter(fmt)

ax.set_ylabel('Error functions')
ax.set_xticks(ind)
ax.set_xticklabels(names, rotation='vertical')
ax.set_xlim([-0.5, ind[-1] + 0.5])
plt.subplots_adjust(bottom=0.25)

ax.spines['top'].set_visible(False)
ax.spines['right'].set_visible(False)
ax.spines['left'].set_visible(False)
ax.yaxis.set_ticks_position('none')
ax.xaxis.set_ticks_position('none')
ax.grid(axis = 'y', color ='white', linestyle='-')

plt.show()
```

References

[Alted2002] F. Alted, I. Vilata, and others. "PyTables: Hierarchical Datasets in Python," 2002–. http://www.pytables.org/

[Brandl2016] G. Brandl and others. "Sphinx: Python documentation generator," version 1.4.2, 2016. http://sphinx-doc.org/

[Chaos2010] M. Chaos, F. L. Dryer. "Chemical-kinetic modeling of ignition delay: Considerations in interpreting shock tube data," *Int. J. Chem. Kinet.*, 42:143–50, 2010. https://dx.doi.org/10.1002/kin.20471

[Chaumeix2007] N. Chaumeix, S. Pichon, F. Lafosse, and C.-E. Paillard. "Role of chemical kinetics on the detonation properties of hydrogen/natural gas/air mixtures," *Int. J. Hydrogen Energy*, 32:2216–2226, 2007. https://dx.doi.org/10.1016/j.ijhydene.2007.04.008

[Cohen1996] S. D. Cohen and A. C. Hindmarsh. "CVODE, A Stiff/Nonstiff ODE Solver in C," *Comput. Phys.*, 10:138–143, 1996. http://dx.doi.org/10.1063/1.4822377

[Duarte2015] M. Duarte. "Notes on Scientific Computing for Biomechanics and Motor Control," GitHub repository, 2015. https://GitHub.com/demotu/BMC

[Frenklach2007] M. Frenklach. "Transforming data into knowledge—Process Informatics for combustion chemistry," *Proc. Combust. Inst.*, 31:125–140, 2007. https://dx.doi.org/10.1016/j.proci.2006.08.121

[Goodwin2016] D. G. Goodwin, H. K. Moffat, and R L. Speth. "Cantera: An object-oriented software toolkit for chemical kinetics, thermodynamics, and transport processes," Version 2.2.1, 2016. http://www.cantera.org

[Grecco2016] H. E. Grecco. Pint version 0.7.2, GitHub repository, 2016. https://GitHub.com/hgrecco/pint

[Haas2009] F. M. Haas, M. Chaos, F. L. Dryer. "Low and intermediate temperature oxidation of ethanol and ethanol–PRF blends: An experimental and modeling study," *Combust. Flame*, 156:2346–2350, 2009. http://dx.doi.org/10.1016/j.combustflame.2009.08.012

[Hanson2014] R. K. Hanson, D. F. Davidson. "Recent advances in laser absorption and shock tube methods for studies of combustion chemistry," *Prog. Energy. Comb. Sci.*, 44:103–14, 2014. http://dx.doi.org/10.1016/j.pecs.2014.05.001

[Hindmarsh2005] A. C. Hindmarsh, P. N. Brown, K. E. Grant, S. L. Lee, R. Serban, D. E. Shumaker, and C. S. Woodward. "SUNDIALS: Suite of nonlinear and differential/algebraic equation solvers," *ACM Trans. Math. Software.*, 31:363–396, 2005. http://dx.doi.org/10.1145/1089014.1089020

[Huff2015] K. Huff and X. Wang. PyRK v0.2, Figshare, Feb 2015. http://dx.doi.org/10.6084/m9.figshare.2009058

[Huff2015b] K. Huff. "PyRK: A Python Package For Nuclear Reactor Kinetics," *Proceedings of the 14th Python in Science Conference*, 87–93, 2015. Editors: K. Huff and J. Bergstra.

[Jones2001] E. Jones, T. Oliphant, P. Peterson, et al. "SciPy: Open source scientific tools for Python," 2001–. http://www.scipy.org/

[Kee1996] R. J. Kee, F. M. Rupley, E. Meeks, and J. A. Miller. "CHEMKIN-III: A FORTRAN chemical kinetics package for the analysis of gas-phase chemical and plasma kinetics," Sandia National Laboratories Report SAND96-8216, May 1996. http://dx.doi.org/10.2172/481621

[Krekel2016] H. Krekel. pytest version 2.9.1, GitHub repository, 2016. https://github.com/pytest-dev/pytest/

[Niemeyer2016] K. E. Niemeyer. "An autoignition performance comparison of chemical kinetics models for *n*-heptane," Spring 2016 Meeting of the Western States Section of the Combustion Institute, Seattle, WA, USA. 21–22 March 2016. https://dx.doi.org/10.6084/m9.figshare.3120724

[Niemeyer2016b] K. E. Niemeyer. PyTeCK version 0.1.0, Zenodo, 2016. https://dx.doi.org/10.5281/zenodo.57565

[Niemeyer2016c] K. E. Niemeyer. "Selected hydrogen chemical kinetic models," figshare, 2016. https://dx.doi.org/10.6084/m9.figshare.3482906.v1

[Niemeyer2016d] K. E. Niemeyer. "Hydrogen shock tube ignition dataset," figshare, 2016. https://dx.doi.org/10.6084/m9.figshare.3482918.v1

[Olm2014] C. Olm, I. G. Zsely, R. Pálvölgyi, T. Varga, T. Nagy, H. J, Curran, and T. Turányi. "Comparison of the performance of several recent hydrogen combustion mechanisms," *Combust. Flame* 161:2219–34, 2014. http://dx.doi.org/10.1016/j.combustflame.2014.03.006

[Olm2015] C. Olm, I. G. Zsely, T. Varga, H. J. Curran, and T. Turányi. "Comparison of the performance of several recent syngas combustion mechanisms," *Combust. Flame* 162:1793–812, 2015. http://dx.doi.org/10.1016/j.combustflame.2014.12.001

[Petersen2009] E. L. Petersen, M. Lamnaouer, J. de Vries, H. J. Curran, J. M. Simmie, M. Fikri, et al. "Discrepancies between shock tube and rapid compression machine ignition at low temperatures and high pressures," *Shock Waves*, 1:739–44, 2009. http://dx.doi.org/10.1007/978-3-540-85168-4_119

[PrIMe2016] "Process Informatics Model," http://primekinetics.org. Accessed: 29-05-2016.

[Sung2014] C. J. Sung, H. J. Curran, "Using rapid compression machines for chemical kinetics studies," *Prog. Energy Comb. Sci.*, 44:1–18, 2014. http://dx.doi.org/10.1016/j.pecs.2014.04.001

[Travis2016] Travis-CI. "travis-ci/travis-api," GitHub repository. Accessed: 30-May-2016. https://github.com/travis-ci/travis-api

[vanderWalt2011] S. van der Walt, S. C. Colbert, and G. Varoquaux. "The NumPy Array: A Structure for Efficient Numerical Computation," *Comput. Sci. Eng.*, 13:22–30, 2011. https://dx.doi.org/10.1109/MCSE.2011.37

[Varga2015a] T. Varga, T. Turányi, E. Czinki, T. Furtenbacher, and A. G. Császár. "ReSpecTh: a joint reaction kinetics, spectroscopy, and thermochemistry information system," Proceedings of the 7th European Combustion Meeting, Budapest, Hungary. 30 March–2 April 2015. http://www.ecm2015.hu/papers/P1-04.pdf

[Varga2015b] T. Varga. "ReSpecTh Kinetics Data Format Specification v1.0," 25 March 2015. http://respecth.hu/

[You2012] X. You, A. Packard, M. Frenklach. "Process Informatics Tools for Predictive Modeling: Hydrogen Combustion," *Int. J. Chem. Kinet.*, 44:101–116, 2012. https://dx.doi.org/10.1002/kin.20627

Linting science prose and the science of prose linting

Michael D. Pacer[‡*], Jordan W. Suchow[‡]

https://youtu.be/S55EFUOu4O0

Abstract—The craft of writing is hard despite the abundance of thoughtful advice available in usage guides and other sources. This is partly a problem of medium: amassing advice is not enough to improve writing. Writing would thus benefit if our collective knowledge about best practices in writing were extracted and transformed into a medium that makes the knowledge more accessible to authors.

We built Proselint, a Python-based linter for English prose that identifies violations of style and usage guidelines. Proselint is open-source software released under the BSD license and is compatible with Pythons 2 and 3. It runs as a command-line utility or as a text-editor plugin. Proselint's modules address redundancy, jargon, illogic, clichés, unidiomatic vocabulary, sexism, inconsistency, misuse of symbols, malapropisms, oxymorons, security gaffes, hedging, apologizing, and pretension. Furthermore, Proselint is extensible, enabling creation of domain-specific modules and implementation of house style guides.

Proselint can be seen as both a language tool for scientists and a tool for language science. On the one hand, Proselint can help scientists communicate their ideas to each other and to the public by improving their writing. On the other hand, scientists can use Proselint to measure language usage, to provide style- and usage-based features for tasks such as authorship identification, and to explore the factors that make a linter useful (e.g., a low false discovery rate).

Index Terms—linters, writing tools, copyediting

The problem

Writing is hard even for the best writers, and it's not for lack of good advice — a tremendous amount of knowledge about the craft is strewn across usage guides, dictionaries, technical manuals, essays, pamphlets, websites, and the hearts and minds of great authors and editors. Consider *Garner's Modern English Usage*, an authoritative usage guide with 11,000 entries covering a broad range of advice that can help writers produce clear and idiomatic prose [Gar16]. Or consider the *Federal Plain Language Guidelines*, a guide created by employees of the U.S. federal government to promote writing that is clear, concise, and well-organized [Pla11]. Professional conferences such as the annual meeting of the American Copy Editors Society are dedicated to sharing knowledge about editing prose. And within the academy, organizations such as the American Psychological Association publish manuals whose guidance on style has been adopted as a standard [Ass94].

* *Corresponding author: mpacer@berkeley.edu*
‡ *University of California, Berkeley*

Advice on writing touches upon everything from superficial conventions to the deepest reflections of our society and its attitudes. For example, advice concerning the preferred forms of words such as *connote* (vs. *connotate*) may help to prune needless variants in spelling, but is unlikely to affect the reader's understanding of the text and its author. In contrast, advice concerning needlessly gendered language (*woman scientist*, *policeman*) helps to eliminate terms that may perpetuate social inequality [MS01], [Phi04].

Amassing a pile of advice is not enough to make writing better. This is because advice, though it may be principled, thoughtful, and worth following, is hard to apply in new settings once it has been learned [AI00]. Thus even if an author could absorb all the knowledge contained in extant sources of advice on writing, the author would still face the problem of recalling and systematically applying that knowledge during the acts of writing and editing. Furthermore, developing a new habit (linguistic or otherwise) is slow, costly, and effortful [FH10], causing errors to appear even if the author knows the rules.

Today, an author who wishes to improve a piece of writing by applying the collective wisdom of experts must rely on indirect means. Publishers often use a division of labor in which dedicated staff copyedit a piece to their satisfaction. For example, *The New Yorker* employs an editing team of fact checkers, editors, grammarians, and others [Nor]. Individuals often uses software-based tools such as spelling and grammar checkers that mark unrecognized words and purported violations of grammatical rules [HJM+82], [CM83], [Ver00], [Nab03], [Mił10], [PRR12].

Neither approach fully solves the problem of successful adoption of best practices in writing. Few people have the resources needed to outsource editing to external staff. Furthermore, doing so inevitably introduces a delay because copy editors must read the text carefully and are normally unavailable during the act of writing. By the time an editor's notes are received, then, an opportunity to strengthen the writer's craft has passed. Time-sensitivity exacerbates this problem because delays introduced by the editing process may diminish the communication's value. In contrast, software-based tools for writing are automated and relatively fast, but are typically incomplete, imprecise, or inaccessible (see *Proselint's approach*).

The solution

To solve this problem, we built Proselint, a real-time linter for English prose. A linter is a computer program that, like a spell checker, scans through a document and analyzes it, identifying problems with its syntax or style [Joh77]. Proselint identifies violations of expert-endorsed style and usage guidelines[1] and gently

alerts the writer of those violations as they are committed, an ideal opportunity to elicit long-term changes in behavior [FS57]. In doing so, Proselint gives voice to the experts while teaching at a speed and scale unreachable by humans.

Proselint is open-source software released under the BSD license and compatible with Pythons 2 and 3. It runs as a command-line utility or editor plugin for Sublime Text, Atom, Emacs, vim, etc. It outputs advice in JSON and the standard linting format (SLF), promoting integration with external services [Was90] and providing human-readable output. Proselint includes modules on a variety of usage problems, including redundancy, jargon, illogic, clichés, sexism, misspelling, inconsistency, misuse of symbols, malapropisms, oxymorons, security gaffes, hedging, apologizing, pretension, and more (see Tables 1 and 2 for a fuller listing).

Proselint is both a language tool for scientists and a tool for language science. On the one hand, it can help scientists communicate their ideas to each other and to the public by improving their writing. On the other hand, scientists can use Proselint to study language and linting.

A language tool for scientists

Scientists use the written word to communicate to each other and to the public. Proselint improves writing across a number of dimensions relevant to science communication, including consistency in terminology & typography, concision, and elimination of redundancy. For example, Proselint detects the letter x used in place of the multiplication symbol × (e.g., 1440 x 900), misspecified p values resulting from data-analysis software that truncates small numbers (e.g., $p = 0.00$), and colloquialisms that obscure the mechanisms of science-based technology (e.g., "lie detector test" for the polygraph machine, which measures arousal, not lying per se).

A tool for language science

Linguistics is largely descriptivist, tending to describe language as it is used rather than prescribe how it ought to be used [Gar16]. Errors are considered mostly in the context of language learning (especially children's) because those errors reveal the structure of the language-learning mechanism (see, e.g., overregularization by young English speakers [MPU+92]). Though linting prose is implicitly prescriptivist because its detection of norm violations presupposes the existence of norms [Gar16], even so, language science can benefit from Proselint's advice without making normative claims. Linguists can use Proselint to detect patterns in usage and style in corpora of written text, to identify authors by their usage, and to enrich standard Natural Language Processing (NLP) techniques with features beyond word frequencies and syntactic structures [BKL09].

The advice

Proselint is built around advice derived from works by Bryan Garner, David Foster Wallace, Chuck Palahniuk, Steve Pinker, Mary Norris, Mark Twain, Elmore Leonard, George Orwell, Matthew Butterick, William Strunk, E.B. White, Philip Corbett, Ernest Gowers, and the editorial staff of the world's finest literary magazines and newspapers, among others.[2]

Our standard for including a new rule is that it should be accompanied by a citation to a recognized expert on language usage who has defined the rule clearly. Though we have no explicit criteria for what makes a citation appropriate, in practice we have given greater weight to works from well-established publishers and those widely cited as reliable sources of advice. The choice of which rules to implement is ultimately a question of feasibility of implementation, utility, and preference. Our guiding preference is to make Proselint widely useful by default. In the case of unresolved conflicts between advice from multiple sources, our default is to exclude all forms of the advice because we find it unreasonable to hold users to a higher standard than we hold the experts, at least one of whom supports the user's choice. Because we aim for excellent defaults without hampering customization, Proselint can be extended by adding new rules or filtered by excluding existing rules through a configuration file.

Tables 1 and 2 list much of the advice that Proselint currently implements. That advice is organized into modules.

Rule modules

Proselint's rules are organized into modules that reflect the structure of usage guides [Gar16]. For example, the `terms` module encourages expressive vocabulary by flagging use of unidiomatic and generic terms. The module has submodules for categories of terms found as entries in usage guides. The submodule `terms.venery` pertains to venery terms, which arose from hunting tradition and describe groups of animals of a particular species — a *pride* of lions or an *unkindness* of ravens. Similarly, the submodule `terms.denizen_labels` pertains to demonyms, which are used to describe people from a particular place — *New Yorkers* (New York), *Mancunians* (Manchester), or *Novocastrians* (Newcastle).

Organizing rules into modules is useful for two reasons. First, it allows for a logical grouping of similar rules, which often require similar computational machinery to implement. Second, it allows users to include and exclude rules at a higher level of abstraction than the individual word or phrase.

Converting a rule to code: rule templates

Suppose a developer wanted to implement the following entry from *Garner's Modern English Usage* as a rule in Proselint:

> **decimate.** Originally this word meant "to kill one in every ten," but this etymological sense, because it's so uncommon, has been abandoned except in historical contexts. Now *decimate* generally means "to cause great loss of life; to destroy a large part of." ... In fact, though, the word might justifiably be considered a SKUNKED TERM. Whether you stick to the original one-in-ten meaning or use the extended sense, the word is infected with ambiguity. And some of your readers will probably be puzzled or bothered. [Gar16]

In general, a rule's implementation need only be a function that takes in a string of text, applies logic identifying whether the rule has been violated, and then returns a value identifying the violation in the correct format. Weak requirements and Python's

1. Proselint differs from a spell-checker in that its recommendations do not specifically counter spelling errors, but rather errors of style and usage. The two occasionally overlap, e.g. in the malapropism "attacking your voracity", where it is not that "voracity" is a spelling error per se but that the appropriate word is its phonetic neighbor "veracity". Compare this to "attacking your verqcity", almost certainly a typo.

2. Proselint has not been endorsed by these individuals; we have merely implemented their words in code.

ID	Description
airlinese.misc	Avoiding jargon of the airline industry
annotations.misc	Catching annotations left in the text
archaism.misc	Avoiding archaic forms
cliches.misc	Avoiding clichés
consistency.spacing	Consistent sentence spacing
consistency.spelling	Consistent spelling
corporate_speak.misc	Avoiding corporate buzzwords
cursing.filth	Avoiding cursing
cursing.nfl	Avoiding words banned by the NFL
dates_times.am_pm	Using the right form for time
dates_times.dates	Stylish formatting of dates
hedging.misc	Not hedging
hyperbole.misc	Not being hyperbolic
jargon.misc	Avoiding miscellaneous jargon
lexical_illusions.misc	Avoiding lexical illusions
links.broken	Linking only to existing sites
malapropisms.misc	Avoiding common malapropisms
misc.apologizing	Being confident
misc.back_formations	Avoiding needless backformations
misc.bureaucratese	Avoiding bureaucratese
misc.but	Avoiding starting a par. with "But..."
misc.capitalization	Capitalizing correctly
misc.chatspeak	Avoiding lolling and other chatspeak
misc.commercialese	Avoiding commerical jargon
misc.currency	Avoiding redundant currency symbols
misc.debased	Avoiding debased language
misc.false_plurals	Avoiding false plurals
misc.illogic	Avoiding illogical forms
misc.inferior_superior	Superior to, not than
misc.latin	Avoiding overuse of Latin phrases
misc.many_a	Many a singular
misc.metaconcepts	Avoiding overuse of metaconcepts
misc.narcisissm	Talking about the subject, not its study
misc.phrasal_adjectives	Hyphenating phrasal adjectives
misc.preferred_forms	Miscellaneous preferred forms

TABLE 1
What Proselint checks.

ID	Description
misc.pretension	Avoiding being pretentious
misc.professions	Calling jobs by the right name
misc.punctuation	Using punctuation assiduously
misc.scare_quotes	Using scare quotes only when needed
misc.suddenly	Avoiding the word suddenly
misc.waxed	Waxing poetic
misc.whence	Using "whence"
mixed_metaphors.misc	Not mixing metaphors
mondegreens.misc	Avoiding mondegreens
needless_variants.misc	Using the preferred form
nonwords.misc	Avoid using nonwords
oxymorons.misc	Avoiding oxymorons
psychology.misc	Avoiding misused psychological terms
redundancy.misc	Avoid redundancy & saying things twice
redundancy.ras_syndrome	Avoiding RAS syndrome
skunked_terms.misc	Avoid using skunked terms
spelling.able_atable	-able vs. -atable
spelling.able_ible	-able vs. -ible
spelling.athletes	Spelling of athlete names
spelling.em_im_en_in	-em vs. -im and -en vs. -in
spelling.er_or	-er vs. -or
spelling.in_un	in- vs. un-
spelling.misc	Spelling words corectly
security.credit_card	Keeping credit card numbers secret
security.password	Keeping passwords secret
sexism.misc	Avoiding sexist language
terms.animal_adjectives	Animal adjectives
terms.denizen_labels	Calling denizens by the right name
terms.eponymous_adjs	Calling people by the right name
terms.venery	Call groups of animals by the right name
typography.diacritics	Using dïacríticâl marks
typography.exclamation	Avoiding overuse of exclamation
typography.symbols	Using the right symbols
uncomparables.misc	Not comparing uncomparables
weasel_words.misc	Avoiding weasel words

TABLE 2
What Proselint checks (cont.).

expressiveness allow developers to build detectors for all computable usage and style requirements, but provide little guidance for implementing new rules.

To provide guidance for implementing new rules, we wrote helper functions that follow the protocol and provide some common logical forms of rules. These include checking for the existence of a given word, phrase, or pattern (`existence_check()`); for intra-document consistency in usage (`consistency_check()`); and for use of a word's preferred form (`preferred_forms_check()`).

The entry on *decimate* bans a word and so can be implemented using the `existence_check` template:

```
def check_for_decimate(text):
    err = "skunked_terms.decimate"
    msg = (u"'{}' is a skunked term -- impossible to
           "use without someone taking issue. Find"
           "another way to say it")
    regex = "decimat(?:e|es|ed|ing)?"
    return existence_check(
        text, [regex], err, msg, join=True)
```

First the function defines an error code, an error message, and a regular expression that matches the word *decimate* in its various forms. Then it applies the existence check.

Using Proselint

Installation

Proselint is available on the Python Package Index and can be installed using pip:

```
pip install proselint
```

Alternatively, developers can retrieve the Git repository from GitHub (https://github.com/amperser/Proselint) and then install the software using setuptools:

```
pip install --editable
```

Command-line utility

Proselint is a command-line utility that reads in a text file:

```
proselint text.md
```

Running this command prints a list of suggestions to stdout, one per line. The GNU Error Message Formatting standard [S+16] is the basis for the format of displaying these suggestions. We further require that the error code (here, the `check_name`) is separated from the error message by a space. Because this format is used by many linters, we call it the Standard Linting Format (SLF). An SLF-formatted suggestion has the form:

```
text.md:<line>:<column>: <check_name> <message>
```

For example,

```
text.md:0:10: skunked_terms.misc 'decimate' is ...
a skunked term -- impossible to use without ...
someone taking issue. Find another way to say it."
```

This message suggests that, at column 10 of line 0, the module `skunked_terms.misc` detected the presence of the skunked term *decimate*. The command-line utility can instead print the list of suggestions in JSON through the `--json` flag. In this case, the output is considerably richer:

```
{
    // The check originating this suggestion
    "check": "uncomparables.misc",

    // The line where the error starts
    "line": 1,

    //The column where the error starts
    "column": 1,

    // Index in the text where the error starts
    "start": 1,

    // the index in the text where the error ends
    "end": 18,

    // start - end
    "extent": 17,

    // Message describing the advice
    "message": "Comparison of an uncomparable: ...
    'very unique\n' is not comparable.",

    // Possible replacements
    "replacements": null,

    // Importance("suggestion", "warning", "error")
    "severity": "warning"
}
```

Text editor plugins

Proselint is available as a plugin for popular text editors, including Emacs, vim, Sublime Text, and Atom. Embedding linters within the tools that people already use to write removes a barrier to adoption the linter and thereby promotes adoption of best practices in writing [Was90].

Proselint's approach

In the following sections, we describe Proselint's approach and its greatest points of departure from previous attempts to lint prose. As part of this analysis, we curated a list of known tools for automated language checking. The dataset contains the name of each tool, a link to its website, and data about its basic features, including languages and licenses (link). The tools are varied in their approaches and coverage, but typically focus on grammar versus usage and style; are unsystematic in choosing sources of advice; or have been abandoned. In general, we regard the tools as being imprecise, incomplete, and inaccessible:

Imprecise. Even the best software-based tools for editing are riddled with false positives. We evaluated many of the tools in our dataset on an earlier version of the corpus. Proselint's false discovery rate of 1 false positive to 10 true positives was 20× better than the next best tool, Microsoft Word, which had a false discovery rate of 2 false positives to 1 true positive.

Incomplete. All software-based tools for editing are incomplete; not one frees our collective knowledge about best practices in writing from its bindings. Completion is likely an unattainable goal, which inspires Proselint's open-source, community-participation model.

Inaccessible. Many existing tools are inaccessible because they cost money, are closed source, or are inextensible. Thus we designed Proselint to be free, open source, and extensible.

What to check: usage, not grammar

Proselint does not detect grammatical errors because it is both too easy and too hard:

Detecting grammatical errors is too easy in the sense that most native speakers can readily identify and easily fix them. The errors that leave the greatest negative impression in the reader's mind are often glaring to native speaker. On the other hand, more subtle errors, such as a disagreement in number set apart by a long string of intermediary text, escapes even a native speaker's notice.

Detecting grammatical errors is too hard in the sense that its most general form is AI-hard, requiring at least human-level artificial intelligence and a native speaker's ear [Yam13]. Modern NLP techniques that detect grammatical errors are unavoidably statistical and produce many false positives [BKL09] [LCGT10]. This is in part because syntax parsers used in grammatical error detection must tolerate grammatical errors, a problem that is compounded in writing by English-language learners [LCGT10]. Once a grammatical error has been detected, determining the correct replacement hinges on the intended meaning. Occasionally, the intended meaning will determine even *whether* a grammatical error is present: e.g., is "Man bites dog" a headline about canine aggression, or are the subject and object swapped in error? In the general case, the problem of determining the intended meaning of a sentence is AI-hard [Yam13].

Instead of focusing on grammatical errors, Proselint addresses errors of usage and style.

Published expertise as primary sources

People have such strong shared intuitions about grammar that a common experimental measure in linguistics is the grammaticality of a sentence as measured by the intuitions of native speakers [Kel00]. But style and usage inspire a multitude of intuitions. Authors of usage guides have done much of the work of hashing out these conflicting intuitions to arrive at sensible everyday advice [Gar16]. Proselint thus defers to these experts, and in doing so embodies our collective understanding about the craft of writing with style.

Levels of difficulty

In a loose analogy to Chomsky's hierarchy of formal grammars [Cho56], usage errors vary in the difficulty of detecting and correcting them:

1) AI-hard

2) NLP, beyond state-of-the-art
3) NLP, state-of-the-art
4) Syntax-dependent rules
5) Regular expressions
6) One-to-one replacement rules.

At the lowest levels of the hierarchy are usage errors that a linter can reliably detect and correct through one-to-one replacement rules. At the highest levels are usage errors whose detection and correction are such hard computational problems that it would require at least human-level intelligence to solve in the general case, if a solution is possible at all [Yam13]. Consider usage errors pertaining to placement of the word *only*, which depends on the intended meaning. For example, in "John hit Peter in his only nose", is the *only* misplaced or is it unusual that Peter has only one nose? Usage errors at this highest level of the hierarchy are hard to detect without introducing false positives and determining the correct replacement requires understanding the intended meaning. Development of Proselint begins at the lowest levels of the hierarchy and builds upwards.

Signal detection theory and the lintscore

Any new tool, for language or otherwise, faces a challenge to its adoption: it must demonstrate that the utility the tool provides outweighs the cost of learning to use it [Was90]. The utility of a prose linter comes in part from its ability to detect usage and style errors. Each issue flagged might be an error, but it might instead be a false positive. Let T be the number of true errors and F be the number of false positives, thus making $T+F$ the total number of flags raised by the tool. An approach that attempts to maximize T by flagging many errors without adequately considering F will identify many genuine errors, but raise so many false positives that writers must evaluate each proposed error.

With Proselint, we aim for a tool precise enough that users can adopt its recommendations unquestioningly and still come out ahead. To achieve this, we penalize the number of false positives F by evaluating Proselint in terms of its *empirical lintscore*. The lintscore gives one point for every true positive T and penalizes on the basis of the false discovery rate $\alpha = \frac{F}{T+F}$. The lintscore is given by

$$l(T,F;k) = T(1-\alpha)^k,$$

where the parameter :math:kgeq1‘ controls the strength of the $1-\alpha$ penalty. Notably, the lintscore does not reflect the number of true and false negatives; we reason that it is more important to be quiet and authoritative than to be loud and risk being untrustworthy (cf. the metrics discussed in [CDIT12]).

The lintscore can be computed exactly if an evaluator can classify each error flagged by the linter as a true or false positive. However, many corpora are large enough to preclude this kind of exhaustive assessment. In these cases, the lintscore can be estimated from the total number of issues flagged and an estimate of the false discovery rate.

Note that the lintscore is not a readability metric because it evaluates linters, not prose. Given a set of documents, signal detection theory makes it possible to estimate a linters' trustworthiness through the lintscore.

Speed via Memoization

Proselint must be efficient for use as a real-time linter. Avoiding redundant computation by storing the results of expensive function calls ("memoization") improves efficiency. Because most paragraphs do not change from moment to moment during editing of a sizable document, memoizing Proselint's output over paragraphs and recomputing only when a paragraph has changed (otherwise returning the memoized result) reduces the total amount of computation and thus improves the running time.

A proof of concept

As a proof of concept, we used Proselint to make contributions to several documents. These include the White House's Federal Source Code Policy; The Open Logic Project textbook on advanced logic; Infoactive's Data + Design book; and many of the other papers submitted to SciPy 2016. In addition, we evaluated Proselint's false discovery rate on a corpus of essays from well-edited magazines such as *Harper's Magazine*, *The New Yorker*, and *The Atlantic* (full list). We then measured the lintscore. Because the essays included in our corpus were edited by a team of experts, we expect Proselint to remain mostly silent. By design, Proselint should comment only on the rare error that slips through unnoticed by the editors or, more commonly, on finer points of usage, about which the experts sometimes disagree. When run over v0.1.0 of our corpus, we achieved a lintscore (k = 2) of 98.8.

Future development and possible applications

We see a number of directions for future development of Proselint that improve the tool and its utility for science:

Context-sensitive rule application and machine learning

Many rules apply better to some kinds of documents than to others. For example, in most cases *extendable* is preferable to *extensible*, but in software development the opposite is true. Applying these rules without consideration of the context will systematically introduce false positives.

Silencing rules that are predicted to be irrelevant because of the context allows a greater variety of rules to be included without introducing false positives. Consider the advice that, when specifying a decade, an apostrophe is unnecessary: Eisenhower was president in the 50s, not the 50's. However, not all instances of *50's* are problematic: one can validly write *50's manager* to refer to 50's manager without making a usage error about decades. To account for this context sensitivity, Proselint detects whether a document's topic is 50 Cent, identifying *50's* as a usage error only when the topic is not detected.

The 50 Cent topic detector was hand-crafted in the fashion of expert knowledge systems [Jac86]. Machine-learning techniques for identifying the topic of a document (e.g., topic models [BL09]) can generalize this ability and will be crucial to safely growing Proselint's coverage of usage errors. Once incorporated, extending this to hierarchical nonparametric topic models will enable document sub-structure to be taken into account as a form of context [BGJ10].

Evaluating linters by testing on multiple corpora

In our internal evaluations of Proselint, we calculate the empirical lintscore manually on a corpus of professionally edited documents, which presumably have few errors. This efficiently alerts us to false positives that are introduced by new rules, but tells us little about its performance in other settings. A major improvement would be to compute the lintscore on corpora such as student essays, which are more likely to have true positives and will thus

improve our estimates of Proselint's positive utility for a more typical user.

Corpora of documents drawn from different content-based categories (technical papers, scientific articles, software documentation, fiction, journalism, etc.) will help in evaluating Proselint's performance in evaluating prose from different fields. Certain rules may be relevant to some fields more than others and testing with diverse corpora will ensure that Proselint can be used by a diverse range of individuals. Furthermore, this will allow us to learn which rule sets are relevant in which contexts.

Observing how a document is modified in accordance with Proselint's suggestions affords new opportunities for evaluation of Proselint, tracking the acceptance of its advice and any effects on the rate of new errors introduced between drafts.

File formats and markup languages for documents (e.g, reStructuredText, LaTeX, Markdown, HTML, etc.) often rely on syntactical conventions that Proselint falsely identifies as errors. Similar concerns arise for documentation written as docstrings or code comments in a variety of programming languages. Corpora focusing on individual formats and languages will aid in identifying and filtering these errors, enabling development targeted at addressing these problems.

Stylometrics and machine learning

The field of stylometrics has extensively studied the problem of identifying the authors of documents [ZLCH06]. Many of these studies focus on the relative frequencies with which individual words are used, especially function words. For example, Mosteller & Wallace inferred the authorship of twelve essays in the *Federalist Papers* on the basis of the frequency of common function words such as *to* and *by* [MW63]. Proselint provides new measures that could be used to improve this kind of stylometric analysis.

Several applications follow from authorship identification:

One application uses Proselint to detect ghost-written documents, which could also have benefits for identifying academic dishonesty (e.g., purchasing and selling of ghost-written essays). This application assumes that there is a ground-truth corpus with samples of the author's writing. On the other hand, someone may be able to use Proselint to *escape* identification by avoiding features that distinguish the author's writing from those of others.

A second application inverts and generalizes the process of identifying authors by selectively introducing, changing, or removing usage choices to obfuscate or encrypt messages. With some modifications and a protocol for establishing usage-based keys, Proselint could become a system for designing content-aware steganographic systems that convey hidden messages through their choice of words and style [BK06]. Encryption would require modifying the Proselint infrastructure to identify when more than one acceptable choice exists.

The errors Proselint can detect are rare compared to the typical linguistic features used in stylometry [ZLCH06], [MW63], [Rud97]. Sparse measures pose difficulty for methods like those in Mosteller & Wallace (1963) [MW63]. Machine-learning techniques for inferring identity from sparse data will thus be particularly applicable. Furthermore, this endeavor will benefit from an approach that considers the cross product of authors and topics [RZGSS04].

Automated usage and style metrics

Readability metrics such as the Flesch–Kincaid Grade Level and the Gunning fog index do not capture usage and style because they measure reading ease rather than conventionality [Fle48]. Proselint could be used to create automated metrics for the consistency and stylishness of prose. Such metrics may also find use as part of automated essay-grading tools [VNC03].

Tracking historical trends in usage

An application of Proselint as a tool for language science is in tracking historical trends in usage. Corpora such as Google Books have been useful for measuring changes in the prevalence of words and phrases over several hundred years [MSA+11]. Our tool can be used in a similar way because it provides a feature set for usage. For example, one might study the prevalence of airlinese (including, e.g., use of "momentarily" to mean "in a moment", as in the phrase "we are taking off momentarily") and its alignment with the rise of that industry.

An unsolved problem: foreign languages

We have no immediate plans for extending Proselint to other languages. This is in part because building a linter for style and usage errors in both American and British English is challenging enough for a native speaker, and in part because attempting to build a linter for languages in which the creators lack fluency would seem to be an exercise in folly. An open problem is how to extend Proselint to become a universal linter for prose.

Missing corpora

To evaluate Proselint's false discovery rate, we built a corpus of text from well-edited magazines believed to contain low rates of usage errors. In the course of assembling this corpus, we discovered a lack of annotated corpora that provide false discovery rates for style and usage violations[3]. The Proselint testing framework is an excellent opportunity to develop such a corpus. Unfortunately, because our current corpus derives from copyrighted work, it cannot be released as part of open-source software. Developing an open-source corpus of style and usage errors will be necessary if these tools are to be made available for NLP research outside internal testing of Proselint.

A critique of normativity in prose styling, and a response

One critique of Proselint [hac] is a concern that introducing any kind of linter-like process to the act of writing diminishes the ability for authors to express themselves creatively. These arguments suggest that authors will find themselves limited by the linter's rules and that, as a result, this will have a shaping or homogenizing effect on language.

In response to this critique, we note that our goal is not to homogenize text for the sake of uniformity (though perhaps there is value there, too), but rather to detect instances of language use that have been identified by experts as problematic. Creative use of language is not flagged unless it has been previously identified as problematic, furthering our aim of a quiet and authoritative tool. And even an author who intentionally flouts conventions for creative reasons will benefit from a thorough understanding of them [Bri04].

3. Editor [edi] has built a corpus which compares the performance of various grammar checkers. Their corpus contains "real-world examples of grammatical mistakes and stylistic problems taken from published sources". A corpus made of errors will maximize true positives, but misestimate false discovery rates in real-world documents. Their corpus is not publicly available, and they do not provide a standard format for describing corpora annotated with false positives and negatives.

Furthermore, technical writing of all kinds is often characterized by consistent language use and precise terminology. Even an author who views all writing as inextricably creative must sometimes direct that creativity toward a particular aim. Software documentation, technical manuals, and legal briefs, and pedagogical writing all feature this need and are improved when the author follows the conventions of a field.

Lastly, science demands consistency to promote clarity and replication. At the same time, scientists are in the business of expressing ideas that challenge even the greatest of minds, and their success depends on conveying those ideas to people who then use the ideas in their own work. When an idea is hard to grasp, simplicity and clarity will further its proliferation.

Contributing to Proselint

The primary avenue for contributing to Proselint is by contributing code to its GitHub repository. In particular, we have developed an extensive set of Issues that range from trivial-to-fix bugs to lofty features whose addition are entire research projects in their own right. To merit inclusion in Proselint, contributed rules should be accompanied by a citation to a recognized expert on language usage who has defined the rule clearly. This is not because language experts are the only arbiters of language usage, but because our goal is explicitly to aggregate best practices as put forth by the experts.

A secondary avenue for contributing to Proselint is through discovery of false positives: instances where Proselint flags well-formed idiomatic prose as containing a usage error. In this way, people with expertise in editing, language, and quality assurance can make a valuable contribution that directly improves the metric we use to gauge success.

Acknowledgments

Proselint is supported in part by the Berkeley Center for Technology, Society and Policy through the CTSP Fellows program, specifically for applying it to the problem of improving governmental communications as laid out in the Federal Plain Language Guidelines. We thank several reviewers who gave feedback on the manuscript, including Dan Lewis, David Lippa, Scott Rostrup, and Stéfan van der Walt. This work was presented as a talk at *SciPy* 2016 (YouTube).

References

[AI00] L. Argote and P. Ingram. Knowledge transfer: A basis for competitive advantage in firms. *Organizational Behavior and Human Decision Processes*, 82:150–169, 2000.

[Ass94] American Psychological Association. *Publication Manual of the American Psychological Association*. American Psychological Association Washington, 1994.

[BGJ10] David M Blei, Thomas L Griffiths, and Michael I Jordan. The nested chinese restaurant process and bayesian nonparametric inference of topic hierarchies. *Journal of the ACM (JACM)*, 57(2):7, 2010.

[BK06] Richard Bergmair and Stefan Katzenbeisser. Content-aware steganography: about lazy prisoners and narrow-minded wardens. In *International Workshop on Information Hiding*, pages 109–123. Springer, 2006.

[BKL09] Steven Bird, Ewan Klein, and Edward Loper. *Natural Language Processing with Python*. O'Reilly Media, Sebastopol, CA, 2009. 504 pages.

[BL09] David M Blei and John D Lafferty. Topic models. *Text mining: classification, clustering, and applications*, 10(71):34, 2009.

[Bri04] Robert Bringhurst. *The Elements of Typographic Style. 3rd revision*. Canada, USA: Hartley & Marks, 2004.

[CDIT12] Martin Chodorow, Markus Dickinson, Ross Israel, and Joel R Tetreault. Problems in evaluating grammatical error detection systems. In *COLING 2012*, pages 611–628, 2012.

[Cho56] Noam Chomsky. Three models for the description of language. *IRE Transactions on Information Theory*, 2(3):113–124, 1956.

[CM83] Lorinda L Cherry and Nina H Macdonald. The unix writers workbench software. *Byte*, 8(10):241, 1983.

[edi] Comparing grammar checkers: Holding grammar scammers' feats to the fire. http://www.serenity-software.com/pages/comparisons.html. Accessed: 2016-07-11.

[FH10] B. J. Fogg and J. Hreha. Behavior wizard: a method for matching target behaviors with solutions. *International Conference on Persuasive Technology*, pages 117–131, 2010.

[Fle48] Rudolph Flesch. A new readability yardstick. *Journal of Applied Psychology*, 32(3):221, 1948.

[FS57] Charles B Ferster and Burrhus Frederic Skinner. Schedules of reinforcement. 1957.

[Gar16] Bryan Garner. *Garner's Modern English Usage*. Oxford University Press, 2016.

[hac] Hacker news: Proselint (proselint.com). https://news.ycombinator.com/item?id=11232882. Accessed: 2016-07-05.

[HJM+82] George E. Heidorn, Karen Jensen, Lance A. Miller, Roy J. Byrd, and Martin S Chodorow. The epistle text-critiquing system. *IBM Systems Journal*, 21(3):305–326, 1982.

[Jac86] Peter Jackson. Introduction to expert systems. 1986.

[Joh77] S. Johnson. Lint, a C program checker. *Computer Science Technical Report 65, Bell Laboratories*, December 1977.

[Kel00] Frank Keller. *Gradience in grammar: Experimental and computational aspects of degrees of grammaticality*. PhD thesis, 2000.

[LCGT10] Claudia Leacock, Martin Chodorow, Michael Gamon, and Joel Tetreault. Automated grammatical error detection for language learners. *Synthesis Lectures on Human Language Technologies*, 3(1):1–134, 2010.

[Mił10] Marcin Miłkowski. Developing an open-source, rule-based proofreading tool. *Software: Practice and Experience*, 40(7):543–566, 2010.

[MPU+92] Gary F Marcus, Steven Pinker, Michael Ullman, Michelle Hollander, T John Rosen, Fei Xu, and Harald Clahsen. Overregularization in language acquisition. *Monographs of the Society for Research in Child Development*, pages i–178, 1992.

[MS01] Casey Miller and Kate Swift. *The handbook of nonsexist writing*. iUniverse, 2001.

[MSA+11] Jean-Baptiste Michel, Yuan Kui Shen, Aviva Presser Aiden, Adrian Veres, Matthew K Gray, Joseph P Pickett, Dale Hoiberg, Dan Clancy, Peter Norvig, Jon Orwant, et al. Quantitative analysis of culture using millions of digitized books. *Science*, 331(6014):176–182, 2011.

[MW63] Frederick Mosteller and David L Wallace. Inference in an authorship problem: A comparative study of discrimination methods applied to the authorship of the disputed Federalist Papers. *Journal of the American Statistical Association*, 58(302):275–309, 1963.

[Nab03] Daniel Naber. *A rule-based style and grammar checker*. PhD thesis, 2003.

[Nor] Copy Editing at The New Yorker with Mary Norris. https://andyrossagency.wordpress.com/2009/09/20/. Accessed: 2016-07-11.

[Phi04] S. U. Philips. *Language and social inequality. A Companion to Linguistic Anthropology*. 2004.

[Pla11] Plain Language Action and Information Network. Federal plain language guidelines. http://www.plainlanguage.gov/howto/guidelines/bigdoc/fullbigdoc.pdf, 2011.

[PRR12] Fabrizio Perin, Lukas Renggli, and Jorge Ressia. Linguistic style checking with program checking tools. *Computer Languages, Systems & Structures*, 38(1):61–72, 2012.

[Rud97] Joseph Rudman. The state of authorship attribution studies: Some problems and solutions. *Computers and the Humanities*, 31(4):351–365, 1997.

[RZGSS04] Michal Rosen-Zvi, Thomas Griffiths, Mark Steyvers, and Padhraic Smyth. The author-topic model for authors and documents. In *Proceedings of the 20th Conference on Uncertainty in Artificial Intelligence*, pages 487–494. AUAI Press, 2004.

[S+16] Richard Stallman et al. GNU coding standards. https://www.gnu.org/prep/standards/html_node/Errors.html/, 2016. Accessed: 2016-07-18.

[Ver00] Alex Vernon. Computerized grammar checkers 2000: Capabilities, limitations, and pedagogical possibilities. *Computers and Composition*, 17(3):329–349, 2000.

[VNC03] Salvatore Valenti, Francesca Neri, and Alessandro Cucchiarelli. An overview of current research on automated essay grading. *Journal of Information Technology Education*, 2:319–330, 2003.

[Was90] Anthony I Wasserman. Tool integration in software engineering environments. In *Software Engineering Environments*, pages 137–149. Springer, 1990.

[Yam13] Roman V Yampolskiy. Turing test as a defining feature of ai-completeness. In *Artificial Intelligence, Evolutionary Computing and Metaheuristics*, pages 3–17. Springer, 2013.

[ZLCH06] Rong Zheng, Jiexun Li, Hsinchun Chen, and Zan Huang. A framework for authorship identification of online messages: Writing-style features and classification techniques. *Journal of the American Society for Information Science and Technology*, 57(3):378–393, 2006.

MDAnalysis: A Python Package for the Rapid Analysis of Molecular Dynamics Simulations

Richard J. Gowers[†††‡‡‡†], Max Linke[**†], Jonathan Barnoud[¶†], Tyler J. E. Reddy[§], Manuel N. Melo[¶], Sean L. Seyler[‡], Jan Domański[§], David L. Dotson[‡], Sébastien Buchoux[‖], Ian M. Kenney[‡], Oliver Beckstein[‡*]

https://youtu.be/zVQGFysYDew

Abstract—MDAnalysis (http://mdanalysis.org) is a library for structural and temporal analysis of molecular dynamics (MD) simulation trajectories and individual protein structures. MD simulations of biological molecules have become an important tool to elucidate the relationship between molecular structure and physiological function. Simulations are performed with highly optimized software packages on HPC resources but most codes generate output trajectories in their own formats so that the development of new trajectory analysis algorithms is confined to specific user communities and widespread adoption and further development is delayed. MDAnalysis addresses this problem by abstracting access to the raw simulation data and presenting a uniform object-oriented Python interface to the user. It thus enables users to rapidly write code that is portable and immediately usable in virtually all biomolecular simulation communities. The user interface and modular design work equally well in complex scripted work flows, as foundations for other packages, and for interactive and rapid prototyping work in IPython / Jupyter notebooks, especially together with molecular visualization provided by nglview and time series analysis with pandas. MDAnalysis is written in Python and Cython and uses NumPy arrays for easy interoperability with the wider scientific Python ecosystem. It is widely used and forms the foundation for more specialized biomolecular simulation tools. MDAnalysis is available under the GNU General Public License v2.

Index Terms—molecular dynamics simulations, science, chemistry, physics, biology

Introduction

Molecular dynamics (MD) simulations of biological molecules have become an important tool to elucidate the relationship between molecular structure and physiological function [DDG+12], [Oro14]. Simulations are performed with highly optimized software packages on HPC resources but most codes generate output trajectories in their own formats so that the development of new trajectory analysis algorithms is confined to specific user communities and widespread adoption and further development is delayed. Typical trajectory sizes range from gigabytes to terabytes so it is typically not feasible to convert trajectories into a range of different formats just to use a tool that requires this specific format. Instead, a framework is required that provides a common interface to raw simulation data. Here we describe the MDAnalysis library [MADWB11] that addresses this problem by abstracting access to the raw simulation data. MDAnalysis presents a uniform object-oriented Python interface to the user. Since its original publication in 2011 [MADWB11], MDAnalysis has been widely adopted and has undergone substantial changes. Here we provide a short introduction to MDAnalysis and its capabilities and an overview over recent improvements.

MDAnalysis was initially inspired by MDTools for Python (J.C. Phillips, unpublished) and MMTK [Hin00]. MDTools pioneered the key idea to use an extensible and object-oriented language, namely, Python, to provide a high-level interface for the construction and analysis of molecular systems for MD simulations. MMTK became a tool kit to build MD simulation applications on the basis of a concise object model of a molecular system. MDAnalysis was built on an object model similar to that of MMTK with a strong focus on providing universal high-level building blocks for the analysis of MD trajectories, but for a much wider range of formats than previously available. MDAnalysis has been publicly available since January 2008 and is one of the longest actively maintained Python packages for the analysis of molecular simulations. Since then many other packages have appeared that primarily function as libraries for providing access to simulation data from within Python. Three popular examples are PyLOOS [RLG14], mdtraj [MBH+15], and pytraj [NRSC16]. PyLOOS [RLG14] consists of Python bindings to the C++ LOOS library [RG09]; in order to aid novice users, LOOS also provides about 140 small stand-alone tools that each focus on a single task. mdtraj [MBH+15] is similar to MDAnalysis in many aspects but focuses even more on being a light-weight building block for other packages; it also includes a number of innovative performance optimizations. pytraj [NRSC16] is a versatile Python frontend to the popular and powerful `cpptraj` tool [RCI13] and is particularly geared towards users of the Amber MD package [CCD+05]. These three packages and MDAnalysis have in common that they are built on an object model of the underlying data (such as groups of particles or a trajectory), use compiled code in C, C++ or Cython to accelerate time critical bottlenecks, and have a "Pythonic" user interface. LOOS and MDAnalysis share a similar object-oriented

† *These authors contributed equally.*
†† *University of Manchester, Manchester, UK*
‡‡ *University of Edinburgh, Edinburgh, UK*
** *Max Planck Institut für Biophysik, Frankfurt, Germany*
¶ *University of Groningen, Groningen, The Netherlands*
§ *University of Oxford, Oxford, UK*
‡ *Arizona State University, Tempe, Arizona, USA*
‖ *Université de Picardie Jules Verne, Amiens, France*
* *Corresponding author: oliver.beckstein@asu.edu*

philosophy in their user interface design. In contrast, mdtraj and pytraj expose a functional user interface. Both approaches have advantages and the existence of different "second generation" Python packages for the analysis of MD simulations provides many good choices for users and a fast moving and stimulating environment for developers.

Overview

MDAnalysis is specifically tailored to the domain of molecular simulations, in particularly in biophysics, chemistry, and biotechnology as well as materials science. The user interface provides physics-based abstractions (e.g., atoms, bonds, molecules) of the data that can be easily manipulated by the user. It hides the complexity of accessing data and frees the user from having to implement the details of different trajectory and topology file formats (which by themselves are often only poorly documented and just adhere to certain community expectations that can be difficult to understand for outsiders). MDAnalysis currently supports more than 25 different file formats and covers the vast majority of data formats that are used in the biomolecular simulation community, including the formats required and produced by the most popular packages such as NAMD [PBW+05], Amber [CCD+05], Gromacs [AMS+15], CHARMM [BBIM+09], LAMMPS [Pli95], DL_POLY [TSTD06], HOOMD [GNA+15] as well as the Protein Data Bank PDB format [BWF+00] and various other specialized formats.

Since the original publication [MADWB11], improvements in speed and data structures make it now possible to work with terabyte-sized trajectories containing up to ~10 million particles. MDAnalysis also comes with specialized analysis classes in the `MDAnalysis.analysis` module that are unique to MDAnalysis such as *LeafletFinder* (in the `leaflet` module), a graph-based algorithm for the analysis of lipid bilayers [MADWB11], or *Path Similarity Analysis* (`psa`) for the quantitative comparison of macromolecular conformational changes [SKTB15].

Code base

MDAnalysis is written in Python and Cython with about 42k lines of code and 24k lines of comments and documentation. It uses NumPy arrays [VCV11] for easy interoperability with the wider scientific Python ecosystem. Although the primary dependency is NumPy, other Python packages such as netcdf4 and BioPython [HM03] also provide specialized functionality to the core of the library (Figure 1).

Availability

MDAnalysis is available in source form under the GNU General Public License v2 from GitHub as MDAnalysis/mdanalysis, and as PyPi and conda packages. The documentation is extensive and includes an introductory tutorial.

Development process

The development community is very active with more than five active core developers and many community contributions in every release. We use modern software development practices [WAB+14], [SM14] with continuous integration (provided by Travis CI) and an extensive automated test suite (containing over 3500 tests with >92% coverage for our core modules). Development occurs on GitHub through pull requests that are reviewed by core developers and other contributors, supported by

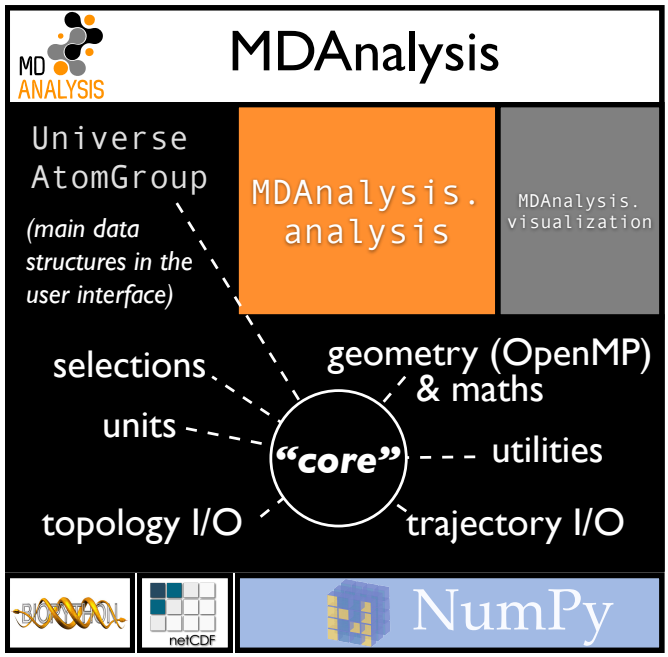

Fig. 1: *Structure of the MDAnalysis package. MDAnalysis consists of the* core *with the* Universe *class as the primary entry point for users. The* `MDAnalysis.analysis` *package contains independent modules that make use of the core to implement a wide range of algorithms to analyze MD simulations. The* `MDAnalysis.visualization` *package contains a growing number of tools that are specifically geared towards calculating visual representations such as, for instance, streamlines of molecules.*

the results from the automated tests, test coverage reports provided by Coveralls, and QuantifiedCode code quality reports. Users and developers communicate extensively on the community mailing list (*Google* groups) and the GitHub issue tracker; new users and developers are very welcome and most user contributions are eventually integrated into the code base. The development and release process is transparent to users through open discussions and announcements and a full published commit history and changes. Releases are numbered according to the semantic versioning convention so that users can immediately judge the impact of a new release on their existing code base, even without having to consult the `CHANGELOG` documentation. Old code is slowly deprecated so that users have ample opportunity to update the code although we generally attempt to break as little code as possible. When backwards-incompatible changes are inevitable, we provide tools (based on the Python standard library's *lib2to3*) to automatically refactor code or warn users of possible problems with their existing code.

Basic usage

The core object in MDAnalysis is the Universe which acts as a nexus for accessing all data contained within a simulation. It is initialized by passing the file names of the topology and trajectory files, with a multitude of different formats supported in these roles. The topology acts as a description of all the particles in the system while the trajectory describes their behavior over time.

```
import MDAnalysis as mda

# Create a Universe based on simulation results
u = mda.Universe('topol.tpr', 'traj.trr')
```

```
# Create a selection of atoms to work with
ag = u.atoms.select_atoms('backbone')
```

The select_atoms method allows for AtomGroups to be created using a human readable syntax which allows queries according to properties, logical statements and geometric criteria.

```
# Select all solvent within a set distance from protein atoms
ag = u.select_atoms('resname SOL and around 5.0 protein')

# Select all heavy atoms in the first 20 residues
ag = u.select_atoms('resid 1:20 and not prop mass < 10.0')

# Use a preexisting AtomGroup as part of another selection
sel1 = u.select_atoms('name N and not resname MET')
sel2 = u.select_atoms('around 2.5 group Nsel', Nsel=sel1)

# Perform a selection on another AtomGroup
sel1 = u.select_atoms('around 5.0 protein')
sel2 = sel1.select_atoms('type O')
```

The AtomGroup acts as a representation of a group of particles, with the properties of these particles made available as NumPy arrays.

```
ag.names
ag.charges
ag.positions
ag.velocities
ag.forces
```

The data from MD simulations comes in the form of a trajectory which is a frame by frame description of the motion of particles in the simulation. Today trajectory data can often reach sizes of hundreds of GB. Reading all these data into memory is slow and impractical. To allow the analysis of such large simulations on an average workstation (or even laptop) MDAnalysis will only load a single frame of a trajectory into memory at any time.

The trajectory data can be accessed through the trajectory attribute of a Universe. Changing the frame of the trajectory object updates the underlying arrays that AtomGroups point to. In this way the positions attribute of an AtomGroup within the iteration over a trajectory will give access to the positions at each frame. Through this approach only a single frame of data is present in memory at any time, allowing for large data sets, from half a million particles to tens of millions (see also section Analysis of large systems), to be dissected with minimal resources.

```
# the trajectory is an iterable object
len(u.trajectory)

# seek to a given frame
u.trajectory[72]
# iterate through every 10th frame
for ts in u.trajectory[::10]:
    ag.positions
```

In some cases it is necessary to access frames of trajectories in a random access pattern or at least be able to rapidly access a starting frame anywhere in the trajectory. Examples for such usage are the calculation of time correlation functions, skipping of frames (as in the iterator `u.trajectory[5000::1000]`), or parallelization over trajectory blocks in a map/reduce pattern [TRB+08]. If the underlying trajectory reader only implements linear sequential reading from the beginning, searching for specific frames becomes extremely inefficient, effectively prohibiting random access to time frames on disk. Many trajectory formats suffer from this shortcoming, including the popular Gromacs XTC and TRR formats, but also commonly used multi-frame PDB files and other text-based formats such as XYZ. LOOS [RG09] implemented a mechanism by which the trajectory was read once on loading and frame offsets on disk were computed that could be used to directly seek to individual frames. Based on this idea, MDAnalysis implements a fast frame scanning algorithm for TRR and XTC files and also saves the offsets to disk (as a compressed NumPy array). When a trajectory is loaded again then instead of reading the whole trajectory, only the persistent offsets are read (provided they have not become stale as checked by conservative criteria such as changes in file name, modification time, and size of the original file, which are all saved with the offsets). In cases of terabyte-sized trajectories, the persistent offset approach can save hundreds of seconds for the initial loading of the `Universe` (after an initial one-time cost of scanning the trajectory). Current development work is extending the persistent offset scheme to all trajectory readers, which will provide random access for all trajectories in a completely automatic and transparent manner to the user.

Example: Per-residue RMSF

As a complete example consider the calculation of the C_α root mean square fluctuation (RMSF) ρ_i that characterizes the mobility of a residue i in a protein:

$$\rho_i = \sqrt{\left\langle (\mathbf{x}_i(t) - \langle \mathbf{x}_i \rangle)^2 \right\rangle} \tag{1}$$

The code in Figure 2 A shows how MDAnalysis in combination with NumPy can be used to implement Eq. 1. The topology information and the trajectory are loaded into a `Universe` instance; C_α atoms are selected with the MDAnalysis selection syntax and stored as the `AtomGroup` instance `ca`. The main loop iterates through the trajectory using the MDAnalysis trajectory iterator. The coordinates of all selected atoms become available in a NumPy array `ca.positions` that updates for each new time step in the trajectory. Fast operations on this array are then used to calculate variance over the whole trajectory. The final result is plotted with matplotlib [Hun07] as the RMSF over the residue numbers, which are conveniently provided as an attribute of the `AtomGroup` (Figure 2 B).

The example demonstrates how the abstractions that MDAnalysis provides enable users to write concise code where the computations on data are cleanly separated from the task of extracting the data from the simulation trajectories. These characteristics make it easy to rapidly prototype new algorithms. In our experience, most new analysis algorithms are developed by first prototyping a simple script (like the one in Figure 2), often inside a Jupyter notebook (see section Interactive Use and Visualization). Then the code is cleaned up, tested and packaged into a module. In section Analysis Module, we describe the analysis code that is included as modules with MDAnalysis.

Interactive use and visualization

The high level of abstraction and the pythonic API, together with comprehensive Python doc strings, make MDAnalysis well suited for interactive and rapid prototyping work in IPython [PG07] and Jupyter notebooks. It works equally well as an interactive analysis tool, especially with Jupyter notebooks, which then contain an executable and well-documented analysis protocol that can be easily shared and even accessed remotely. Universes and AtomGroups can be visualized in Jupyter notebooks using nglview, which interacts natively with the MDAnalysis API (Figure 3).

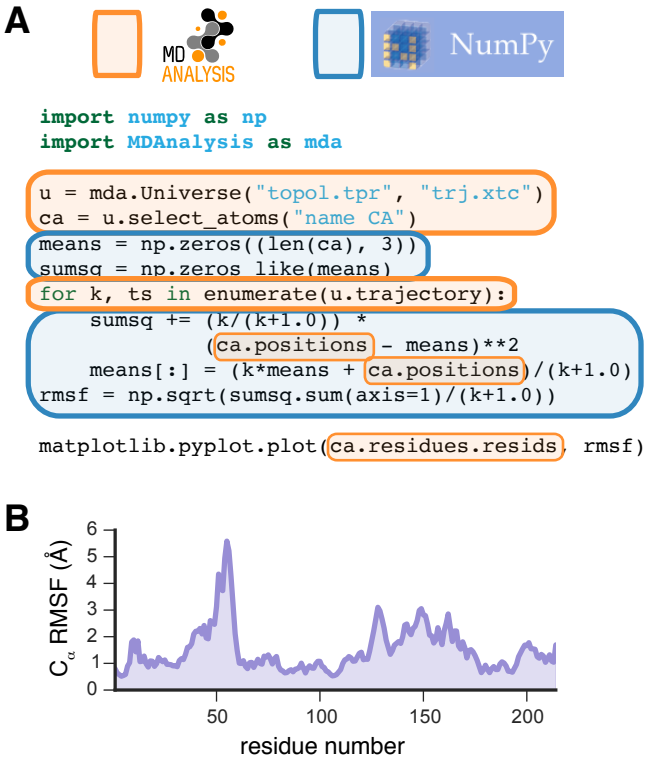

Fig. 2: *Example for how to calculate the root mean square fluctuation (RMSF) for each residue in a protein with MDAnalysis and NumPy. **A**: Based on the input simulation data (topology and trajectory in the Gromacs format (TPR and XTC), MDAnalysis makes coordinates of the selected C_α atoms available as NumPy arrays. From these coordinates, the RMSF is calculated by averaging over all frames in the trajectory. The RMSF is then plotted with matplotlib. The algorithm to calculate the variance in a single pass is due to Welford [Wel62]. **B**: C_α RMSF for each residue.*

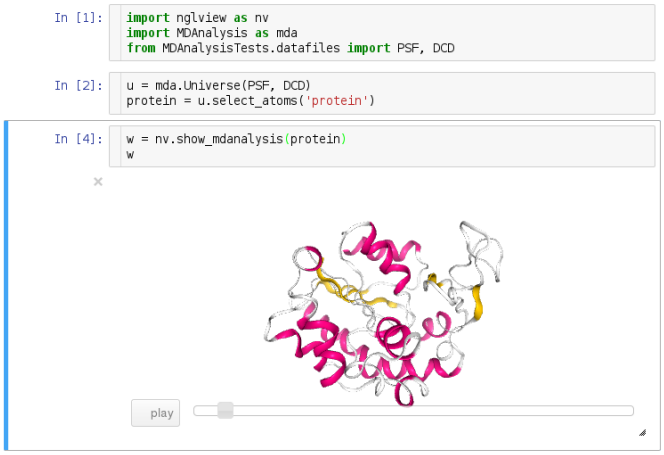

Fig. 3: *MDAnalysis can be used with nglview to directly visualize molecules and trajectories in Jupyter notebooks. The adenylate kinase (AdK) protein from one of the included test trajectories is shown. .*

Other Python packages that have become extremely useful in notebook-based analysis work flows are pandas [McK10] for rapid analysis of time series analysis, distributed [Roc15] for simple parallelization, FireWorks [JOC+15] for complex work flows, and MDSynthesis [DGS+16] for organizing, bundling and querying many simulations.

Analysis module

In the `MDAnalysis.analysis` module we provide a large variety of standard analysis algorithms, like RMSD (root mean square distance) and RMSF (root mean square fluctuation) calculations, RMSD-optimized structural superposition [LAT10], native contacts [BHE13], [FKDD07], or analysis of hydrogen bonds as well as unique algorithms, such as the *LeafletFinder* in `MDAnalysis.analysis.leaflet` [MADWB11] and *Path Similarity Analysis* (`MDAnalysis.analysis.psa`) [SKTB15]. Historically these algorithms were contributed by various researchers as individual modules to satisfy their own needs but this lead to some fragmentation in the user interface. We have recently started to unify the interface to the different algorithms with an *AnalysisBase* class. Currently `PersistenceLength`, `InterRDF`, `LinearDensity` and `Contacts` analysis have been ported. `PersistenceLength` calculates the persistence length of a polymer, `InterRDF` calculates the pairwise radial distribution function inside of a molecule, `LinearDensity` generates a density along a given axis and `Contacts` analysis native contacts, as described in more detail below. The API to these different algorithms is being unified with a common `AnalysisBase` class, with an emphasis on keeping it as generic and universal as possible so that it becomes easy to, for instance, parallelize analysis. Most other tools hand the user analysis algorithms as black boxes. We want to avoid that and allow the user to adapt an analysis to their needs.

The new `Contacts` class is a good example of a generic API that allows straightforward implementation of algorithms while still offering an easy setup for standard analysis types. The `Contacts` class is calculating a contact map for atoms in a frame and compares it with a reference map using different metrics. The used metric then decides which quantity is measured. A common quantity is the fraction of native contacts, where native contacts are all atom pairs that are close to each other in a reference structure. The fraction of native contacts is often used in protein folding to determine when a protein is folded. For native contacts two major types of metrics are considered: ones based on differentiable functions [BHE13] and ones based on hard cut-offs [FKDD07] (which we set as the default implementation). We have designed the API to choose between the two metrics and pass user defined functions to develop new metrics or measure other quantities. This generic interface allowed us to implement a "q_1q_2" analysis [FKDD07] on top of the `Contacts` class; q_1 and q_2 refer to the fractions of native contacts that are present in a protein structure relative to *two* reference states 1 and 2. Below is an incomplete code example that shows how to implement a q_1q_2 analysis, the default value for the *method* keyword argument is overwritten with a user defined method *radius_cut_q*. A more detailed explanation can be found in the documentation.

```
def radius_cut_q(r, r0, radius):
    y = r <= radius
    return y.sum() / r.size

contacts = Contacts(u, selection,
```

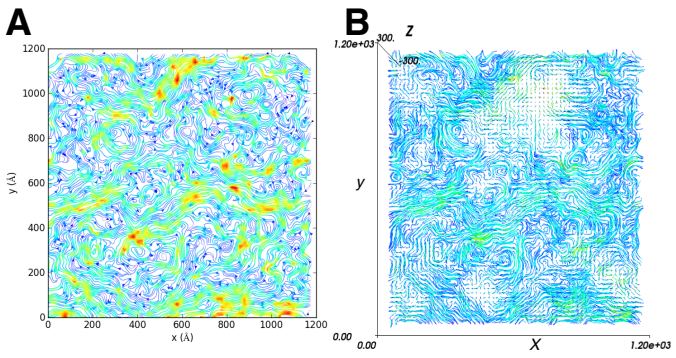

Fig. 4: *Visualization of the flow of lipids in a large bilayer membrane patch. **A**: 2D stream plot (produced with* `MDAnalysis.visualization.streamlines` *and plotted with matplotlib [Hun07]). **B**: 3D stream plot, viewed down the z axis onto the membrane (produced with* `MDAnalysis.visualization.streamlines_3D` *and plotted with MayaVi [RV11]).*

```
(first_frame, last_frame),
radius=radius,
method=radius_cut_q,
start=start, stop=stop,
step=step,
kwargs={'radius': radius})
```

This type of flexible analysis algorithm paired with a collection of base classes enables rapid and easy analysis of simulations as well as development of new ones.

Visualization module

The new `MDAnalysis.visualization` name space contains modules that primarily produce visualizations of molecular systems. Currently it contains functions that generate specialized streamline visualizations of lipid diffusion in membrane bilayers [CRG+14]. In short, the algorithm decomposes any given membrane into a grid and tracks the displacement of lipids between different grid elements, emphasizing collective lipid motions. Both 2D (`MDAnalysis.visualization.streamlines`) and 3D (`MDAnalysis.visualization.streamlines_3D`) implementations are available in MDAnalysis, with output shown in Figure 4. Sample input data files are available online from the Flows website along with the expected output visualizations.

Improvements in the internal topology data structures

Originally MDAnalysis followed a strict object-oriented approach with a separate instance of an Atom object for each particle in the simulation data. The AtomGroup then simply stored its contents as a list of these Atom instances. With simulation data now commonly exceeding 10^6 particles this solution did not scale well and so recently this design was overhauled to improve the scalability of MDAnalysis.

Because all Atoms have the same property fields (i.e. mass, position) it is possible to store this information as a single NumPy array for each property. Now an AtomGroup can keep track of its contents as a simple integer array, which can be used to slice these property arrays to yield the relevant data.

Overall this approach means that the same number of Python objects are created for each Universe, with the number of particles only changing the size of the arrays. This translates into a much smaller memory footprint (1.3 GB vs. 3.6 GB for a 10.1 M atom system), highlighting the memory cost of millions of simple Python objects.

# atoms	v0.15.0	v0.16.0	speed up
1.75 M	19 ms	0.45 ms	42
3.50 M	18 ms	0.54 ms	33
10.1 M	17 ms	0.45 ms	38

TABLE 1: *Performance comparison of subselecting an AtomGroup from an existing one using the new system (upcoming release v0.16.0) against the old (v0.15.0). Subselections were slices of the same size (82,056 atoms). Shorter processing times are better. The benchmarks systems were taken from the vesicle library [KB15] and are listed with their approximate number of particles ("# atoms"). Benchmarks were performed on a laptop with an Intel Core i5 2540M 2.6 GHz processor, 8 GB of RAM and a SSD drive.*

# atoms	v0.15.0	v0.16.0	speed up
1.75 M	250 ms	35 ms	7.1
3.50 M	490 ms	72 ms	6.8
10.1 M	1500 ms	300 ms	5.0

TABLE 2: *Performance comparison of accessing attributes with new AtomGroup data structures (upcoming release v0.16.0) compared with the old Atom classes (v0.15.0). Shorter access times are better. The same benchmark systems as in Table 1 were used.*

This transformation of the data structures from an Array of Structs to a Struct of Arrays also better suits the typical access patterns within MDAnalysis. It is quite common to compare a single property across many Atoms, but rarely are different properties within a single Atom compared. Additionally, it is possible to utilize NumPy's faster indexing capabilities rather than using a list comprehension. This new data structure has lead to performance improvements in our whole code base. The largest improvement is in accessing subsets of Atoms which is now over 40 times faster (Table 1), an operation that is used everywhere in MDAnalysis. Speed-ups of a factor of around five to seven were realized for accessing Atom attributes for whole AtomGroup instances (Table 2). The improved topology data structures are also much faster to initialize, which translates into speed-ups of about three for the task of loading a system from a file (for instance, in the Gromacs GRO format or the Protein Databank PDB format) into a *Universe* instance (Table 3). Given that for systems with 10 M atoms this process used to take over 100 s, the reduction in load time down to a third is a substantial improvement — and it came essentially "for free" as a by-product of improving the underlying topology data structures.

Analysis of large systems

MDAnalysis has been used extensively to study extremely large simulation systems for long simulation times. Marrink and co-workers [IME+14] used MDAnalysis to analyze a realistic model of the membrane of a mammalian cell with 63 different lipid species and over half a million particles for 40 µs. They discovered that transient domains with liquid-ordered character formed and disappeared on the microsecond time scale, with different lipid

# atoms	v0.15.0	v0.16.0	speed up
1.75 M	18 s	5 s	3.6
3.50 M	36 s	11 s	3.3
10.1 M	105 s	31 s	3.4

TABLE 3: *Performance comparison of loading a topology file with 1.75 to 10 million atoms with new AtomGroup data structures (upcoming release v0.16.0) compared with the old Atom classes (v0.15.0). Shorter loading times are better. The same benchmark systems as in Table 1 were used.*

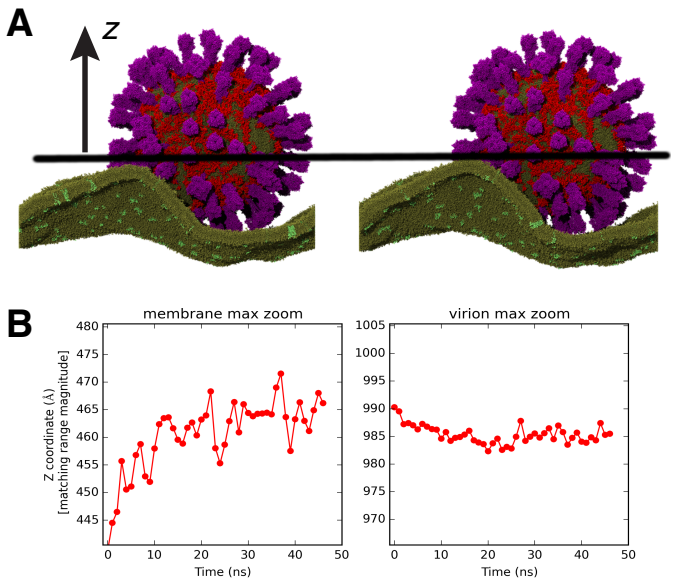

Fig. 5: *Simulation of a coarse-grained model of the influenza A virion membrane (purple/red) close to a model of the human plasma membrane (brown).* ***A****: Left: initial frame. Right: system after 40 ns . A horizontal black guide line is used to emphasize the rising plasma membrane position. The images were produced with VMD [HDS96].* ***B*** *Maximum Z (vertical) coordinate values for the influenza A virus envelope and the plasma membrane are tracked over the course of the simulation, indicating that the membrane rises to rapidly.*

species clustering in a lipid-specific manner. A coarse-grained model of the influenza A virion outer lipid envelope (5 M particles) was simulated for 5 microseconds and the resulting trajectory was analyzed using MDAnalysis [RSP+15] and the open source MDAnalysis-based lipid diffusion analysis code, which calculates the diffusion constants of lipids for spherical structures and planar bilayers [Red14]. The construction of the CG dengue virion envelope (1 M particles) was largely dependent on MDAnalysis [RS16]. The symmetry operators in the deposited dengue protein shell PDB file were applied to a simulated asymmetric unit in a bilayer, effectively tiling both proteins and lipids into the appropriate positions on the virion surface.

More recently, a 12.7 M CG particle system combining the influenza A envelope and a model of a plasma membrane [KS15] were simulated together (Figure 5 A). MDAnalysis was used to assess the stability of this enormous system by tracking, for example, the changes in Z coordinate values for different system components (Figure 5 B). In this case, the membrane appeared to rise too rapidly over the course of 50 ns, which suggests that the simulation system will likely have to be redesigned. Such large systems are challenging to work with, including their visualization, and analysis of quantities based on particle coordinates is essential to assess the correct behavior of the simulations.

Other packages that use MDAnalysis

The user interface and modular design work well in complex scripted work flows and for interactive work, as discussed in section Interactive Use and Visualization. MDAnalysis also serves as foundation for other packages. For example, ProtoMD [SMO16] is a toolkit that facilitates the development of algorithms for multiscale (MD) simulations and uses MDAnalysis for on-the-fly calculations of the collective variables that drive the coarse-grained degrees of freedom. The ENCORE package [TPB+15] enables users to compare conformational ensembles generated either from simulations alone or synergistically with experiments. MDAnalysis is also the back end for ST-analyzer [JJW+14], a standalone graphical user interface tool set to perform various trajectory analyses. MDSynthesis [DGS+16] (which is based on datreant (Dotson et al, this issue)) gives a Pythonic interface to molecular dynamics trajectories using MDAnalysis, giving the ability to work with the data from many simulations scattered throughout the file system with ease. It makes it possible to write analysis code that can work across many varieties of simulation, but even more importantly, MDSynthesis allows interactive work with the results from hundreds of simulations at once without much effort.

Conclusions

MDAnalysis provides a uniform interface to simulation data, which comes in a bewildering array of formats. It enables users to rapidly write code that is portable and immediately usable in virtually all biomolecular simulation communities. It has an active international developer community with researchers that are expert developers and users of a wide range of simulation codes. MDAnalysis is widely used (the original paper [MADWB11] has been cited more than 195 times) and forms the foundation for more specialized biomolecular simulation tools. Ongoing and future developments will improve performance further, introduce transparent parallelization schemes to utilize multi-core and GPU systems efficiently, and interface with the SPIDAL library for high performance data analytics algorithms [QJLF14].

Acknowledgments

We thank the members of the MDAnalysis community for their contributions in the form of code contributions (see the file AUTHORS in the source distribution for the names of all 44 contributors), bug reports, and enhancement requests. RG was supported by BBSRC grant BB/J014478/1. ML was supported by the Max Planck Society. JB was supported by the TOP programme of Prof. Marrink, financed by the Netherlands Organisation for Scientific Research (NWO). TR was supported by the Canadian Institutes of Health Research, the Wellcome Trust, the Leverhulme Trust, and Somerville College; computational resources for TR's work were provided by PRACE, HPC-Europa2, CINES (France), and the SBCB unit (Oxford). MNM was supported by the NWO VENI grant 722.013.010. SLS was supported in part by a Wally Stoelzel Fellowship from the Department of Physics at Arizona State University. JD was in part supported by a Wellcome Trust grant 092970/Z/10/Z. DLD was in part supported by a Molecular

Imaging Fellowship from the Department of Physics at Arizona State University. IMK was supported by a REU supplement to grant ACI-1443054 from the National Science Foundation. OB was supported in part by grant ACI-1443054 from the National Science Foundation; computational resources for OB's work were in part provided by the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation grant number ACI-1053575 (allocation MCB130177 to OB). The MDAnalysis *Atom* logo was designed by Christian Beckstein.

REFERENCES

[AMS+15] Mark James Abraham, Teemu Murtola, Roland Schulz, Szilárd Páll, Jeremy C. Smith, Berk Hess, and Erik Lindahl. GROMACS: High performance molecular simulations through multi-level parallelism from laptops to supercomputers. *SoftwareX*, 1–2:19 – 25, 2015. URL: http://www.gromacs.org, doi:10.1016/j.softx.2015.06.001.

[BBIM+09] B R Brooks, C L Brooks III., A D Jr Mackerell, L Nilsson, R J Petrella, B Roux, Y Won, G Archontis, C Bartels, S Boresch, A Caflisch, L Caves, Q Cui, A R Dinner, M Feig, S Fischer, J Gao, M Hodoscek, W Im, K Kuczera, T Lazaridis, J Ma, V Ovchinnikov, E Paci, R W Pastor, C B Post, J Z Pu, M Schaefer, B Tidor, R M Venable, H L Woodcock, X Wu, W Yang, D M York, and M Karplus. CHARMM: the biomolecular simulation program. *J Comput Chem*, 30(10):1545–1614, Jul 2009. URL: https://www.charmm.org, doi:10.1002/jcc.21287.

[BHE13] Robert B Best, Gerhard Hummer, and William A Eaton. Native contacts determine protein folding mechanisms in atomistic simulations. *Proc Natl Acad Sci USA*, 110(44):17874–17879, 2013. doi:10.1073/pnas.1311599110.

[BWF+00] Helen M. Berman, John Westbrook, Zukang Feng, Gary Gilliland, T. N. Bhat, Helge Weissig, Ilya N. Shindyalov, and Philip E. Bourne. The Protein Data Bank. *Nucleic Acids Res*, 28(1):235–242, 2000. URL: http://www.rcsb.org/pdb/.

[CCD+05] David A Case, Thomas E Cheatham, 3rd, Tom Darden, Holger Gohlke, Ray Luo, Kenneth M Merz, Jr, Alexey Onufriev, Carlos Simmerling, Bing Wang, and Robert J Woods. The amber biomolecular simulation programs. *J Comput Chem*, 26(16):1668–1688, 2005. URL: http://ambermd.org/, doi:10.1002/jcc.20290.

[CRG+14] Matthieu Chavent, Tyler Reddy, Joseph Goose, Anna Caroline E. Dahl, John E. Stone, Bruno Jobard, and Mark S. P. Sansom. Methodologies for the analysis of instantaneous lipid diffusion in MD simulations of large membrane systems. *Faraday Discuss.*, 169:455–475, 2014. doi:10.1039/C3FD00145H.

[DDG+12] Ron O Dror, Robert M Dirks, J P Grossman, Huafeng Xu, and David E Shaw. Biomolecular simulation: a computational microscope for molecular biology. *Annu Rev Biophys*, 41:429–52, 2012. doi:10.1146/annurev-biophys-042910-155245.

[DGS+16] David Dotson, Richard Gowers, Sean Seyler, Max Linke, and Oliver Beckstein. MDSynthesis: release-0.6.1. (source code), May 2016. URL: https://github.com/datreant/MDSynthesis, doi:10.5281/zenodo.51506.

[FKDD07] Joel Franklin, Patrice Koehl, Sebastian Doniach, and Marc Delarue. MinActionPath: Maximum likelihood trajectory for large-scale structural transitions in a coarse-grained locally harmonic energy landscape. *Nucleic Acids Res*, 35(SUPPL.2):477–482, 2007. doi:10.1093/nar/gkm342.

[GNA+15] Jens Glaser, Trung Dac Nguyen, Joshua A. Anderson, Pak Lui, Filippo Spiga, Jaime A. Millan, David C. Morse, and Sharon C. Glotzer. Strong scaling of general-purpose molecular dynamics simulations on gpus. *Computer Physics Communications*, 192:97–107, 7 2015. URL: http://glotzerlab.engin.umich.edu/hoomd-blue/, doi:10.1016/j.cpc.2015.02.028.

[HDS96] W. Humphrey, A. Dalke, and K. Schulten. VMD – Visual Molecular Dynamics. *J Molec Graphics*, 14:33–38, 1996. URL: http://www.ks.uiuc.edu/Research/vmd/.

[Hin00] K. Hinsen. The molecular modeling toolkit: a new approach to molecular simulations. *J Comput Chem*, 21(2):79–85, 2000.

[HM03] Thomas Hamelryck and Bernard Manderick. PDB file parser and structure class implemented in python. *Bioinformatics*, 19(17):2308–2310, 2003. doi:10.1093/bioinformatics/btg299.

[Hun07] John D. Hunter. Matplotlib: A 2D graphics environment. *Computing in Science & Engineering*, 9(3):90–95, May-Jun 2007. URL: http://matplotlib.org.

[IME+14] Helgi I Ingólfsson, Manuel N Melo, Floris J Van Eerden, Clement Arnarez, Cesar A López, Tsjerk A Wassenaar, Xavier Periole, Alex H De Vries, D Peter Tieleman, and Siewert J Marrink. Lipid organization of the plasma membrane. *J Am Chem Soc*, 136(41):14554–14559, 2014. doi:10.1021/ja507832e.

[JJW+14] Jong Cheol Jeong, Sunhwan Jo, Emilia L Wu, Yifei Qi, Viviana Monje-Galvan, Min Sun Yeom, Lev Gorenstein, Feng Chen, Jeffery B Klauda, and Wonpil Im. ST-analyzer: a web-based user interface for simulation trajectory analysis. *J Comput Chem*, 35(12):957–63, May 2014. doi:10.1002/jcc.23584.

[JOC+15] Anubhav Jain, Shyue Ping Ong, Wei Chen, Bharat Medasani, Xiaohui Qu, Michael Kocher, Miriam Brafman, Guido Petretto, Gian-Marco Rignanese, Geoffroy Hautier, Daniel Gunter, and Kristin A. Persson. Fireworks: a dynamic workflow system designed for high-throughput applications. *Concurrency and Computation: Practice and Experience*, 27(17):5037–5059, 2015. URL: https://github.com/materialsproject/fireworks, doi:10.1002/cpe.3505.

[KB15] Ian M. Kenney and Oliver Beckstein. SPIDAL Summer REU 2015: Biomolecular benchmark systems. Technical report, Arizona State University, Tempe, AZ, October 2015. doi:10.6084/m9.figshare.1588804.v1.

[KS15] Heidi Koldsø and Mark S. P. Sansom. Organization and dynamics of receptor proteins in a plasma membrane. *J Am Chem Soc*, 137(46):14694–14704, 2015. PMID: 26517394. doi:10.1021/jacs.5b08048.

[LAT10] Pu Liu, Dimitris K Agrafiotis, and Douglas L. Theobald. Fast Determination of the Optimal Rotational Matrix for Macromolecular Superpositions. *J Comput Chem*, 31(7):1561–1563, 2010. doi:10.1002/jcc.21439.

[MADWB11] Naveen Michaud-Agrawal, Elizabeth Jane Denning, Thomas B. Woolf, and Oliver Beckstein. MDAnalysis: A toolkit for the analysis of molecular dynamics simulations. *J Comput Chem*, 32:2319–2327, 2011. URL: http://mdanalysis.org, doi:10.1002/jcc.21787.

[MBH+15] Robert T. McGibbon, Kyle A. Beauchamp, Matthew P. Harrigan, Christoph Klein, Jason M. Swails, Carlos X. Hernández, Christian R. Schwantes, Lee-Ping Wang, Thomas J. Lane, and Vijay S. Pande. MDTraj: A modern open library for the analysis of molecular dynamics trajectories. *Biophysical J*, 109(8):1528 – 1532, 2015. URL: http://mdtraj.org, doi:10.1016/j.bpj.2015.08.015.

[McK10] Wes McKinney. Data Structures for Statistical Computing in Python. *Proceedings of the 9th Python Science Conference*, 1697900(Scipy):51–56, 2010. URL: http://conference.scipy.org/proceedings/scipy2010/mckinney.html.

[NRSC16] Hai Nguyen, Daniel R. Roe, Jason Swails, and David A. Case. PYTRAJ: Interactive data analysis for molecular dynamics simulations. (source code), 2016. URL: https://github.com/Amber-MD/pytraj.

[Oro14] Modesto Orozco. A theoretical view of protein dynamics. *Chem. Soc. Rev.*, 43:5051–5066, 2014. doi:10.1039/C3CS60474H.

[PBW+05] JC Phillips, R Braun, W Wang, J Gumbart, E Tajkhorshid, E Villa, C Chipot, RD Skeel, L Kale, and K Schulten. Scalable molecular dynamics with NAMD. *J Comput Chem*, 26:1781–1802, 2005. URL: http://www.ks.uiuc.edu/Research/namd/, doi:10.1002/jcc.20289.

[PG07] Fernando Pérez and Brian E. Granger. IPython: A system for interactive scientific computing. *Comput Sci Eng*, 9(3):21–29, 2007. URL: https://ipython.org/, doi:10.1109/MCSE.2007.53.

[Pli95] Steve Plimpton. Fast parallel algorithms for short-range molecular dynamics. *J Comput Phys*, 117(1):1–19, 1995. URL: http://lammps.sandia.gov/index.html, doi:10.1006/jcph.1995.1039.

[QJLF14] Judy Qiu, Shantenu Jha, Andre Luckow, and Geoffrey C. Fox. Towards HPC-ABDS: An initial high-performance big data

stack. In *Building Robust Big Data Ecosystem*, San Diego Supercomputer Center, San Diego, CA, 2014. ISO/IEC JTC 1 Study Group on Big Data. URL: http://spidal.org.

[RCI13] Daniel R. Roe and Thomas E. Cheatham III. PTRAJ and CPPTRAJ: Software for processing and analysis of molecular dynamics trajectory data. *J Chemical Theory Computation*, 9(7):3084–3095, 2013. URL: https://github.com/Amber-MD/cpptraj, doi:10.1021/ct400341p.

[Red14] Tyler Reddy. diffusion_analysis_MD_simulations: Initial release. (source code), September 2014. URL: https://github.com/tylerjereddy/diffusion_analysis_MD_simulations, doi:10.5281/zenodo.11827.

[RG09] Tod D. Romo and Alan Grossfield. LOOS: An extensible platform for the structural analysis of simulations. In *31st Annual International Conference of the IEEE EMBS*, pages 2332–2335, Minneapolis, Minnesota, USA, 2009. IEEE. URL: http://loos.sourceforge.net/.

[RLG14] Tod D. Romo, Nicholas Leioatts, and Alan Grossfield. Lightweight object oriented structure analysis: Tools for building tools to analyze molecular dynamics simulations. *J Comput Chem*, 35(32):2305–2318, 2014. URL: http://loos.sourceforge.net/, doi:10.1002/jcc.23753.

[Roc15] Matthew Rocklin. Dask: Parallel computation with blocked algorithms and task scheduling. In *Proceedings of the 14th Python in Science Conference*, number 130–136, 2015. URL: https://github.com/dask/dask.

[RS16] T. Reddy and M. S. Sansom. The role of the membrane in the structure and biophysical robustness of the Dengue virion envelope. *Structure*, 24(3):375–382, Mar 2016. doi:10.1016/j.str.2015.12.011.

[RSP+15] T. Reddy, D. Shorthouse, D. L. Parton, E. Jefferys, P. W. Fowler, M. Chavent, M. Baaden, and M. S. Sansom. Nothing to sneeze at: a dynamic and integrative computational model of an influenza A virion. *Structure*, 23(3):584–597, Mar 2015. doi:10.1016/j.str.2014.12.019.

[RV11] P. Ramachandran and G. Varoquaux. Mayavi: 3D visualization of scientific data. *Computing in Science & Engineering*, 13(2):40–51, 2011. URL: http://code.enthought.com/projects/mayavi/.

[SKTB15] Sean L. Seyler, Avishek Kumar, M. F. Thorpe, and Oliver Beckstein. Path similarity analysis: A method for quantifying macromolecular pathways. *PLoS Comput Biol*, 11(10):e1004568, 10 2015. doi:10.1371/journal.pcbi.1004568.

[SM14] Victoria Stodden and Sheila Miguez. Best practices for computational science: Software infrastructure and environments for reproducible and extensible research. *J Open Research Software*, 2(1):e21, July 2014. doi:10.5334/jors.ay.

[SMO16] Endre Somogyi, Andrew Abi Mansour, and Peter J. Ortoleva. ProtoMD: A prototyping toolkit for multiscale molecular dynamics. *Computer Physics Communications*, 202:337 – 350, 2016. URL: https://github.com/CTCNano/proto_md, doi:10.1016/j.cpc.2016.01.014.

[TPB+15] Matteo Tiberti, Elena Papaleo, Tone Bengtsen, Wouter Boomsma, and Kresten Lindorff-Larsen. ENCORE: Software for quantitative ensemble comparison. *PLoS Comput Biol*, 11(10):e1004415, 10 2015. doi:10.1371/journal.pcbi.1004415.

[TRB+08] T. Tu, C.A. Rendleman, D.W. Borhani, R.O. Dror, J. Gullingsrud, MO Jensen, J. L. Klepeis, P. Maragakis, P. Miller, K.A. Stafford, and David E. Shaw. A scalable parallel framework for analyzing terascale molecular dynamics simulation trajectories. In *International Conference for High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008.*, pages 1–12, Austin, TX, 2008. IEEE. doi:10.1109/SC.2008.5214715.

[TSTD06] Ilian T Todorov, William Smith, Kostya Trachenko, and Martin T Dove. DL_POLY_3: new dimensions in molecular dynamics simulations via massive parallelism. *Journal of Materials Chemistry*, 16(20):1911–1918, 2006. URL: http://www.ccp5.ac.uk/DL_POLY_CLASSIC/.

[VCV11] Stefan Van Der Walt, S. Chris Colbert, and Gael Varoquaux. The NumPy array: A structure for efficient numerical computation. *Comput Sci Eng*, 13(2):22–30, 2011. URL: http://www.numpy.org/, arXiv:1102.1523, doi:10.1109/MCSE.2011.37.

[WAB+14] Greg Wilson, D A Aruliah, C Titus Brown, Neil P Chue Hong, Matt Davis, Richard T Guy, Steven H D Haddock, Kathryn D Huff, Ian M Mitchell, Mark D Plumbley, Ben Waugh, Ethan P White, and Paul Wilson. Best practices for scientific computing. *PLoS Biol*, 12(1):e1001745, Jan 2014. doi:10.1371/journal.pbio.1001745.

[Wel62] B. P. Welford. Note on a method for calculating corrected sums of squares and products. *Technometrics*, 4(3):419–420, 1962. doi:10.1080/00401706.1962.10490022.

Validating Function Arguments in Python Signal Processing Applications

Patrick Steffen Pedersen‡*, Christian Schou Oxvig‡, Jan Østergaard‡, Torben Larsen‡

Abstract—Python does not have a built-in mechanism to validate the value of function arguments. This can lead to nonsensical exceptions, unexpected behaviour, erroneous results and the like. In the present paper, we define the concept of so-called application-driven data types which place a layer of abstraction on top of Python data types. With this concept in mind, we discuss the current argument validation solutions of PyDBC, Traitlets and Numtraits, MyPy, PyValid, and PyContracts. We find that they share the issue of expressing the validation scheme in terms of Python objects rather than in terms of the data they hold. Consequently, we lay out a suggestion for a validation strategy including what qualifies as a validation scheme, how to create an interface which promotes both usability and readability, and which Python constructs to encourage using for validation encapsulation. A reference implementation of the suggested validation strategy is part of the open-source Python package, Magni which is thus presented along with a number of examples of the usages of this package.

Index Terms—Function Argument Validation, Application-driven Data Types, Signal Processing, Computational Science

Introduction

Python is a dynamically typed language that does not have a built-in mechanism to ensure that the value of an argument passed to a function conforms to the intentions of that particular argument. This can lead to nonsensical exceptions, unexpected behaviour, erroneous results and the like. In signal processing applications and scientific computing in general, large amounts of numerical data are passed to any number of functions that inherently impose limitations upon that data. If such functions do not validate their arguments, these limitations may be violated without raising exceptions leading to potentially erroneous results. Thus, although impairing the performance, explicit validation may not only spare the user a lot of frustration by providing useful exceptions but may also prevent erroneous results and thereby ensure the credibility of works in scientific computing.

The usage of explicit function argument validation could be considered "unpythonic"[1] as it goes against dynamic typing [CVS13] and duck typing [CVS13] by not relying on documentation, clear code and testing to ensure correct usage. Even so, there exist a number of solutions for validating function arguments in Python relying on a wide range of language constructs and interfaces. The validation capabilities of these solutions vary greatly from type, attribute, and value checks to fully customisable checks. Among these solutions are PyDBC, Traitlets and Numtraits, MyPy, PyValid, and PyContracts which are all discussed later. Most of the solutions do, however, seem to have an interface which relates to the data model used by Python and therefore translates to Python check in a straightforward way.

Unfortunately, there are a number of shortcomings with the existing validation strategies as implemented in the existing Python packages with validation capabilities; in particular in signal processing applications. Some of the existing solutions lack generality, some do not promote readability, and some are inconvenient to use. However, the primary issue is that the validation scheme of function arguments is expressed in terms of Python objects rather than in terms of the data they hold, and this poses a number of problems. Even with these shortcomings, the existing solutions represent a large variety of validation strategies which are an obvious source of inspiration.

In the present effort, we suggest the concept of so-called application-driven data types as a signal processing data model for programming. These data types are intended for expressing the validation scheme of function arguments. Furthermore, based on existing solutions, we lay out a new strategy for validating function arguments in Python signal processing applications. Finally, we present the open-source Python package, Magni which includes a reference implementation of the suggested validation strategy, and we show a number of examples of the usage of this package.

The remainder of the present paper is organised as follows. We first take a look at validation in Python at a glance before presenting the concept of application-driven data types. Next, we discuss some of the existing solutions with an emphasis on the validation strategies they represent. Drawing on the observations made, we then present the suggested Python validation strategy. Following this specification, we detail a reference implementation of it and give examples of its usage. Finally, we conclude on what is achieved by the presented validation strategy and reference implementation as well as when to use them. All code examples have been run with Python 2.7 unless otherwise noted, all tracebacks have been removed to save space, and exception messages and the like have been broken across multiple lines using trailing backslashes where necessary.

* *Corresponding author: psp@es.aau.dk*
‡ *Faculty of Engineering and Science, Department of Electronic Systems, Section of Signal and Information Processing, Aalborg University, 9220 Aalborg, Denmark*

1. For an informal yet fitting definition, see http://stackoverflow.com/questions/25011078/what-does-pythonic-mean

Validation in Python at a glance

For the purpose of exemplifying the concepts discussed in this section, we define a simple Python function for returning the square root of the first item of a sequence. Obviously, only a sequence with a non-negative, numerical first item, $a_0 \in \mathbb{R}_{\geq 0}$, should be a valid argument of this function.

```
def do_something(a):
    print(a[0]**0.5)
```

To quote the Zen of Python[2], "there should be one-- and preferably only one --obvious way to do it" when faced with solving a task in Python, and the obvious ways to solve common tasks are oftentimes referred to as pythonic idioms. When it comes to function argument validation in Python, the most pythonic idiom is to clearly document what a function expects and then just try to use whatever gets passed to the function and either let exceptions propagate or catch attribute errors and raise other exceptions instead. This approach is well-suited for Python because it is a dynamically typed language. Basically, this means that variables, such as the function argument in the example, are not limited to hold values of a certain type. Instead, we can pass a number, a sequence, a mapping, or any other type to the example function. Regardless of the type, Python tries to use whatever value gets passed to the function which is a consequence of duck typing. The basic principle is that if a bird looks like a duck, swims like a duck, and quacks like a duck, then it probably is a duck. That is, if a value exhibits the desired behaviour, then that value probably is valid. Translated to our example, if the value of the function argument, `a`, has the `__getitem__` attribute which Python uses internally for retrieving the first item, then `a` probably is valid. Thus, the most pythonic idiom would rely on documentation, clear code, and testing to ensure correct usage rather than explicitly testing function arguments to ensure conformity to the intentions of the function.

What happens, then, if the value of a function argument is invalid by the reckoning of duck typing? This is the case with the following call as the built-in `int` type does not define `__getitem__`:

```
>>> integer = 42
>>> do_something(integer)
TypeError: 'int' object has no attribute \
'__getitem__'
```

With the following call, a `TypeError` exception is raised with a message that "`'int'` object has no attribute `'__getitem__'`". Even with this simple example, such an exception message is less sensible than desired. Furthermore, such an exception is as likely to occur in some obscure function call and, thus, be accompanied by a traceback with more levels than anyone would want. However, at least the presence of an exception indicates that something did not go as expected. What happens, however, if the value of a function argument is valid by the reckoning of duck typing but does not conform to the intentions of the function? This is the case with the following call as the built-in `dict` type defines `__getitem__` but with a different purpose than the `__getitem__` of sequences:

```
>>> dictionary = {-1: 0, 0: 1}
>>> do_something(dictionary)
1.0
```

The intention of the function is to operate on the first item of the function argument, but `dictionary` is unordered meaning that there is no such thing as a first item. However, the call does not raise an exception because of duck typing. This is an example of unexpected or erroneous behaviour.

The two examples of calls presented showcase how the lack of function argument validation can lead to hard-to-debug exceptions or even worse to unexpected or erroneous behaviour. The benefit of explicit function argument validation is that the mentioned problems should be avoided. Furthermore, by having such validation for functions that are part of a public API of released packages, the package is made more trustworthy and user-friendly.

How to Test for Validity

One way to test for validity would be to check if the value of a variable has a certain type. That is, to determine the validity based on what a value *is*. For example, we could rewrite the `do_something` example in the following way:

```
def do_something(a):
    if not isinstance(a, list):
        raise TypeError('Descriptive message.')

    if not isinstance(a[0], int):
        raise TypeError('Descriptive message.')

    print(a[0]**0.5)
```

Obviously, this approach to validation goes against dynamical typing as it restricts variables to only hold values of certain types. In the example, `a` may hold values of the type `list` or of a derived type, and the first item of `a` may hold values of the type `int` or of a derived type. Clearly, the validation in the above example is too restrictive: as the intention of the function is to allow a sequence with a non-negative, numerical first item, the following call should pass but instead fails the validation checks:

```
>>> sequence = (0., 1.)
>>> do_something(sequence)
TypeError: Descriptive message.
```

The issue is that a number of Python types represent sequences, and a number of Python types represent numbers. This could be accounted for in the example, but the point to stress is that the programmer should not have to know about every single Python type, nor should he or she have to explicitly list a large number of Python types for each validation check.

Another way to test for validity would be to check if the value of a variable displays a certain behaviour. That is, to determine the validity based on what a value *can do*. For example, we could rewrite the `do_something` example in the following way:

```
def do_something(a):
    if not hasattr(a, '__getitem__'):
        raise TypeError('Descriptive message.')

    if not hasattr(a[0], '__pow__'):
        raise TypeError('Descriptive message.')

    print(a[0]**0.5)
```

Clearly, this approach to validation is along the lines of duck typing as it explicitly checks for the presence of the required attribute. In the example, `a` may hold values of any type that defines the `__getitem__` attribute, and `a[0]` may hold values of any type that defines the `__pow__` attribute. Unlike with the first way to test for validity, the validation in the above example is not restrictive enough as already explained using the example with the dictionary. The same check could be achieved in a cleaner and

2. See https://www.python.org/dev/peps/pep-0020/

more thorough way using abstract base classes[3], but this solution would essentially suffer from the same type of problem.

Neither of the two ways to test for validity mentioned, consider the fact that the square root operation is only defined for non-negative `a[0]` values if complex numbers are ignored. Thus, a third way to partially test for validity would be to check if the value of a variable is in a set of valid values. That is, to determine validity based on what a value *contains*. For example, we could rewrite the `do_something` example in the following way:

```
def do_something(a):
    if len(a) < 1:
        raise ValueError('Descriptive message.')

    if a[0] < 0:
        raise ValueError('Descriptive message.')

    print(a[0]**0.5)
```

Obviously, this approach would have to be combined with something else to ensure that `a` is indeed a sequence and `a[0]` is indeed a number as covered by the first two ways to test for validity.

The Concept of Application-Driven Data Types

The approaches presented in the previous section do not even consider less common although valid cases such as non-derived types that only implicitly define the required attributes. Even more so, it is apparent that there is no straightforward way to test for validity based solely on what a value *is*, *can do*, or *contains*. A possible explanation for this is that all three approaches express the validation scheme in terms of Python objects rather than in terms of the data they hold. Indeed, it was easy to identify and in plain writing express that the function argument of the `do_something` example must be a sequence with a non-negative, numerical first item. Expressing the validation scheme in this way does provide a layer of abstraction.

Instead of checking if the value of `a` is a certain Python type, it would be convenient to be able to check if the value of `a` is a sequence. Likewise, instead of checking if the value of `a[0]` is a certain Python type containing a non-negative value, it would be convenient to be able to check if the value of `a[0]` is a non-negative, numerical type. Both "sequence" and "non-negative, numerical type" are examples of data types at a higher abstraction level than actual Python types, and we will name these abstractions application-driven data types.

In the context of scientific computing and signal processing in particular, the most relevant and interesting application-driven data types are numerical types. Here, an application-driven data type is some "mental" intersection between math and computer science in scientific computing and signal processing in particular. For example, the set of real-valued matrices with dimensions m times n, $\mathbb{R}^{m \times n}$, is an example of an application-driven data type. If the user is able to test the validity of a function argument against this application-driven data type, there is no need for the user to consider the distinction between Python floats, numpy generics, numpy ndarrays, and so on.

Existing Solutions

As mentioned in the introduction, there exist a number of solutions to validating function arguments in Python relying on a wide range of language constructs and interfaces and thereby representing a large variety of validation strategies. As these strategies are a source of inspiration for any new validation strategy, this section is used to briefly discuss some existing solutions with a focus on the three aspects which make up the suggested validation strategy: 1) The validation schemes that can be expressed and through that the abstraction level of the application-driven data types. 2) The way the interface of the implementation allows the validation scheme to be specified. 3) The Python constructs used to allow Python to validate the function arguments against the validation specification. Additionally, the relevant versions of Python are mentioned as 4) under each solution. Thus, the emphasis of this section is not to give a complete review of all existing solutions.

PyDBC

Although the original PyDBC[4] is long outdated, it represents an approach worth mentioning. The package allows so-called contracts to be specified using method preconditions, method postconditions, and class invariants. Thus, function argument validation can be performed using method preconditions. In the following example, the function argument, `a`, of the function, `exemplify` is validated to be a real scalar in the range $[0; 1]$:

```
import dbc
__metaclass__ = dbc.DBC

class Example:
    def exemplify(self, a):
        pass  # do something

    def exemplify__pre(self, a):
        assert isinstance(a, float)
        assert 0 <= a <= 1
```

When an invalid value is passed, the following assertion error occurs:

```
>>> example = Example()
>>> example.exemplify(-0.5)
AssertionError
```

As for validation strategy, the following observations are made:

1) As shown in the example above, the validation function, `exemplify__pre` contains custom validity checks, as PyDBC does not include any functionality for specifying a validation scheme.
2) Without any functionality for specifying a validation scheme, there is no fixed interface, and the user instead writes a number of `assert` statements to validate the function arguments.
3) The Python constructs used rely on object oriented Python by using metaclasses. When the metaclass creates the class, it rewrites the function `exemplify` to first invoke the function named `exemplify__pre` when `exemplify` is called following a fixed naming scheme.
4) PyDBC was intended for Python 2.2 and has not been changed since 2005, but the package does work with Python 2.7. It does, however, not work with Python 3, but the same functionality could indeed be implemented in Python 3.

Traits, Traitlets, and Numtraits

Traits[5] is an extensive package by Enthought which provides class attributes with the additional characteristics of customisable initialisation, validation, delegation, notification, and even

3. See https://docs.python.org/2/glossary.html#term-abstract-base-class

4. See http://www.nongnu.org/pydbc/

visualisation. Traitlets[6] is a lightweight Traits-like module which provides customisable validation, default values, and notification. Finally, Numtraits[7] adds to Traitlets with a numerical trait with more versatility in validation than that of the numerical traits of Traitlets. Thus, although hardly as intended by the developers, function argument validation can be performed using an attribute for each function argument. In the following example, the function argument, `a`, of the function, `exemplify` is validated to be a real scalar in the range $[0; 1]$:

```
from numtraits import NumericalTrait
from traitlets import HasTraits

class Example(HasTraits):
    _a = NumericalTrait(ndim=0, domain=(0, 1))

    def exemplify(self, a):
        self._a = a

        pass  # do something
```

When an invalid value is passed, the following assertion error occurs:

```
>>> example = Example()
>>> example.exemplify(-0.5)
traitlets.traitlets.TraitError: _a should be in \
the range [0:1]
```

As for validation strategy, the following observations are made:

1) The validation scheme of Traitlets requires specifying a static Python type, allows specifying a valid range of values for numerical types, and allows specifying relevant properties for other specific types. Furthermore, the validation scheme of the numerical trait of Numtraits does not require specifying a static Python type but allows specifying the number of dimensions and the shape of a value.
2) As shown in the example above, the interface of the implementation lets the user specify the validation scheme using a single call for each function argument with named arguments, named keyword arguments and in some cases unspecified keyword arguments using `**kwargs`.
3) The Python constructs used rely on object oriented Python by using descriptors which modify the retrieving and modification of attribute values of objects. Thus, when assigning a new value to an attribute, the relevant descriptor validates the new value.
4) Traitlets and Numtraits work with Python 2.7 and with Python 3.3 or above.

Annotations, Type Hints, and MyPy

PEP 3107[8] is a Python enhancement proposal on function annotations which is a feature which has recently been added to Python. This PEP allows arbitrary annotations without assigning any meaning to the particular annotations. PEP 484[9] is a PEP on type hints which attach a certain meaning to particular annotations to hint the type of argument values and return values of functions. The most important goal of this is static analysis, but runtime type checking is mentioned as a potential goal also. For more information, see PEP 483[10] on the theory of type hints and PEP 482[11] for a literature overview for type hints. MyPy[12] is a static type checker which, thus, does not enforce data type conformance at runtime. In the following example, the function argument, `a`, of the function, `exemplify` is validated to be a real scalar:

```
def exemplify(a: float):
    pass  # do something

exemplify('0')
```

When the script above is passed to MyPy using Python 3.5, the following message is produced:

```
$ mypy example.py
example.py:4: error: Argument 1 to "exemplify" has \
incompatible type "str"; expected "float"
```

As for validation strategy, the following observations are made:

1) The validation scheme of MyPy requires specifying a static Python type or a union of static Python types. This is hardly surprising for a static type checker.
2) As mentioned, the syntax of annotations is given by PEP 3107, and the format of the type hints is given by PEP 484 making the type hints explicit and readable although a less well-known feature of Python.
3) The Python constructs used rely only on annotations and runs offline and separately of normal execution of Python code.
4) PEP 484 was accepted for Python 3.5, but the syntax is compatible with that of PEP 3107 which was accepted for Python 3.0, and thus MyPy works with Python 3.2 or above. Furthermore, PEP 484 suggests a syntax for Python 2.7 using comments instead of annotations, and MyPy supports this and thus also works with Python 2.7.

PyValid

As the name suggests, PyValid[13] is a Python validation package, and it allows validation of function arguments and function return values. In the following example, the function argument, `a`, of the function, `exemplify` is validated to be a real scalar:

```
from pyvalid import accepts

@accepts(float)
def exemplify(a):
    pass  # do something
```

When an invalid value is passed, the following assertion error occurs:

```
>>> exemplify(0)
pyvalid.__exceptions.ArgumentValidationError: The \
1st argument of exemplify() is not in a \
[<type 'float'>]
```

As for validation strategy, the following observations are made:

1) The validation scheme for PyValid requires specifying one or more static Python types and acts as a runtime type checker. Thus, in terms of validation scheme capabilities, this is equivalent to MyPy.
2) As shown in the example above, the interface of the implementation lets the user specify the validation scheme

5. See http://docs.enthought.com/traits/
6. See http://traitlets.readthedocs.org/
7. See http://github.com/astrofrog/numtraits/
8. See https://www.python.org/dev/peps/pep-3107/
9. See https://www.python.org/dev/peps/pep-0484/
10. See https://www.python.org/dev/peps/pep-0483/
11. See https://www.python.org/dev/peps/pep-0482/
12. See http://mypy.readthedocs.org/
13. See http://uzumaxy.github.com/pyvalid/

using a single call for an entire function with a single argument or keyword argument for each validated function argument.
3) The Python constructs used rely on decorators by including an `accept` decorator in order to precede function execution by function argument validation.
4) PyValid works with Python 2.6 or above and with Python 3.

PyContracts

PyContracts[14] is a Python package that allows declaring constraints on function arguments and return values. In the following example, the function argument, `a`, of the function, `exemplify` is validated to be a real scalar in the range $[0; 1]$:

```
from contracts import contract

@contract(a='float,>=0,<=1')
def exemplify(a):
    pass  # do something
```

When an invalid value is passed, the following assertion error occurs:

```
>>> exemplify(-0.5)
contracts.interface.ContractNotRespected: Breach \
for argument 'a' to exemplify().
Condition -0.5 >= 0 not respected
checking: >=0              for value: Instance of \
<type 'float'>: -0.5
checking: float,>=0,<=1    for value: Instance of \
<type 'float'>: -0.5
Variables bound in inner context:
```

As for validation strategy, the following observations are made:

1) The capabilities of PyContracts allows specifying any conceivable validation scheme. This is achieved in part through built-in capabilities including specifying one or more static types in a flexible way, specifying value ranges, and specifying flexible length/shape constraints. And in part through custom specifications by using so-called custom contracts.
2) As shown in the example above, the interface of the implementation lets the user specify the validation scheme using a single call for an entire function with a single keyword argument for each validated function argument. The validation schemes for the individual arguments are specified using a custom string format. As the validation scheme becomes more advanced, the specification becomes less Python-like and less readable. For example, the following was taken from an official presentation and allows an argument to be a list containing a maximum of two types of objects: `list(type(t)|type(u))`.
3) The Python constructs used rely on decorators by including a `contract` decorator in order to precede function execution by function argument validation. Depending on the preference of the user, the validation scheme is either specified through arguments of the decorator, through annotations in the form of type hints or custom annotations, or through docstrings following a specific format.
4) PyContracts works with Python 2 and with Python 3.

14. See http://andreacensi.github.com/contracts/

The Suggested Python Validation Strategy

This section lays out a suggestion for a Python validation strategy for validating function arguments in signal processing applications. This strategy uses the introduced concept of application-driven data types and the observations made on the strategies of existing solutions. As mentioned in the previous section, the suggested validation strategy is made up of three aspects which are discussed separately in the following.

The Suggested Validation Schemes

As described in a previous section, we want to specify validation schemes in terms of application-driven data types rather than in terms of what a valid Python object *is*, *can do*, or *contains*. Needless to say, a translation must still be made from application-driven data types to Python data types, but this task is left for the validation package according to the suggested validation strategy. For an early implementation, any application-driven data type will allow only a limited set of Python data types. This does, however, not mean that the application-driven data type is limited to a few Python data types. Rather, more Python data types may be added along the way as long as they provide the necessary attributes with the desired interpretation. Thus, effectively, the suggested validation strategy can be considered less strict than static type checking but more strict than duck type checking.

The numerical trait of the Numtraits package has an interesting approach which is not too different from the concept of application-driven data types. The numerical trait does not distinguish between Python data types as long as they are numerical, and this corresponds to the most general numerical application-driven data type able to assume any numerical value of any shape. Furthermore, the numerical trait allows restricting the data type to more restrictive data types by specifying a number of dimensions, a specific shape, and/or a range of valid values. Indeed, signal processing applications could benefit from having such an application-driven data type. However, in some applications it may be necessary to work with boolean values, integral values, real values, or complex values only. Therefore, it should be possible to restrict the data type to suit these cases in addition to the other possible restrictions allowed by numerical traits.

To summarise, in Python signal processing applications, there should be an application-driven data type representing the most general numerical value being able to assume any numerical value of any shape. This data type should be able to be restricted to less general data types by specifying the mathematical set, the range or domain of valid values, the number of dimensions, and/or the specific shape of the data type. The suggested validation schemes should be expressed in terms of the desired application-driven data type.

The Suggested Interface Type

Most of the existing solutions which were mentioned in the previous section specify the validation scheme of all function arguments of a function in a single call to the validation package in question. This is not the case with the traits of the Trailets and Numtraits packages which only specify the validation scheme of a single function argument in each call to the validation package. From the perspective of the authors, the latter approach yields the better readability. Therefore, the suggested interface type should only let the user specify the validation scheme of a single function argument in each call.

As for the specifics of the interface, the validation scheme must be easy both for the programmer to state and for users to read. The PyContracts details its own format where the validation scheme is given by a string. However, it would be desirable to use a more standard Python interface to ease the usages even if it means having to be more verbose. On the other hand, the numerical trait of the Numtraits package uses named named arguments and keyword arguments which relate to the possible restrictions of the application-driven data types. From the perspective of the authors, the latter approach works well with application-driven data types and result in logical, easy to use interfaces. Therefore, the suggested interface should use named arguments and keyword arguments related to the possible restrictions of the general numerical application-driven data type to specify the validation scheme of function arguments.

The Suggested Python Constructs to Use

There are a lot of Python constructs which could potentially be used as showcased by the existing solutions. PyContracts allows the user to specify the validation scheme through the docstring of a function. However, most users would not expect docstrings to be parsed to yield the validation scheme, and furthermore the format used to specify the validation scheme would not be obvious because of the lack of restrictions put on docstrings. Therefore, docstrings are not suggested as a Python construct to use here. Annotations, as used by MyPy, are relatively new to Python, but that should not disqualify them from being used. However, the format used would not be obvious because there are few restrictions put on annotations so with the exception of type hints which are insufficient for this purpose. Therefore, annotations are not suggested as a Python construct to use here.

Next, there are the object oriented Python constructs. Metaclasses, as used by, PyDBC, have existed for a long time. However, these have changed over time, and so the metaclass attribute feature of Python 2 no longer works in Python 3, and only one metaclass is allowed per class in the more recent Python versions. Furthermore, the behaviour of metaclasses makes them impair the readability, especially to users that are unfamiliar with the construct. Therefore, metaclasses are not suggested as a Python construct to use here. Descriptors, as used by Traits, Traitlets, and Numtraits, are another feature applicable to object oriented Python, and these can provide flexibility and readability. However, they are limited to object oriented Python, and furthermore it seems unpythonic to validate function arguments by invoking descriptors through class instance attribute assignment. Therefore, descriptors are not suggested as a Python construct to use here.

Decorators, as used by PyValid and PyContracts, are a well-known and general Python construct. However, it is not immediately apparent if something goes on "under the hood", and the pythonic approach is to specify the validation scheme of all function arguments in a single decorator call, both of which affect readability. Therefore, decorators are not suggested as a Python construct to use here.

The suggested Python construct values explicit over implicit and promotes readability. The suggestion is to define and explicitly call a nested validation function with no arguments. There are a number of obvious alternatives which are not suggested for different reasons:

- It is not suggested to precede the function code by calls directly to a validation package because this does not clearly separate validation from the rest of the code.
- It is not suggested to use arguments for the validation function because this could potentially lead to error-prone validation if the validation function arguments are wrongly named or ordered, or the function arguments are renamed or reordered.
- It is not suggested to use a global rather than nested validation function because this could potentially separate the validation from the function and thus reduce readability.

Magni Reference Implementation

A reference implementation of the **suggested validation strategy** is made available by the open source Magni Python package [OPA+14] through the subpackage `magni.utils.validation`. The subpackage contains the following functions:

```
decorate_validation(func)
disable_validation()
validate_generic(
    name, type_, value_in=None, len_=None,
    keys_in=None, has_keys=None, ignore_none=False,
    var=None)
validate_levels(name, levels)
validate_numeric(
    name, type_, range_='[-inf;inf]', shape=(),
    precision=None, ignore_none=False, var=None)
```

Of these, `validate_generic` and `validate_levels` are concerned with validating objects outside the scope of the present paper. The function, `disable_validation` can be used to disable validation globally. Although discouraged, this can be done to remove the overhead of validating function arguments. As the name suggests, `decorate_validation` is a decorator, and this should be used to decorate every validation function with the sole purpose of being able to disable validation. Using the suggested validation strategy with Magni, the following structure is used for all validation adhering to **the suggested Python constructs to use**:

```
from magni.utils.validation import decorate_validation

def func(*args, **kwargs):
    @decorate_validation
    def validate_input():
        pass  # validation calls

    validate_input()

    pass  # the body of func
```

The remaining function, `validate_numeric`, is used to validate numeric objects based on application-driven data types as proposed by **the suggested validation scheme** of the validation strategy. This is done using the interface as proposed by **the suggested interface type** of the validation strategy: The `type_` argument is used for specifying one or more of the `boolean`, `integer`, `floating`, and `complex` subtype specifiers. The `range_` argument is used for specifying the set of valid values with a minimum value and a maximum value both of which may be included or excluded. The `shape` argument is used for specifying the shape with the entry, -1 allowing an arbitrary shape for a given dimension and any non-negative entry giving a fixed shape for a given dimension.

The remaining arguments of `validate_numeric` are not directly related to the validation scheme but rather to the surrounding Python code. The `precision` argument is used for specifying one or more allowed precisions in terms of bits per

value. The `name` argument is used for specifying which argument of the function to validate with the particular validation call. The `ignore_none` argument is a flag indicating if the validation call should ignore `None` objects and thereby accept them as valid. The `var` argument is irrelevant to the scope of the present paper and the reader is referred to the documentation for more information.

Additional resources for `magni` are:

- Official releases: doi:10.5278/VBN/MISC/Magni
- Online documentation: http://magni.readthedocs.io
- GitHub repository: https://github.com/SIP-AAU/Magni

Examples

As mentioned in relation to the suggested validation schemes, there should be an application-driven data type representing the most general numerical value being able to assume any numerical value of any shape. The following example validates a variable against exactly this application-driven data type. The validation only fails when a non-numerical object is passed as argument to `func`.

```
from magni.utils.validation import decorate_validation
from magni.utils.validation import validate_numeric
import numpy as np

def func(var):
    @decorate_validation
    def validate_input():
        all_types = ('boolean', 'integer',
                     'floating', 'complex')
        validate_numeric(
            'var', all_types, shape=None)

    validate_input()

    pass  # the body of the func
```

When valid values are passed, nothing happens:

```
>>> func(42)
>>> func(3.14)
>>> func(np.empty((5, 5), dtype=np.complex_))
```

However, when a non-numerical object is passed, the following exception occurs:

```
>>> func('string')
TypeError: The value(s) of >>var<<, 'string', must \
be numeric.
```

In the next example, the application-driven data type is any non-negative real scalar, i.e., $\mathbb{R}_{\geq 0}$.

```
from magni.utils.validation import decorate_validation
from magni.utils.validation import validate_numeric

def func(var):
    @decorate_validation
    def validate_input():
        real = ('integer', 'floating')
        validate_numeric(
            'var', real, range_='[0;inf]')

    validate_input()

    pass  # the body of the func
```

When valid values are passed, nothing happens:

```
>>> func(0)
>>> func(3.14)
```

However, when a complex object or a negative float is passed, the following exception occurs:

```
>>> func(1j)
TypeError: The value(s) of >>var.dtype<<, \
<type 'complex'>, must be in ('integer', 'floating').

>>> func(-3.14)
ValueError: The value(s) of >>min(real(var))<<, \
-3.14, must be >= 0.
```

Notice, that the `range_` argument in the validation call of the previous includes the values zero and infinity using `[...]`. One or both of these values could be excluded using `(...)` or `]...[` as is the case in the next example, i.e., $\mathbb{R}_{>0}$.

```
from magni.utils.validation import decorate_validation
from magni.utils.validation import validate_numeric

def func(var):
    @decorate_validation
    def validate_input():
        real = ('integer', 'floating')
        validate_numeric(
            'var', real, range_='(0;inf)')

    validate_input()

    pass  # the body of the func
```

When a valid value is passed, nothing happens:

```
>>> func(3.14)
```

However, when a zero-valued object is passed, the following exception occurs:

```
>>> func(0.)
ValueError: The value(s) of >>min(real(var))<<, \
0.0, must be > 0.
```

In the final example, the application-driven data type is any real matrix with its first dimension equal to 5, i.e. $\mathbb{R}^{5\times n}$ for any non-negative integer n.

```
from magni.utils.validation import decorate_validation
from magni.utils.validation import validate_numeric
import numpy as np

def func(var):
    @decorate_validation
    def validate_input():
        real = ('integer', 'floating')
        validate_numeric(
            'var', real, shape=(5, -1))

    validate_input()

    pass  # the body of the func
```

When a valid value is passed, nothing happens:

```
>>> func(np.empty((5, 5)))
>>> func(np.empty((5, 10)))
```

However, when an $\mathbb{R}^{10\times 5}$ object or an $\mathbb{R}^{5\times 5\times 5}$ object is passed, the following exception occurs:

```
>>> func(np.empty((10, 5)))
ValueError: The value(s) of>>var.shape[0]<<, 10, \
must be 5.

>>> func(np.empty((5, 5, 5)))
ValueError: The value(s) of >>len(var.shape)<<, 3, \
must be 2.
```

Requirements

The required dependencies for `magni` (as of version 1.4.0) are:

- Python >= 2.7 / 3.3
- Matplotlib [Hun07] (Tested on version >= 1.3)

- NumPy [vdWCV11] (Tested on version >= 1.8)
- PyTables[15] (Tested on version >= 3.1)
- SciPy [Oli07] (Tested on version >= 0.14)

It should be noted that the requirements other than Python and NumPy are due to `magni` rather than `magni.utils.validation`. In addition to the above requirements, `magni` has a number of optional dependencies but none of these are relevant to the usage of `magni.utils.validation`.

Quality Assurance

The Magni Python package has been developed according to best practices for developing scientific software [WAB+14], and every included piece of code has been reviewed by at least one person other than its author. Furthermore, the PEP 8[16] style guide is adhered to, no function has a cyclomatic complexity [McC76] exceeding 10, the code is fully documented, and an extensive test suite accompanies the package. More details about the quality assurance of `magni` is given in [OPA+14].

Conclusions

We have argued that function arguments should be validated according to data types at a higher abstraction level than actual Python types, and we have named these application-driven data types. Based on a discussion of existing validation solutions, we have suggested a Python validation strategy including three aspects: 1) The validation schemes that can be expressed. 2) The way the interface of the implementation allows the validation scheme to be specified. 3) The Python constructs used to allow Python to validate the function arguments. A reference implementation of this strategy is available in the open source Magni Python package which we have presented along with a number of examples. In short, `magni` and more generally the validation strategy should be used to abstract function argument validation from Python to signal processing, to make validation ease to write, and to enhance readability of validation.

Acknowledgements

This work was supported in part by the Danish Council for Independent Research (DFF/FTP) under Project 1335-00278B/12-134971 and in part by the Danish e-Infrastructure Cooperation (DeIC) under Project DeIC2013.12.23.

References

[CVS13] José Cordeiro, Maria Virvou, and Boris Shishkov. *Software and Data Technologies: 5th International Conference, ICSOFT 2010, Athens, Greece, July 22-24, 2010. Revised Selected Papers*. Springer, 2013.

[Hun07] John D. Hunter. Matplotlib: A 2D Graphics Environment. *Computing in Science & Engineering*, 9(3):90–95, May 2007. doi:10.1109/MCSE.2007.55.

[McC76] Thomas J. McCabe. A Complexity Measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, December 1976. doi:10.1109/TSE.1976.233837.

[Oli07] Travis E. Oliphant. Python for Scientific Computing. *Computing in Science & Engineering*, 9(3):10–20, May 2007. doi:10.1109/MCSE.2007.58.

[OPA+14] Christian Schou Oxvig, Patrick Steffen Pedersen, Thomas Arildsen, Jan Østergaard, and Torben Larsen. Magni: A Python Package for Compressive Sampling and Reconstruction of Atomic Force Microscopy Images. *Journal of Open Research Software*, 2(1):e29, October 2014. doi:10.5334/jors.bk.

[vdWCV11] Stéfan van der Walt, S. Chris Colbert, and Gaël Varoquaux. The NumPy Array: A Structure for Efficient Numerical Computation. *Computing in Science & Engineering*, 13(2):22–30, March 2011. doi:10.1109/MCSE.2011.37.

[WAB+14] Greg Wilson, D. A. Aruliah, C. Titus Brown, Neil P. Chue Hong, Matt Davis, Richard T. Guy, Steven H. D. Haddock, Kathryn D. Huff, Ian M. Mitchell, Mark D. Plumbley, Ben Waugh, Ethan P. White, and Paul Wilson. Best Practices for Scientific Computing. *PloS Biology*, 12(1):e1001745, January 2014. doi:10.1371/journal.pbio.1001745.

15. See http://www.pytables.org/
16. See https://www.python.org/dev/peps/pep-0008/
17. See https://travis-ci.org/

Spreading the Adoption of Python in India: the FOSSEE Python Project

Prabhu Ramachandran[‡§*]

https://youtu.be/6UnuPhTPdnM

Abstract—The FOSSEE (Free Open Source Software for Science and Engineering Education) project (http://fossee.in) is funded by the Ministry of Human Resources and Development, MHRD, (http://mhrd.gov.in) of the Government of India. The FOSSEE project is based out of IIT Bombay and the goal of the project is to eliminate the use of proprietary tools in the college curriculum. FOSSEE promotes various open source packages. Python is one of them.

In this paper, the Python-related activities and initiatives of FOSSEE are discussed. The group focuses on promoting the use of Python in the college curriculum. The important activities of this group include the creation of spoken-tutorials on Python, the creation of 400+ IPython-based textbook companions, an online testing tool for a variety of programming languages, a course akin to software carpentry at IIT Bombay, the organization of the SciPy India conference, and finally spreading the adoption of Python in schools and colleges. The paper discusses how these tools may be used to teach Python in the context of collegiate education and computational science.

Introduction

The FOSSEE project (http://fossee.in) started in 2009 with the goal of helping minimize the use of proprietary software in the college curriculum in India. The project is funded by the Ministry of Human Resources and Development, MHRD (http://mhrd.gov.in) of the Government of India. FOSSEE is part of the MHRD's National Mission on Education through ICT (NMEICT). NMEICT started in 2009 as an initiative to improve the quality of education in India. As part of this mission there have been several initiatives. One important example is the NPTEL project (http://nptel.ac.in) which provides content for over 900 courses at the undergraduate and graduate level (400 web-based and 500 video-based) online. These are proving to be extremely useful all over the country. Other projects include the Spoken Tutorial project (http://spoken-tutorial.org) which has also been previously presented at SciPy 2014 [kmm14]. FOSSEE is one such project that is the outcome of the NMEICT funding.

The FOSSEE project is based out of IIT Bombay and promotes the use of various open source packages in order to help eliminate the use of proprietary packages in the curriculum. A large number of colleges tend to unnecessarily purchase commercial licenses when they really do not need it. The difficulty with using commercial packages to teach basic concepts and computational techniques is well known:

- The packages are typically expensive, the money could be better spent on equipment. This is especially relevant in India.
- Students cannot legally take the software with them home or after they complete their course.
- Academic licenses are not enough as the students end up becoming dependent on the packages after the leave the institution. Furthermore, the packages are even more expensive when used in a commercial setting.

In order to help reduce the dependence on commercial packages, the FOSSEE project's efforts are focused towards training students and teachers to use FOSS tools for their curricular activities. This also requires development efforts in order to either enhance existing projects or fill in any areas where FOSS tools are lacking. There are around ten PIs actively involved in various sub-projects. Some of the most active projects are Scilab, Python, eSim (an Electronic Design Automation tool), OpenFOAM, and Osdag (open source design of steel structures).

After the initial efforts in 2009 and 2010 it was found that some of the initiatives worked and scaled up well whereas others did not. As a result, all of the FOSSEE sub-projects follow a similar structure. Typically each sub-project produces the following output:

- Generates "spoken-tutorials" that new users can use to self-learn a particular software package.
- Organize a crowd-sourced development of "textbook companions" for popular textbooks used in the curriculum. A textbook companion is created by coding every solved example in a text using a particular open source software package like Scilab or Python.
- Support user questions on a forum for the packages that are promoted.
- Develop new software that is useful in a particular domain.
- Support hardware interfacing to encourage open experimentation.
- Migrate labs that use proprietary packages and help them switch to a FOSS equivalent.
- Conduct workshops and conferences to spread the word and teach students and teachers.

Some of these are project specific. For example, the Scilab project is able to perform lab migrations as Scilab is a close

* Corresponding author: prabhu@aero.iitb.ac.in
‡ Department of Aerospace Engineering
§ IIT Bombay, Mumbai, India

equivalent to Matlab and this makes it easier for people to switch to it from Matlab. Kannnan Moudgalya's paper in 2014 [kmm14] discusses in detail the approach and design decisions made by the FOSSEE and spoken-tutorials projects. In particular the paper discusses spoken tutorials, textbook companions, and lab migrations.

The focus of the present paper is to elucidate some the Python-specific activities [FOSSEE-Python] that are of potential direct interest to the SciPy community. The overarching goal of the Python related activities is to help spread the use of Python in the curriculum. In 2016, the scope has expanded to help spread the use of Python in high-schools as well. The focus of this paper is to discuss the approaches that have been taken by the FOSSEE-Python group towards these goals. The lessons and approaches taken by this project are potentially of benefit for similar projects around the world.

The paper starts by discussing spoken tutorials, which are carefully created screencasts that are well suited for self learning. The paper then discusses a variant of the software carpentry course that has been offered at IIT Bombay. The subsequent section discusses a convenient tool called Yaksh for assessing programming skills of students. The approach taken to create a large amount of user-created documentation in the form of "textbook companions" is then discussed. The paper concludes with some brief information about the SciPy India conference series which is organized by FOSSEE and how that fits in with the overarching theme.

Spoken-tutorials

When the project started in 2009, many live workshops were conducted to teach Python but this proved to be too time consuming and did not scale. There are more than 3000 colleges in the country and live workshops cannot reach all of these institutions. At this time it was felt that preparing self-learning material that students can learn on their own would be much more effective and scalable. A sister project, the spoken-tutorial project (http://spoken-tutorial.org) pioneered the generation and dissemination of spoken-tutorials. A spoken tutorial is basically a carefully designed screencast for a roughly 10 minute duration or less. Any screencast cannot qualify as a spoken-tutorial. A spoken tutorial requires a carefully written script. Notably, a spoken tutorial should be made such that a novice can understand it. The spoken-tutorial project ensures that all new tutorials undergo a novice check to make sure that this is indeed the case. This involves asking a novice to go over the script and ensure that they are able to reproduce the entire script and follow it. This carefully written script allows a spoken tutorial to be dubbed into multiple languages. A series of spoken tutorials can thus be used to effectively teach a programming language or software package. As such, a spoken tutorial is not a substitute for classroom instruction of the traditional kind. It has been most effectively used to teach a programming language or introduce a software package.

The major advantage of the spoken tutorial is that it retains a high quality of instruction, can be used for self-learning, and scales extremely well. In addition, these tutorials can be dubbed into various local languages. The spoken tutorial project has trained over a million students and teachers on a variety of software packages. The project hosts over 700 individual spoken-tutorials. Over 20 different Indian languages are supported.

As part of the Python initiative the FOSSEE Python group has created about 40 spoken tutorials to teach non-CSE undergraduate students how to use Python for their curricular computational tasks. A new set of around 50 tutorials is currently being recorded. The spoken tutorials include tutorials on starting with IPython, plotting with matplotlib, etc. Currently these are only available in English.

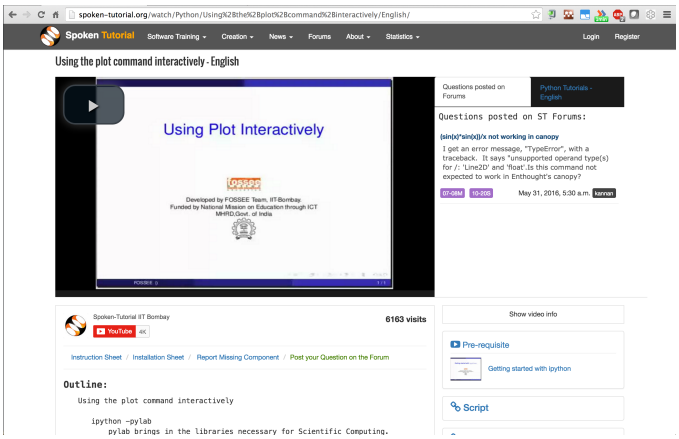

Fig. 1: An example of a Python spoken tutorial. The video can be viewed, an outline of the material is available below the video. An instruction sheet and installation sheet is also available. Prerequisite videos are listed and users can also post questions on a forum.

Fig. 1 shows a typical Python spoken tutorial as hosted on the spoken-tutorial website. It shows the main screencast video. Below the video is an outline of the tutorial. Information on installation and other instructions is also listed. Users can easily navigate to prerequisite tutorials. In addition, users can post their questions on the forum.

These spoken tutorials can be accessed by anyone and can also be downloaded into a self-contained CD by users. Around 40000 users have gone over this material. Detailed statistics for the various tutorials are available here: http://spoken-tutorial.org/statistics/training/

The FOSSEE team generates the spoken tutorials and the spoken tutorial team coordinates the conduct of workshops where students use this material to teach themselves Python. FOSSEE staff members support these workshops by attending to user questions that may arise.

Spoken tutorials have thus become an effective way to scale up training on open source packages. For a motivated and skilled user, spoken-tutorials and documentation alone are often enough to self-learn. However, this is not enough for the average user. There are many software packages, tools, web sites and books related to computational science. It is never easy for a student (undergraduate or graduate) to choose the right set of packages or practices they should follow. The next section discusses a course that is designed and run by the FOSSEE group at IIT Bombay that helps address this.

The SDES course

SDES is an acronym that stands for Software Development Techniques for Engineers and Scientists. As discussed earlier, the Python group initially focused on teaching Python at various colleges. It was soon felt that students needed to learn how to use Unix shells effectively, use version control, basic knowledge of LaTeX, good software development practices in addition to Python. Students are often unaware of the right set of tools

to learn. Most students undergo a basic computer programming course in their first year but this is rarely enough for them to perform their curricular tasks.

In order to fill this need, a course was designed in late 2009. The course is titled Software Development techniques for Engineers and Scientists (SDES). This course takes inspiration from the Software Carpentry Course material [SWC]. However, the course is tailored for undergraduate students. The course is offered at IIT Bombay so students at the undergraduate and graduate levels could take this as part of their course-work. Students can certainly learn this material from several online resources, however, the existence of this course allows students to credit this as part of their course requirements.

The course starts with teaching students on how to use Unix command line tools to carry out common (mostly text processing) tasks. The course then goes on to teach students how to automate typical tasks using basic shell-scripting. The students are then taught version control. The course originally used mercurial, however, this has changed to git. The students are then taught basic and advanced Python. The emphasis is on typical engineering/numerical computations such as those that involve (basic) manipulation of large arrays in an efficient manner. Good programming style is discussed along with debugging and test driven development. They also learn LaTeX and document creation with reStructuredText. The course material is available from github, at http://github.com/FOSSEE/sees.

As part of the evaluation, students pick a software project and attempt to apply all that they have learned. Students are also given many programming assignments to test their ability to program. We have built a convenient online testing tool called Yaksh that is discussed in a subsequent section for this task. This makes the examinations interesting for students and is helpful for instructors to assess student's understanding.

The course has been offered twice thus far and will be offered again in the fall of 2016. The course has been well received by students and is quite popular. The number of students is restricted to about 60 each time. During the last delivery it was felt that the student projects were not done well enough. A more aggressive and systematic approach is needed to push students to work consistently over the duration of the course, rather than in the last minute. It was also found that it is difficult for students and instructors to pick meaningful projects that are neither too trivial or too difficult. For the next delivery, the plan is to encourage students to work systematically on their projects. Studying the git logs of the student project repositories to assess team contribution and systematic work is one approach that is being considered. Instead of always picking new projects, one possibility is to give them an existing project and ask them to improve it.

The SDES course was offered as part of a 1000 teacher training course offered in 2011 at IIT Bombay. This course had over 600 participants who took the course and was well received. Unfortunately, it is not clear how well this course eventually helped teachers and if the teachers went on to teach this material in their colleges.

Teaching the course has generally been enjoyable and rewarding. Students seem to find the course useful and generally continue to use the tools that they have learned. The course is rather demanding from the perspective of assessment and a good team of TAs is necessary. Fortunately, the FOSSEE Python team helps in this regard.

Online test tool: Yaksh

Assessing the programming skills of students is a very important task during training. This is necessary both from the perspective of effective teaching as well as learning. For an instructor, testing early and often is helpful because it provides immediate feedback on which students need help and which of them are doing well. For students, doing well in a test gives them confidence and doing poorly teaches them where they should concentrate harder or get help. Unfortunately, assessment is not usually a pleasant task. Assessment is doubly important when learning a programming language as in India there are students who learn how to program but never write more than a few lines of code. Programming requires practice and encouraging students to program is very important.

For FOSSEE this is also important from the perspective of being able to certify students. The Spoken Tutorial team conducts a large number of workshops all over the country and it would be good if the tests required that students be able to write simple programs at least.

In 2011, the author saw Chris Boesch run a programming contest at PyCon APAC 2011. The contest was entirely online, and users could submit their code and obtained instant feedback. The system was built on top of Google App Engine. This made testing programming lively and enjoyable. The author along with the FOSSEE team have built a Django application to do something similar. The package is called Yaksh, is Open Source, and the sources are available at http://github.com/FOSSEE/online_test. The initial version of Yaksh was used to administer programming quizzes for the online teacher training course based on the SDES course in late 2011. More than 600 simultaneous users took their tests on this interface. This work was presented at SciPy India 2011 [PR11].

Yaksh provides a simple interface for an instructor to create a question paper with mutiple-choice questions (MCQ) as well as full-fledged programming questions. A programming question consists of a problem statement and the user writes the code on the interface. This code is immediately checked against several test cases and any failures are reported directly to the user by providing a suitable traceback. By design, a programming question can be answered many times until the user gets it completely correct. This encourages students to try and submit their answers. An MCQ can only be answered once for obvious reasons.

It was found that the approach of allowing multiple submissions and providing instant feedback instead of the traditional approach where a student would upload the answers on an interface and obtain the grades later to be much more effective. Instant feedback makes the process lively and entertaining for the student. The ability to submit multiple times gives them comfort in that they know that they can gradually fix their code. This makes students less anxious. They also immediately know that their answer is correct if they get it right. This makes a significant difference. Clearly this is not enough to teach all aspects of programming, however, this is a very useful aid.

Yaksh provides a convenient monitoring interface for the instructor which provides, at a glance, information on the students' performance. Each submission of a student is logged and can be seen by the moderator. This is useful for an instructor.

Yaksh works best with Python since it has been used mostly for Python tests but does support multiple other programming languages like C, C++, Java, Bash, and Scilab.

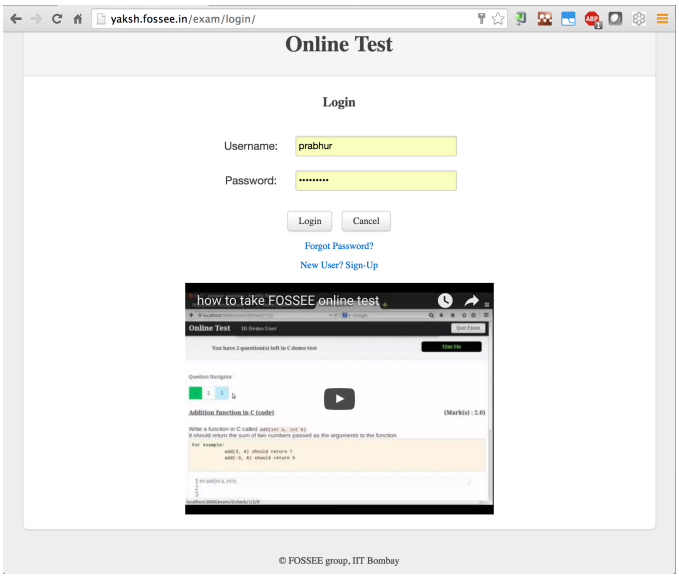

Fig. 2: *The Yaksh application login screen with a video on how one can use it.*

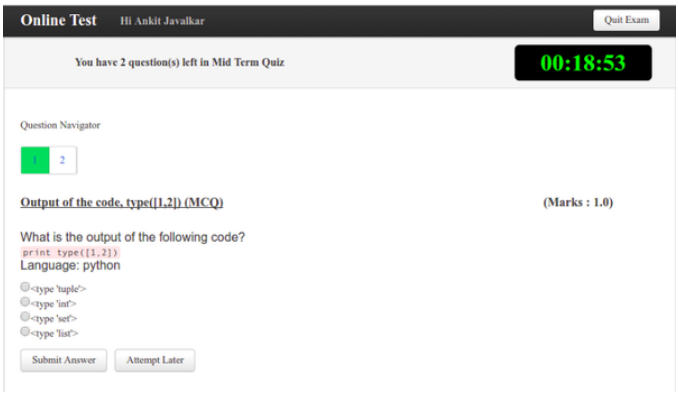

Fig. 3: *The interface for a multiple-choice question on yaksh.*

Yaksh sandboxes the user code and runs the code as "nobody" when configured to do so. The code execution can also be performed in a docker container. This minimizes any damage a student can do. Since all answers are logged before execution, it is easy to find out if a student has been malicious -- this has never happened in the current usage Yaksh.

Fig. 2 shows the login screen for Yaksh, which features a small video that demonstrates how the interface can be used. Fig. 3 shows the interface for an MCQ and Fig. 4 shows the interface for a programming question. The top bar shows the time remaining to take the question. A question navigator is provided for students to quickly move between questions.

Fig. 5 shows a typical moderator interface while monitoring a running quiz. The interface shows the number of questions each student has completed. On clicking on a user, all the answers they have submitted are visible.

Installation and running a demo

Yaksh is a Python package and is distributed on PyPI. Yaksh can be installed with pip. When installed, an executable script `yaksh` is created. To setup a demo instance on can run

```
$ yaksh create_demo
```

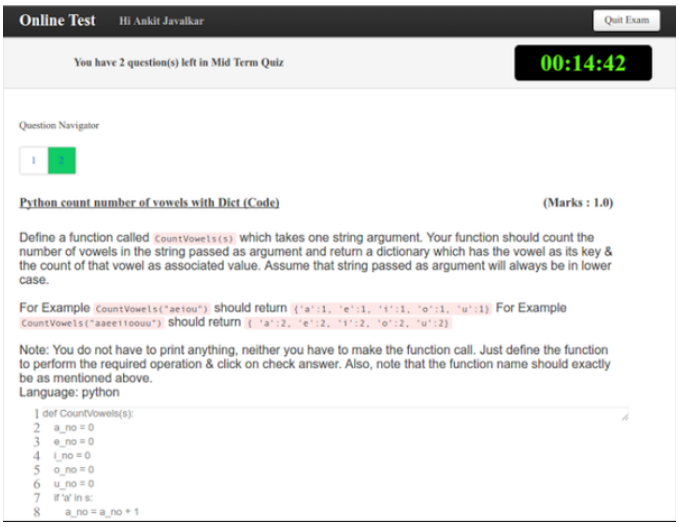

Fig. 4: *The interface for a programming question on yaksh.*

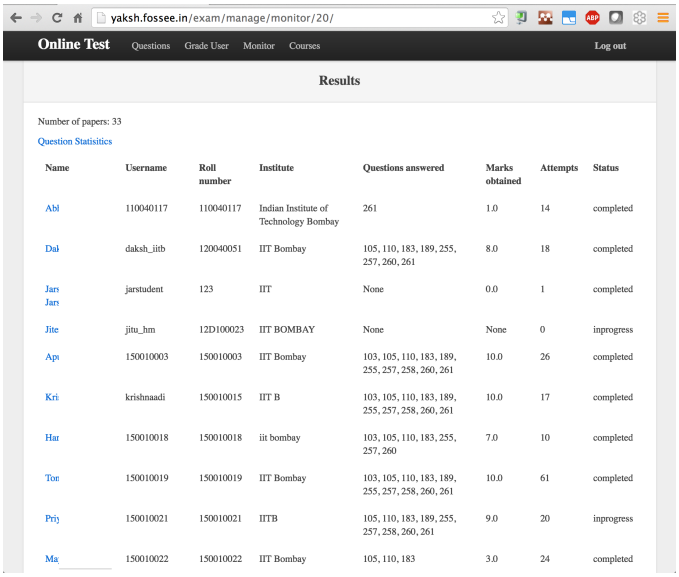

Fig. 5: *The moderator interface for monitoring student progress during an exam on yaksh.*

This creates a new demo Django project called `yaksh_demo` with a demo database and a couple of users added. One is a moderator and other is an examinee. It also loads a few simple demo questions and a quiz. One can then simply run:

```
$ yaksh run_demo
$ sudo yaksh run_code_server
```

This starts up a server on the `localhost` and also runs the code evaluator as nobody. The server is tested to work on Linux and OS X but not on Windows although technically it should not be difficult to do this. Note that a malicious user could fork bomb the machine in this case as the service is still running on the machine. Resource limiting is possible but not currently implemented.

The above instructions are only for a demo and are not suitable for a production installation as a sqlite database is used in the demo case. More detailed instructions for a production installation are available in the repository.

Design overview

In order to create a quiz the teacher/instructor (also called the moderator) must first create a course. Users can login and register for the course with the instructor's approval. The moderator can add any number of questions to yaksh through the online interface. These can be either MCQ questions or programming questions. The programming questions will require a set of test cases. In the case of a Python programming question, a simple question could be of the form:

```
Write a function called factorial(n) which takes
a single integer argument and returns the
factorial of the number given.
```

The question will also be accompanied with a few test cases of the form:

```
assert factorial(0) == 1
assert factorial(1) == 1
assert factorial(5) == 120
```

As many questions as desired may be created. For other languages assertions are not easily possible but standard input/output based questions are easily handled. More sophisticated test support is also possible (for example one could easily support some form of assertions for C/C++ if a template were used to generate the files). The architecture of yaksh supports this fairly easily.

Questions could also be imported from a Python script. The interface lets users export and import questions. The moderator then creates a quiz and an associated question paper. A quiz may have a pre-requisite quiz and can have a passing criterion. Quizzes have active durations and each question paper will have a particular time within which it must be completed. For example one could conduct a 15 minute quiz with a 30 minute activity window. The students can be allowed to attempt the quiz either once or multiple times as desired. This is often useful when teaching new users. Questions are automatically graded. A user either gets the full marks or zero if the tests fail. In the future yaksh will also support partial grading depending on the number of test cases the code passes.

In terms of the internal design, yaksh is fairly simple.

- The Django app manages the questions, quizzes, users etc.
- A separate code-server process runs as "nobody" to limit the amount of damage malicious code could have. This process runs an XML/RPC server. The Django app creates an XML/RPC `ServerProxy` instance and invokes the code server with the user code and any additional data (like the test cases etc.). This is executed by the server process.
- Unfortunately, XML/RPC can only handle 2 simultaneous connections. Therefore, a pool of these servers is created and managed. The Django app then connects to any available server and executes the code.
- In order to prevent issues with infinite loops, we use the `signal` module to send `SIGALRM` in a finite amount of time. The default is 2 seconds but this can be easily configured.

The code server can be easily run within a docker container and this is also supported by Yaksh. Some documentation for this is also provided in the production README.

In addition to these features yaksh also has an experimental web-API that allows an instructor to utilize yaksh from their own web sites or HTML documents. An instructor could create questions and a question paper from the yaksh interface but have users take the test on say an Jupyter notebook interface. This is still being developed but a proof of concept is available. In order to do this, a user could simply add `yaksh.js` to their HTML and call a few API methods to fetch as well as submit user answers.

Some experiences using yaksh

Yaksh has been used while delivering the SDES course at IIT Bombay. This has worked quite well and is well received by students. As mentioned before, Yaksh has also been used for the online course with over 600 participants and worked quite well. This was however done in 2011 and thereafter has only been used for smaller classes.

Recently, Yaksh was used by the author to teach first year undergraduate students Python as part of a data analysis and interpretation course. Many students were new to programming and a lot was learned about how well this could work.

Yaksh definitely made it much easier to assess the understanding of students. Initially the students were not given tests but were given Jupyter notebooks as well as exercises to solve at home. The assumption was that the students would follow the material since it was done slowly in class. This was not the case. A take-home assignment was given using Yaksh where students would solve simple problems (many taken from the exercise problems that were already given). Surprisingly, many of the students were struggled badly. Even the best students were not able to finish all problems. This showed that a lot more practice was needed. As a result, 7 different quizzes with a few problems each were conducted. After about 5 such quizzes it was found that some students were still having difficulties understanding basic concepts. These were students who were completely new to programming. Around 20 poorly performing students were identified. These students came to a special class and solved 10 problems using yaksh over the course of 2 hours. The monitoring facility was immensely useful as one could walk over to a struggling student and provide assistance or point a TA in their direction. The students all seemed to like the experience and understood the importance of actually programming versus learning the language syntax. Their performance in the subsequent quizzes and assignments improved significantly.

One major lesson learned was that one should ensure that students are tested from the get-go rather than towards the end. This would result in a much smoother experience. Based on the overall experience, it is clear that Yaksh is an effective tool for students and teachers alike.

Plans

Yaksh will continue to be improved based on the needs of the FOSSEE team and that of others. It is hoped that this is also of use to the community. The future goals for the yaksh project are to:

- Clean up and come up with a stable web-API.
- Support the use of Jupyter notebooks for tests.
- Support more programming languages.
- Integrate Yaksh into the spoken-tutorial website in order to help them test students.

Textbook companions

Spoken-tutorials allow FOSSEE to reach out to a larger audience and train students and teachers on the use of FOSS tools and

packages. The SDES course is similar to the Software Carpentry effort and offers a full-fledged course that readies students for computational science. Yaksh facilitates both of these by making it easier to test students on their programming skills.

While Python in general and the SciPy project in particular have plenty of good online documentation, this may not always be adequate from the perspective of a beginner. Good quality documentation is not easy to write and requires both expertise as well as the ability to explain things at the level of the user. This is often difficult for a developer who knows almost everything about the package. On the other hand it is not always easy for an inexperienced user to write documentation either.

Students are often interested in taking internships and desire to participate in software projects that are relevant to their area of interest. Is it possible to engage these students in a way where they are able to contribute meaningful documentation in an area of their interest?

Textbook companions offer an interesting approach in this context. As discussed in detail in [kmm14], textbook companions are created by writing Python code for every solved example in a textbook. Students create these textbook companions which are then reviewed by either teachers or reviewers at FOSSEE. This task scales very well as students are eager to take up the task. They already know the subject matter as the textbook is part of their curriculum. The examples are already solved, so they have to convert the solved example into appropriate Python code. Students are given an honorarium and a certificate after their textbooks pass a review. Currently, there are over 530 Scilab textbook companions [STC] created. The Python project has 416 completed books with over 200 textbooks in progress. The Python textbook companions are hosted online at http://tbc-python.fossee.in

The Python Textbook Companions (PTC's) are submitted in the form of IPython notebooks. This is important for several reasons:

- IPython notebooks allow one to put together formatted HTML, code, and the results in one self-contained file.
- IPython notebooks are easy to render and a HTML listing can be generated.
- The file can also be hosted online and interactively used.
- The huge popularity of the notebook makes this a very useful resource.

The FOSSEE group has also customized the generated HTML such that users can leave comments on the IPython notebooks. This is done by linking disqus comments to each rendered notebook. The disqus API is then queried for any new comments each day and contributors are sent a consolidated email about any potential comments for them to address. This feature is relatively new and needs more user testing.

The submission process and hosting of the IPython notebooks is done using a Django web application that can be seen at http://tbc-python.fossee.in. The code for the interface is also available from github (https://github.com/FOSSEE/Python-TBC-Interface). Once a textbook is reviewed it is also committed to a git repository on github: https://github.com/FOSSEE/Python-Textbook-Companions.

The process works as follows:

1) The student picks a few possible textbooks that have not been completed and informs the textbook companion coordinator.

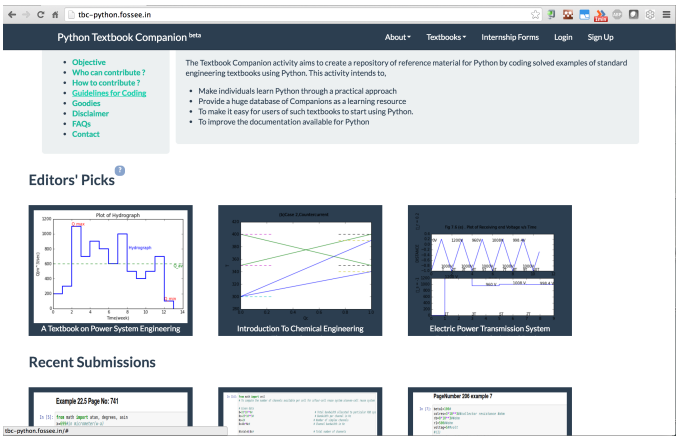

Fig. 6: *The Django application which hosts the Python textbook companions.*

2) Once a particular book is assigned to the contributor, the student submits one sample chapter which is reviewed by the coordinator.
3) The student then completes the entire book. Each chapter is submitted as a separate IPython notebook.
4) The student also uploads a few screenshots of their favorite notebooks that are displayed on the interface.
5) The submitted code is reviewed and any corrections are made by the contributor.
6) The notebooks are then committed to the git repository.
7) The completed notebooks are hosted by the TBC web application.

After the textbook is reviewed and accepted the student is sent an honorarium for their work. Fig. 6 shows the main Python TBC interface with information about the project and the editor's picks.

Approximately 3 proposals for new textbooks are submitted each week. Of these, around one is rejected if the book is either a programming language book or it is already completed. Initially many proposals were C or C++ programming books which were being converted to Python. This has since been discontinued and such books are no longer accepted. Of the submissions, around 70% of the submissions are from males, 40% of the submissions are by students, another 40% from teachers, and the remaining 20% from working professionals.

Fig. 7 shows a typical textbook. The IPython notebooks for each chapter can be viewed or downloaded. More information on the book itself can be seen including an ISBN search link for the student to learn more about a book, a link to the actual IPython notebook on github and other details are also available. The entire book can be downloaded as a ZIP file.

Upon clicking a chapter, a typical rendered HTML file is seen. This is seen in Fig. 8. A button to edit the chapter is seen, this will fire up a tmpnb instance which allows users to easily modify and run the code. This makes it convenient to view, modify, and learn the created content. In the figure, one can see an icon for entering comments. This links to a disqus comment field at the bottom of the page. This lists all current comments and allows users to submit new comments on the particular chapter.

A large number of solved examples are indeed quite simple but there are several that are fairly involved. Some of the nicer textbooks are highlighted in the editor's pick section.

Fig. 7: *A typical textbook is shown. The figure shows some screenshots to pique the interest of the casual reader. The Jupyter notebook corresponding to each chapter is listed and can be viewed or downloaded.*

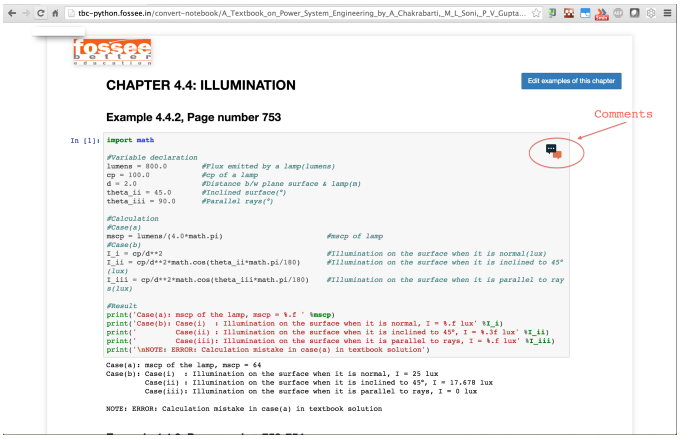

Fig. 8: *A typical textbook chapter being rendered. The button to edit examples of the chapter fires up a tmpnb instance so users can edit the code and try their changes.*

The Python textbook companion effort of FOSSEE has not been formally announced and advertised in the wider SciPy community. Once announced, the plan is to start to analyze the usage and popularity of this resource. It is still unclear as to how different people are using the notebooks. Some good feedback has been received from the contributors [testimonials] to the project. Many of them have enjoyed creating these notebooks and have benefited by this effort. Some contributor comments are quoted in [kmm14].

In summary, the textbook companions are of interest because:

1) They provide ready-to-use examples of how to apply a given software package or set of tools to a particular problem.
2) They scale well and can be easily crowd-sourced.
3) The scale of the current effort allows one to ask interesting questions, for example "what are the different uses of the FFT in science and engineering?".
4) It provides an interesting alternative to internships and projects for undergraduate students looking to learn and contribute something meaningful.

The texbook companions thus complement the other initiatives of the FOSSEE-Python group.

Scipy India

The SciPy India conference provides an opportunity for those interested in Python to learn of new developments, talk about how they have used Python, meet other interested users/developers and participate in the community.

The Python FOSSEE group has been organizing the SciPy India conference since 2009. Seven conferences have been organized thus far. The conferences have traditionally been held in December. They are largely funded by the FOSSEE project. The project staff manage the local organization almost completely. The conference website is at http://scipy.in

There is an attendance of about 200 people each year. A large number of these are new users. The conference is typically well received and many people are aware of the SciPy community through these efforts. Each year a leading expert in the community is invited to keynote at the conference. The first conference had Travis Oliphant keynote and the conference in 2015 had Andreas Kloeckner as the keynote. Several other important members of the extended SciPy community from India and abroad have spoken at the conference.

Originally, sprints were conducted but this did not prove very effective. The conference now focuses on high-quality tutorials for two days and a single day for the conference itself. Many college professors attend the conference and many go back and encourage their students to use the tools and participate in the future.

Plans for the future

The Python group plans to build on the existing work. The team will continue to generate textbook companions, provide support for the workshops conducted by the spoken-tutorial team, and continue to work on the Yaksh interface. The existing Python spoken tutorials will be updated and new ones will be created as required. These spoken tutorials will also be dubbed to other Indian languages.

In addition the Python group plans to promote the use of Python in the CBSE (Central Board of Secondary Education) school curriculum. The CBSE board has already included Python as an alternative to C++ in the 11th and 12th grade exams. Unfortunately, there is quite a bit of resistance towards this as many teachers are unfamiliar with Python. The plan is to support schools in this initiative over the next year. Textbook companions will be prepared for the school initiative. Spoken-tutorials tailor-made to the school curriculum will also be generated. This is an exciting new development but a significant amount of work is still necessary.

Conclusions

As discussed in this paper, the FOSSEE project has used several interesting approaches to spread Python in India. Spoken tutorials help deliver good-quality self-learning training material to a large audience. The SDES course allows students to learn effective computational skills as part of their curriculum. Yaksh is an open source tool that can be used to effectively test the programming skills of a student. Together, these tools and materials maybe be effectively used by instructors to teach computational tools and programming to a large number of students. The author's experience with using Yaksh while teaching students at different levels has also been shared. It seems that testing students often on their programming is an effective way to have them practice their programming skills and provide quick feedback to the instructor.

Textbook companions offer an interesting alternative to documentation and scales well. The very fact that FOSSEE has helped facilitate around 500+ textbook companions shows that this activity scales and has potential to make a difference.

The FOSSEE Python group has helped spread the use of Python in India. The group has also helped the other sister FOSSEE groups with respect to any Python related support when possible. It is hoped that the code and other material that has been generated is of use to the wider community across the world.

Acknowledgments

FOSSEE would not exist but for the continued support of MHRD and we are grateful to them for this. The project would not be a success without the efforts of the many PIs of the FOSSEE project especially Prof. Kannan Moudgalya of IIT Bombay who also leads the spoken-tutorial project. The author wishes to thank Asokan Pichai who helped shape the FOSSEE project over the first few years. This work would not be possible without the efforts of the many FOSSEE staff members. The past and present members of the project are listed here: http://python.fossee.in/about/ the author wishes to thank them all. The author wishes to thank the reviewers of this manuscript for their suggestions that have made this manuscript better.

REFERENCES

[kmm14] Kannan Moudgalya, Campaign for IT literacy through FOSS and Spoken Tutorials, Proceedings of the 13th Python in Science Conference, SciPy, July 2014.

[FOSSEE-Python] FOSSEE Python group website. http://python.fossee.in, last seen on June 2nd 2016.

[STC] Scilab Team at FOSSEE, Scilab textbook companions, http://scilab.in/Textbook_Companion_Project, May 2016.

[SWC] Greg Wilson. Software Carpentry, http://software-carpentry.org, Seen on May 2016.

[PR11] Prabhu Ramachandran. FOSSEE: Python and Education, Python for science and education, Scipy India 2011, 4th-11th December 2011, Mumbai India.

[testimonials] Python texbook companion testimonials. http://python.fossee.in/testimonials/1/ Seen on Jun 1, 2016

PySPH: a reproducible and high-performance framework for smoothed particle hydrodynamics

Prabhu Ramachandran[‡§*]

https://youtu.be/6UnuPhTPdnM

Abstract—Smoothed Particle Hydrodynamics (SPH) is a general purpose technique to numerically compute the solutions to partial differential equations such as those used to simulate fluid and solid mechanics. The method is grid-free and uses particles to discretize the various properties of interest (such as density, fluid velocity, pressure etc.). The method is Lagrangian and particles are moved with the local velocity.

PySPH is an open source framework for Smoothed Particle Hydrodynamics. It is implemented in a mix of Python and Cython. It is designed to be easy to use on multiple platforms, high-performance and support parallel execution. Users write pure-Python code and HPC code is generated on the fly, compiled, and executed. PySPH supports OpenMP and MPI for distributed computing, in a way that is transparent to the user. PySPH is also designed to make it easy to perform reproducible research. In this paper we discuss the design and implementation of PySPH.

Background and Introduction

SPH (Smoothed Particle Hydrodynamics) is a general purpose technique to numerically compute the solutions to partial differential equations used to simulate fluid and solid mechanics. The method is grid-free and uses particles to discretize the various properties of interest. The method is Lagrangian and particles are moved with the local velocity. The method was originally developed for astrophysical problems [Luc77], [GM77] (compressible gas-dynamics) but has since been extended to simulate incompressible fluids [Mon94], solid mechanics [GMS01], free-surface problems [Mon94] and a variety of other problems. Monaghan [Mon05], provides a good review of the method.

The SPH method is relatively easy to implement. This has resulted in a large number of schemes and implementations proposed by various researchers. SPH schemes differ in the details of how the governing equations are approximated. It is often difficult to reproduce published results due to the variety of implementations. While a few standard packages like SPHysics [devb], DualSPHysics [deva], JOSEPHINE [CPR12], GADGET-2 [Spr05] etc. exist, they are usually tailor-made for particular applications and are not general purpose. They are all implemented in FORTRAN (77 or 90) or C, and do not have a convenient Python interface.

* *Corresponding author: prabhu@aero.iitb.ac.in*
‡ *Department of Aerospace Engineering*
§ *IIT Bombay, Mumbai, India*

Our group has been developing PySPH (http://pysph.bitbucket.org) over the last 5 years. PySPH is open source, and distributed under the new BSD license. Our initial implementation was based on Cython [BBC+11] and also featured some parallelization using MPI. This was presented at SciPy 2010 [RK10]. Unfortunately, this previous version of PySPH proved difficult to use as users were forced to implement most of their code in Cython. This was not a matter of simply writing a few high performance functions in Cython. The PySPH library is object oriented and supporting a new SPH formulation would require subclassing one or more classes and this would need to be done with Cython. This made the design more rigid as all the types needed to be pre-defined. Writing all this in Cython meant that users had to manage compilation and linking the Cython code during development. This made development with PySPH inconvenient.

It was felt that we might as well have implemented the core library in C++ and exposed a Python interface to it. A traditional compiled language has more developer tooling around it. For example debugging, performance tuning, profiling would all be easier if everything were written in C or C++. Unfortunately, such a mixed code-base would not be as easy to use, extend or maintain as a largely pure Python library. In our experience, a pure Python library is a lot easier for say an undergraduate student to grasp and use over a C/C++ code. Others are also finding this to be true [Per15]. Many of the top US universities are teaching Python as their first language [Guo14]. This means that a Python library would also be easier for relatively inexperienced programmers. It is also true that a Python library would be easier and shorter to write for the other non-high-performance aspects (which is often a significant amount of code). So it seemed that our need for performance was going against our desire for an easy to use Python library that could be used by programmers who were not C/C++ developers.

In early 2013, we redesigned PySPH so that users were able to implement an entire simulation using pure Python. This was done by auto-generating HPC code from the pure Python code that users provided. This version ended up being faster than our original Cython implementation! Since we were auto-generating code, with a bit of additional effort it was possible to support OpenMP as well. The external user API did not change so users did not have to modify their code at all to benefit from this development. PySPH has thus matured into an easy to use, yet high-performance framework where users can develop their schemes in pure Python and yet obtain performance close to that of a lower-level language implementation. PySPH has always supported running on a cluster

of machines via MPI. This is seamless and a serial script using PySPH can be run with almost no changes using MPI.

PySPH features a reasonable test-suite and continuous integration servers are used to test it on Linux and Windows. The documentation is hosted at http://pysph.readthedocs.org. The framework supports several of the standard SPH schemes. A suite of about 30 examples are provided. These are shipped as part of the sources and installed when a user does a pip install. The examples are written in a way that makes it easy to extend and also perform comparisons between schemes. These features make PySPH well suited for reproducible numerical work. In fact one of the author's recent papers [RP16] was written such that every figure in the paper is automatically generated using PySPH.

In this paper we discuss the use, design, and implementation of PySPH. In the next section we provide a high-level overview of the SPH method.

Smoothed Particle Hydrodynamics

The SPH method works by approximating the identity:

$$f(x) = \int f(x')\delta(x-x')dx',$$

where, δ is the Dirac Delta distribution. This identity is approximated using:

$$f(x) \approx \int f(x')W(x-x',h)dx', \tag{1}$$

where W is a smooth and compact function and is called the kernel. It is an approximate Dirac delta distribution that is parametrized on the parameter h and $W \to \delta$ as $h \to 0$. h is called the smoothing length or smoothing radius of the kernel. The kernel typically will need to satisfy a few properties if this approximation is to be accurate. Notably, its area should be unity and if it is symmetric, it can be shown that the approximation is at least second order in h. The above equation can be discretized as,

$$f(x) \approx \langle f(x) \rangle = \sum_{j \in \mathcal{N}(x)} W(x-x_j,h) f(x_j) \Delta x_j, \tag{2}$$

where x_j is the position of the particle j, Δx_j is the volume associated with this particle. $\mathcal{N}(x)$ is the set of particle indices that are in the neighborhood of x. In SPH each particle carries a mass m and associated density ρ with it and the particle volume is typically chosen as $\Delta x_j = m_j/\rho_j$. This results in the following SPH approximation for a function,

$$< f(x) >= \sum_{j \in \mathcal{N}(x)} \frac{m_j}{\rho_j} W(x-x_j,h) f(x_j). \tag{3}$$

Derivatives of functions at a location x_i are readily approximated by taking the derivative of the smooth kernel. This results in,

$$\frac{\partial f_i}{\partial x_i} = \sum_{j \in \mathcal{N}(x)} \frac{m_j}{\rho_j}(f_j - f_i)\frac{\partial W_{ij}}{\partial x_i}. \tag{4}$$

Here $W_{ij} = W(x_i - x_j)$. Similar discretizations exist for the divergence and curl operators. Given that derivatives can be approximated one can solve differential equations fairly easily. For example the conservation of mass equation for a fluid can be written as,

$$\frac{d\rho}{dt} = -\rho \nabla \cdot \vec{v}, \tag{5}$$

where v is the velocity of the fluid and the LHS is the material or total derivative of the density. The equation 5 is in a Lagrangian form, in that it represents the rate of change of density as one is moving locally with the fluid. If an SPH discretization of this equation were performed we would get,

$$\frac{d\rho_i}{dt} = -\rho_i \sum_{j \in \mathcal{N}(x)} \frac{m_j}{\rho_j} \vec{v}_{ji} \cdot \nabla_i W_{ij}, \tag{6}$$

where $\vec{v}_{ji} = \vec{v}_j - \vec{v}_i$. This equation is typical of most SPH discretizations. SPH can therefore be used to discretize any differential equation. This works particularly well for a variety of continuum mechanics problems. Consider the momentum equation for an inviscid fluid,

$$\frac{d\vec{u}}{dt} = -\frac{1}{\rho}\nabla p \tag{7}$$

A typical SPH discretization of this could be written as,

$$\frac{d\vec{u}_i}{dt} = -\sum_j m_j \left(\frac{p_j}{\rho_j^2} + \frac{p_i}{\rho_i^2}\right) \nabla W_{ij} \tag{8}$$

More details of these and various other equations can be seen in the review by Monaghan [Mon05]. It is easy to see that equations 6 and 8 are ordinary differential equations that govern the rate of change of the density and velocity of a fluid particle. In principle, one can integrate these ODEs to obtain the flow solution given a suitable initial condition and appropriate boundary conditions.

Numerical implementation

As discussed in the previous section, in an SPH scheme, the field properties are first discretized into particles carrying them. Partial differential equations are reduced to a system of coupled ordinary differential equations (ODEs) and discretized using an SPH approximation. This results in a system of ODEs for each particle. These ODEs need to be integrated in time along with suitable boundary and initial conditions in order to solve a particular problem. To summarize, a typical SPH computation proceeds as follows,

- Given an initial condition, the field variables are discretized into particles carrying the various properties.
- Depending on the scheme used to integrate the ODEs, the RHS of the ODEs needs to be computed (see equations 6 and 8). These RHS terms are called "accelerations" or "acceleration terms".
- Once the RHS is computed, the ODE can be integrated using a suitable scheme and the fluid properties are found at the next timestep.

The RHS is typically computed as follows:

- Initalize the particle accelerations (i.e. the RHS terms).
- For each particle in the flow, identify the neighbors of the particle which will influence the particle.
- For each neighbor compute the acceleration due to that particle and increment the acceleration.

Given the total accelerations, the ODEs can be readily integrated with a variety of schemes. Any general purpose abstraction of the SPH method must hence provide functionality to:

1) Easily represent the discretized properties of particles. This is easily done with `numpy` arrays representing the property values in Python.
2) Given a particle, identify the neighbors that influence the particle. This is typically called Nearest Neighbor Particle Search (NNPS) in the literature.

3) Define the interactions between the particles, i.e. an easy way to specify the inter particle accelerations. In PySPH these are called "Equations".
4) Define how the ODEs should be integrated.

Of the above, the NNPS algorithm is usually a well-known algorithm. For incompressible flows where the smoothing radius of the particles, h, is constant, a simple bin-based linked list implementation is standard. For cases where h varies, a tree-based algorithm is typically used. Users usually do not need to experiment or modify the NNPS. PySPH allows the rest of the tasks to be all implemented in pure Python.

The PySPH framework

PySPH allows a user to specify the inter-particle interactions as well as the ODE integration in pure Python with a rather simple and low-level syntax. This is described in greater detail further below. As discussed in the introduction, with older versions of PySPH as discussed in [RK10], these interactions would all need to be written in Cython. This was not very easy or convenient. It was also rather limiting.

The current version of PySPH supports the following:

- Define a complete SPH simulation entirely in Python.
- High-performance code is generated from this high-level Python code automatically and called. The performance of this code is comparable to hand-written FORTRAN solvers.
- PySPH can use OpenMP seamlessly. Users do not need to modify their code at all to use this. This works on Linux, OS X, and Windows, and produces good scale-up.
- PySPH also works with MPI and once again this is transparent to the user in that the user does not have to change code to use multiple machines. This feature requires mpi4py and Zoltan to be installed.
- PySPH provides a built-in 3D viewer for the particle data generated. The viewer requires Mayavi [RV11] To be installed.
- PySPH is also open-source and currently hosted at http://pysph.bitbucket.org

Currently, PySPH supports the simulation of compressible and incompressible fluid flows (with and without free-surfaces), simple rigid-body motion, and elastic dynamics for solids. It does not support astro-physical simulations since it lacks the tree-code needed to simulate graviational forces. This can be added but is not the current focus.

In the following subsection we provide a high-level overview of PySPH and see how it can be used by a user. Subsequent subsections discuss the design and implementation in greater detail.

High-level overview

PySPH is tested to work with Python-2.6.x to 2.7.x and also with Python 3.4/3.5. PySPH is a typical Python package and can be installed fairly easily by running:

```
$ pip install pysph
```

PySPH will require a C++ compiler. On Linux, this is trivial to get and usually pre-installed. On OS X, clang will work as will gcc (which can be easily installed using brew). On Windows the Visual C++ Compiler for Python will need to be installed. Detailed instructions for all these are available from the PySPH documentation.

If one wishes to use OpenMP,

- On Linux one needs to have libgomp installed.
- On OS X one needs to install OpenMP for clang or one could use GCC which supports OpenMP via brew.
- On Windows, just having the Visual C++ computer for Python will work.

If one wishes to use MPI for distributed computing, one must install Zoltan which is typically easy to install. PySPH provides a simple script for this. mpi4py is also needed in this case. Zoltan is used for load-balancing and distributing the particles efficiently on distributed machines. Unfortunately, MPI is not tested on Windows by us currently. PySPH also provides an optional 3D viewer and this depends on Mayavi.

In summary, PySPH is easy to install if one has a C++ compiler installed. MPI support is a little involved due to the requirement to install Zoltan.

Once PySPH is installed an executable called `pysph` is available. This is a convenient entry point for various tasks. Running `pysph -h` will provide a listing of these possible tasks. For example, the test suite can be run using:

```
$ pysph test
```

This uses nose internally and can be passed any arguments that `nosetests` accepts.

PySPH installs about 30 useful examples along with the sources and any of these examples can be readily run. For example:

```
$ pysph run
1. cavity
   Lid driven cavity using the Transport Velocity
   formulation. (10 minutes)
[...]
Enter example number you wish to run:
```

Provides a listing of the examples available and prompts for a particular one. Each example also provides a convenient (but rough) time estimate for the example to run to completion in serial. If the name of the example is known, one may directly specify it as:

```
$ pysph run elliptical_drop
```

The examples will accept a large number of command line arguments. To find these one can run:

```
$ pysph run elliptical_drop -h
```

`pysph run` will execute the standard example. Note that internally this is somewhat equivalent to running:

```
$ python -m pysph.examples.elliptical_drop
```

The example may therefore be imported in Python and also extended by users. This is by design.

When the example is run using `pysph run`, the example documentation is first printed and then the example is run. The example will typically dump the output of the computations to a directory called `example_name_output`, in the above case this would be `elliptical_drop_output`. This output can be viewed using the Mayavi viewer. This can be done using:

```
$ pysph view elliptical_drop_output
```

This will start up the viewer with the saved files dumped in the directory. Figure 1 shows the viewer in action. The viewer

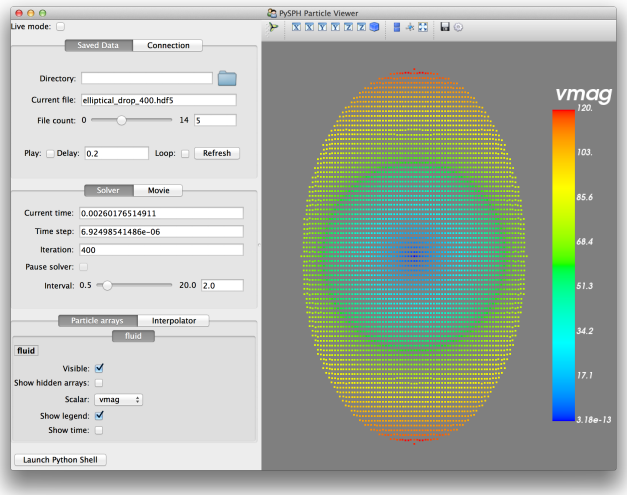

Fig. 1: *The viewer provides a convenient interface to view data dumped by simulations.*

provides a very convenient interface to view the data. On the right side, one has a standard Mayavi widget which also features a Mayavi icon on the toolbar. Clicking this will open the Mayavi UI with which one can easily change the visualization. On the left pane there are three sub panels. On the top, one can see a slider for the file count. This can be used to move through the simulation in time. This can be also animated by checking the "Play" checkbox which will iterate over the files. The "Directory" button allows one to view data from a different output directory. Hitting the refresh button will rescan the directory to check for any new files. This makes it convenient to visualize the results from a running simulation. The "Connection" tab can be used when the visualization is in "Live mode" when it can connect to a running simulation and view the data live. While this is very useful in principle, it is seldom used in practice as it is a lot more efficient to just view the dumped files and use the "Refresh" button is convenient. Regardless, it does show another feature of PySPH in that one can actually pause a running simulation and query it if needed. Below this pane is a "Solver" pane which shows the various solver parameters of interest. The "Movie" tab allows a user to dump screenshots and easily produce a movie if needed. At the bottom of the interface are two panels called "Particle arrays" and "Interpolator". The particle arrays lists all the particles and different scalar properties associated with the SPH simulation. Selecting different scalars will display those scalars. The interpolator tab allows a user to specify a rectilinear region on which the particle properties may be interpolated and visualized -- for example if one wishes to see a contour of velocity magnitudes this would be useful. Right at the bottom is a button to launch a Python shell. This can be used for advanced scripting and is seldom used by beginners. This entire viewer is written using about 1024 lines of code and ships with PySPH.

PySPH output can be dumped either in the form of `.npz` files (which are generated by NumPy) or HDF5 files if h5py is installed. These files can be viewed using other tools or with Python scripts if desired. The HDF5 in particular can be viewed more easily. In addition, the `pysph dump_vtk` command can be used to dump VTK output files that can be used to visualize the output using any tool that supports VTK files like ParaView etc. This can use either Mayavi or can use pyvisfile which has no dependency on VTK. Finally, the saved data files can be loaded in Python very easily, for example:

```
from pysph.solver.utils import load
data = load('elliptical_drop_100.hdf5')
# if one has only npz files the syntax is the same.
data = load('elliptical_drop_100.npz')
```

This provides a dictionary from which one can obtain the particle arrays and solver data:

```
particle_arrays = data['arrays']
solver_data = data['solver_data']
fluid = particle_arrays['fluid']
p = fluid.p
```

where `particle_arrays` is a dictionary of all the PySPH particle arrays. `solver_data` is another dictionary with solver properties and `p` is a NumPy array of the pressure of each particle. Particle arrays are described in greater detail in the following sections. Our intention here is to show that the dumped data can be very easily loaded into Python if desired.

As discussed earlier, PySPH supports OpenMP and MPI. To use multiple cores on a computer one can simply run an example or script as:

```
$ pysph run elliptical_drop --openmp
```

This will use OpenMP transparently and should work for all the PySPH examples. PySPH will honor the `OMP_NUM_THREADS` environment variable to pick the number of threads. If PySPH is installed with MPI support through Zoltan, then one may run for example:

```
$ mpirun -np 4 pysph run dam_break_3d
```

This will run the `dam_break_3d` example with 4 processors. The amount of scale-up depends on the size of the problem and the network. OpenMP will scale fairly well for moderately sized problems. Note that for a general PySPH script written by the user, the command to run would simply be:

```
$ mpirun -np 4 python my_script.py
```

Similarly when using OpenMP:

```
$ python my_example.py --openmp
```

This provides a very high-level introduction to PySPH in general. The next section discusses some essential software engineering used in the development of PySPH. This is followed by details on the underlying design of PySPH.

Essential software engineering

PySPH follows several of the standard software development practices that most modern open source implementations follow. For example:

- Our sources are hosted on bitbucket (http://pysph.bitbucket.org). We are thinking of shifting to GitHub because GitHub has much better integration with continuous integration services and this is a rather frustrating pain point with bitbucket.
- We use pull requests to review all new features and bug fixes. At this point there is only a single reviewer (the author) but this should hopefully increase over time.
- PySPH has a reasonable set of unit tests and functional tests. Each time a bug is found, a test case is first created

(when possible or reasonable), and then fixed. nose is used for discovering and executing tests. One of our functional tests runs one time step of every single example that ships with PySPH. tox based tests are also supported. This makes it easy to test on Python 2.6, 2.7 and 3.x.

- We use continuous integration services from http://shippable.com for Linux, http://appveyor.com for Windows and http://codeship.com for faster Linux builds.
- Our documentation is generated using Sphinx and hosted online on http://pysph.readthedocs.io.
- Releases are pushed to the Python Package Index (PyPI).
- The pysph-users mailing list is also available where users can post their questions. Unfortunately, the response time is currently slow as the author does not have the time for this but we are hoping this will improve as more graduate students start getting involved with PySPH.

These greatly improve the quality, reliability and usability of the software and also encourage open collaboration.

Design overview

In the previous sections a high-level description of the project was provided. This section provides more design details of how PySPH works internally. The general approach used in PySPH is as follows:

1) Create particles: discretize the initial materials into particles with suitable properties.
2) Choose an appropriate kernel for the SPH approximation.
3) Create equations: write out the equations that specify the inter-particle interactions.
4) Setup the integrator and specify the integration steps, for example one could use an Euler scheme or a predictor-corrector scheme and each of these involve slightly different integration steps. These need to be specified explicitly.

PySPH allows a user to do all of these from pure Python.

1) In PySPH, particles of a particular kind are managed by a `ParticleArray` instance. A particle array is assigned a unique name and manages a collection of properties. Each property is internally represented as a contiguous block of memory. All properties have the same number of elements. A particle array may also have any number of "constants" associated with it. Each constant can be a scalar or an array but its size is independent of the number of particles.
2) The kernels are implemented in pure Python and a default collection of kernels is available in `pysph.base.kernels`. A new kernel class would implement the following methods, note that the default arguments have no meaning except that they help the code generator use the correct types:

```
class MyKernel(object):
    def __init__(self, dim):
        # ...
    def kernel(self, xij=[0., 0, 0], rij=1.0,
               h=1.0):
        # ...
    def gradient(self, xij=[0., 0, 0], rij=1.0,
                 h=1.0, grad=[0, 0, 0]):
        # ...
```

3) In PySPH, the equations can also be created in pure Python and this is discussed in detail in the following.
4) The integrators are split into two parts, an integrator and an integrator step. This is also written in pure Python and discussed with an example further below.

A typical example is considered first to illustrate the design. Consider the example `pysph/pysph/examples/elliptical_drop.py`. When installed, this may be imported as `import pysph.examples.elliptical_drop`. This example simulates the evolution of a fluid drop that is initially circular and imposed an initial velocity field of the form $\vec{V} = -100x\hat{i} + 100y\hat{j}$. This problem is a simple benchmark problem that was first solved in the context of SPH by [Mon94]. The key parts of the example are shown below:

```
from numpy import array, ones_like, mgrid, sqrt

# PySPH base and carray imports
from pysph.base.utils import get_particle_array
from pysph.base.kernels import Gaussian

# PySPH solver and integrator
from pysph.solver.application import Application
from pysph.sph.integrator import EPECIntegrator
from pysph.sph.scheme import WCSPHScheme

class EllipticalDrop(Application):
    def initialize(self):
        # ...
    def create_particles(self):
        # ...
    def create_scheme(self):
        # ...
    def post_process(self, info_file_or_dir):
        # ...

if __name__ == '__main__':
    app = EllipticalDrop()
    app.run()
    app.post_process(app.info_filename)
```

This illustrative example deliberately excludes several details to focus on the general structure and API. There are a few common imports at the top starting with NumPy specific imports first. The next imports are PySPH specific:

- `get_particle_array` is a convenient function that helps create a `ParticleArray` instance.
- The `Gaussian` kernel is used for the SPH simulation.
- The `Application` class is subclassed to create the new example.
- The `WCSPHScheme` encapsulates a particular scheme, in this case this class abstracts out the requirements for a weakly-compressible scheme applied to incompressible flows. Internally the WCSPH scheme is responsible to setup the equations and the integrator. By abstracting this into a scheme it becomes easy to reuse this instead of spelling out the equations for each example.

The typical entry point for a user is to subclass `Application` to solve their particular problem. The methods listed above are:

- `initialize`, this is automatically called by `Application.__init__` and is typically not used but sometimes useful when one wishes to have some common attributes setup.

- `create_particles` generates the initial particle distribution and returns a sequence of `ParticleArray` instances.
- `create_scheme` creates the particular scheme. A `SchemeChooser` is also available which can be given multiple schemes and allows the user to switch between them via command line arguments.
- the `post_process` method is run in the end to compute any useful quantities that may be used to check the accuracy of the simulation or facilitate comparisons between different schemes.

The `if __name__` block is listed to just illustrate how this application can be used. When `run` is called, the command line arguments are parsed, the various objects involved are suitably configured and the simulation executed. At the end, the `post_process` method is called. This also shows that a user could potentially rewrite the post processing code and simply rerun that part instead of re-running the simulation (which can sometimes run for days).

We next look inside the `create_particles` and `create_scheme` methods:

```
1  def create_particles(self):
2      x, y = mgrid[-1.:1.05:dx,-1.:1.05:dx]
3      x, y = x.ravel(), y.ravel()
4      m = ones_like(x)*dx*dx
5      h = ones_like(x)*hdx*dx
6      # ...
7      u = -100*x
8      v = 100*y
9
10     # remove particles outside the circle
11     indices = []
12     for i in range(len(x)):
13         dist = sqrt(x[i]*x[i] + y[i]*y[i])
14         if dist - 1 > 1e-10:
15             indices.append(i)
16
17     pa = get_particle_array(
18         x=x, y=y, m=m, rho=rho, h=h, p=p,
19         u=u, v=v, cs=cs, name='fluid')
20     pa.remove_particles(indices)
21     self.scheme.setup_properties([pa])
22     return [pa]
23
24 def create_scheme(self):
25     s = WCSPHScheme(
26         ['fluid'], [], dim=2, rho0=self.ro, c0=co,
27         h0=self.dx*self.hdx, hdx=self.hdx,
28         gamma=7.0, alpha=0.1, beta=0.0
29     )
30     kernel = Gaussian(dim=2)
31     dt = 5e-6; tf = 0.0076
32     s.configure_solver(
33         kernel=kernel,
34         integrator_cls=EPECIntegrator,
35         dt=dt, tf=tf, adaptive_timestep=True,
36         cfl=0.3, n_damp=50,
37     )
38     return s
```

The `create_particles` method above is straightforward. NumPy arrays are created that set the position, mass, smoothing radius h, the velocity etc. The arrays are all one dimensional. The indices that are outside the circle are identified between lines 11 and 14 and these are removed in line 20. This could have also been done with pure NumPy indexing. In Line 17 the particle array instance is created and is called `'fluid'`. Line 22 delegates to the `scheme` to setup any additional properties for the particle array and finally a list of particle arrays is returned.

The `create_scheme` method is fairly simple. A `WCSPHScheme` is instantiated and passed arguments as defaults. The kernel is created and this is all passed to a scheme method called `configure_solver`, this also specifies the integrator to use, the timestep to use, the time for which the simulation is to be run etc. To someone who is familiar with SPH, these are fairly obvious parameters. The scheme may also allow a user to set these parameters via command line arguments. This can be found by simply running:

```
$ pysph run elliptical_drop -h
```

The `post_process` method is also fairly straightforward and is entirely optional. With just this code, one may run the example. As soon as this is done, PySPH will generate high-performance code, compile it, and use that code to run the example.

The scheme in this case is really doing a lot of work because it encapsulates the creation of the equations and the integrators. In order to understand this better, we look at a lower-level implementation of the same example. This example also ships with PySPH and is called `elliptical_drop_no_scheme.py`. Unsurprisingly, this example can be run as:

```
$ pysph run elliptical_drop_no_scheme
```

This implementation does not use a scheme but instead creates the equations and the `Solver` instance directly. The example differs from the `elliptical_drop` in that there is no `create_scheme` method but instead there are two additional methods: - `create_equations` which explicitly creates the equations. - `create_solver` which sets up the solver, stepper and integrators. The `create_particles` and `post_process` etc. are all identical. The code is listed below:

```
def create_equations(self):
    equations = [
        Group(equations=[
          TaitEOS(
              dest='fluid', sources=None,
              rho0=self.ro, c0=self.co, gamma=7.0),
        ], real=False),

        Group(equations=[
            ContinuityEquation(
                dest='fluid', sources=['fluid',]),

            MomentumEquation(
                dest='fluid', sources=['fluid'],
                alpha=self.alpha, beta=0.0,
                c0=self.co),

            XSPHCorrection(dest='fluid',
                           sources=['fluid']),
        ]),
    ]
    return equations
```

As can be seen, the equations are simply instantiated. We look closer at equations further below but at this stage it can be seen that:

- Each equation has a destination `dest` and a list of sources. A destination is a particle on which the acceleration is to be computed a source is one that influences the particle. In this problem there is only one destination and source, `"fluid"`. Note that the names of the arrays are used here to determine the appropriate particle array.
- The `TaitEOS` is an equation of state, i.e. it does not depend on any neighbors and is simply an equation of

the form $p = (\rho - \rho_0)c^2$ or something along those lines. This does not require any "sources".

- Equations can be "grouped" using a `Group`. Each time the acceleration is computed, all equations in a group are evaluated for all the particles before the next group is considered. This is important in the above case as an equation of state is needed to compute the pressure. The pressure must be found for all particles before the other accelerations are evaluated.
- The other equations describe the physics of the problem, namely, continuity and momentum. The `XSPHCorrection` is an SPH-specific correction (see [Mon05]).
- The group containing `TaitEOS` has an additional argument `real=False` this is only used when the example is run via MPI and specifies that the equation of state be computed for all particles local and remote.

```
def create_solver(self):
    kernel = Gaussian(dim=2)

    integrator = EPECIntegrator(fluid=WCSPHStep())

    dt = 5e-6; tf = 0.0076
    solver = Solver(
        kernel=kernel, dim=2, integrator=integrator,
        dt=dt, tf=tf, adaptive_timestep=True,
        cfl=0.3, n_damp=50,
        output_at_times=[0.0008, 0.0038])

    return solver
```

The `create_solver` method simply instantiates a `EPECIntegrator` and asks that the fluid particles be stepped with the `WCSPHStep` stepper. A solver is then constructed which combines the kernel, integrator, and any integration parameters. The scheme automatically creates the equations and solver. Specifying equations directly can be error prone and schemes make this task a lot easier. Schemes also support command line arguments which the direct example would require additional code for.

The only thing that remains is to see how the equations and steppers are actually implemented. Let us consider the continuity equation (6) and see how the `ContinuityEquation` class is implemented.

```
class ContinuityEquation(Equation):
    def initialize(self, d_idx, d_arho):
        d_arho[d_idx] = 0.0

    def loop(self, d_idx, d_arho, s_idx,
             s_m, DWIJ, VIJ):
        vijdotdwij = DWIJ[0]*VIJ[0] + \
            DWIJ[1]*VIJ[1] + DWIJ[2]*VIJ[2]
        d_arho[d_idx] += s_m[s_idx]*vijdotdwij
```

In this class there are two methods:

- `initialize`: this is called first for every destination particle with index `d_idx`.
- `loop`: this is called for every destination source pair. Thus, internally all the nearest neighbors of the destination particle are identified and looped over.

There are some simple conventions followed with the variable names.

- `d_*` indicates a destination array. The name that follows `d_` is the same as the property name of the array.
- `s_*` indicates a source array.
- `d_idx` is a destination index and `s_idx` the source index.
- A method can take any arguments in arbitrary order and these are automatically passed in the right order.

Clearly this seems rather low-level, however, it is simple to write and maps almost exactly with the actual SPH discretized equation (see equation 6).

The integrator and integrator stepper code is similarly quite simple and low level. It is written entirely in pure Python. More details are available in the online PySPH design overview document.

This approach allows a user to specify new equations and integration schemes very easily and use them to perform SPH simulations. The `Application` class also has several other convenient methods that can be overridden by the user to perform a variety of tasks. For example:

- `add_user_options` can be overridden to add any user-defined command line arguments. The argument parsing is done using argparse. Once processed, the options are available in `self.options`.
- `consume_user_options` is used to use any of the parsed options. This is called after the command line arguments are parsed but before the `create_particles` etc.
- `create_domain` can be used to create a periodic domain.
- `configure_scheme` can be used to configure a created scheme based on command line arguments. This is also useful in conjunction with user-defined command line arguments.
- `pre_step`, `post_step`, `post_stage` are convenient methods which will be called before each timestep, after each timestep and after each integration stage if these are defined. These are convenient for a variety of user defined actions including debugging, adaptive refinement, checking for errors etc.

Together, these features are extremely powerful and allow a user a great deal of flexibility.

High performance

While PySPH allows a user to write the code in pure Python, internally, high-performance Cython code is generated, compiled, and used to extract as much performance as possible. This is done using Mako templates. A general Mako template is written to compute the accelerations, this is in `pysph/sph/acceleration_eval_cython.mako`. The main module is `pysph.sph.acceleration_eval` which is implemented in pure Python. A helper class `pysph.sph.acceleration_eval_cython_helper` uses all the high-level information from the user code and provides several methods that are called from the mako template.

The user Python code is already implemented in a low-level allowing us to directly inject the sources into the Cython code. The `pysph.base.cython_generator` module helps with the generation of Cython code from Python code. The `pysph.base.ext_module` takes the generated Cython and compiles this. The extension modules are stored in `~/.pysph/source` in a Python version and architecture specific directory. The md5sum of the Cython code is checked and

if an extension for that md5sum exists the code is not recompiled. Care is taken to look for changes in dependencies of this generated source.

As a result of this, the code performs almost as well as a hand-written FOTRAN code. We have compared running both 2D and 3D problems with the SPHysics serial code. In 2D our code is about 1.5 times slower. This is in part because by default the PySPH implementation is 3D. In 3D, PySPH is about 1.3 times slower. SPHysics symmetrizes the inter-particle computations, i.e. while computing the interaction of a source on a destination, they also compute the opposite force and store it. This appears to provide additional performance gains. Regardless, it is clear that PySPH is comparable in performance with SPHysics. However, PySPH is a lot easier to use and much easier to extend.

PySPH also displays good scale-up with OpenMP. Consider the cube example which considers a cube of a user-defined number of particles (100000 by default), and takes 5 timesteps. One can run `pysph run cube --disable-output` and compare the time taken to run this with `--openmp`. On a quad-core Macbook Pro this produces a speedup of about 4.16. This shows that the scale up is excellent. Good scale up has been observed in the distributed case but is not discussed here.

Reproducibility

The object-oriented API of PySPH makes it easy to extend and use. The design allows for a large amount of code reuse.

We have found that it is extremely important to treat our examples to be as important as the source itself and that these should be shipped with the installation as part of the sources. This forces us to design our examples to be reusable. This is extremely important as:

- it forces a clean API for an end-user. This drives us to minimize repetitive code, and simplify the API.
- the examples are all reusable. If a user wishes to try a new scheme they need to just focus on the new scheme.
- it makes the library easier to use.

While post-processing results, the post-processed data is dumped into a separate file. This makes it trivial to compare the output of different schemes. Some simple tools in `pysph.tools.automation` are provided which make it easy to use PySPH in an automation framework.

Recently, we have used these features to make an entire publication [RP16] completely reproducible. Every figure produced in the paper (a total of 23 in number) is produced with a single driver script making it possible to rerun all the simulations with a single command. This will be described in a future publication. However, it is important to note that PySPH allows for reproducible computation with the SPH method.

Plans

In the future, the plan is to develop the following features:

- A GPU backend which should allow effective utilization of GPUs with minimal changes to the API.
- Cleanup and potential generalization of the parallel code.
- Implement more SPH schemes.
- Better support for variable h.
- Cleanup of many of the current equations implemented.
- Support for implicit SPH schemes and other related particle methods.
- Advanced algorithms for adaptive resolution.

Conclusions

In this paper a broad overview of the SPH method was provided. The background and context of the PySPH package was discussed. A very high-level description of the PySPH features were provided followed by an overview of the design. From the description it can be seen that PySPH provides a powerful API and allows users to focus on the specifics of the SPH scheme which they are interested in. By abstracting out the high-performance aspects even inexperienced programmers can use the high-level API and produce useful simulations that run quickly and scale well with multiple cores and processors. The paper also discusses how PySPH facilitates reproducible research.

Acknowledgments

I would like to thank Kunal Puri, Chandrashekhar Kaushik, Pankaj Pandey and the other PySPH developers and contributors for their work on PySPH. I thank the department of aerospace engineering, IIT Bombay for their continued support, excellent academic environment and academic freedom that they have extended to me over the years.

References

[BBC+11] S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D.S. Seljebotn, and K. Smith. Cython: The best of both worlds. *Computing in Science Engineering*, 13(2):31 –39, March-April 2011. URL: http://www.cython.org, doi:10.1109/MCSE.2010.118.

[CPR12] J.M. Cherfils, G. Pinon, and E. Rivoalen. JOSEPHINE: A parallel {SPH} code for free-surface flows. *Computer Physics Communications*, 183(7):1468 – 1480, 2012. doi:http://dx.doi.org/10.1016/j.cpc.2012.02.007.

[deva] DualSPHysics developers. Dualsphysics home page. URL: http://www.dual.sphysics.org/.

[devb] SPHysics developers. Sphysics home page. URL: https://wiki.manchester.ac.uk/sphysics/index.php/SPHYSICS_Home_Page.

[GM77] R. A. Gingold and J. J. Monaghan. Smoothed particle hydrodynamics: Theory and application to non-spherical stars. *Monthly Notices of the Royal Astronomical Society*, 181:375–389, 1977.

[GMS01] J.P. Gray, J. J. Monaghan, and R.P. Swift. SPH elastic dynamics. *Computer Methods in Applied Mechanics and Engineering*, 190:6641–6662, 2001.

[Guo14] Philip Guo. Python is now the most popular introductory teaching language at top u.s. universit ies, 2014. http://cacm.acm.org/blogs/blog-cacm/176450-python-is-now-the-most-popular-introductory-teaching-language-at-top-us-universities/fulltext.

[Luc77] L. B. Lucy. A numerical approach to testing the fission hypothesis. *The Astronomical Journal*, 82(12):1013–1024, 1977.

[Mon94] J. J. Monaghan. Simulating free surface flows with SPH. *Journal of Computational Physics*, 110:399–406, 1994.

[Mon05] J. J. Monaghan. Smoothed Particle Hydrodynamics. *Reports on Progress in Physics*, 68:1703–1759, 2005.

[Per15] Jefferey M. Perkel. Pickup Python. *Nature*, 518:125–126, February 2015.

[RK10] Prabhu Ramachandran and Chandrashekhar Kaushik. PySPH: A python framework for smoothed particle hydrodynamics. In Stéfan van der Walt and Jarrod Millman, editors, *Proceedings of the 9th Python in Science Conference*, pages 16 – 20, 2010.

[RP16] Prabhu Ramachandran and Kunal Puri. Entropically damped artificial compressibility for SPH. *Under review*, 2016.

[RV11] Prabhu Ramachandran and Gaël Varoquaux. Mayavi: 3d visualization of scientific data. *Computing in Science and Engineering*, 13(2):40–51, 2011.

[Spr05] Volker Springel. The cosmological simulation code gadget-2. *Monthly Notices of the Royal Astronomical Society*, 364:1105–1134, 2005.

An Ecological Approach to Software Supply Chain Risk Management

Sebastian Benthall[‡§*], Travis Pinney[‡¶], JC Herz[**‡], Kit Plummer[‖‡]

https://youtu.be/6UnuPhTPdnM

Abstract—We approach the problem of software assurance in a novel way inspired by an analytic framework used in natural hazard risk mitigation. Existing approaches to software assurance focus on evaluating individual software projects in isolation. We demonstrate a technique that evaluates an entire ecosystem of software projects, taking into account the dependencey structure between packages. Our model analytically separates vulnerability and exposure as elements of software risk, then makes minimal assumptions about the propagation of these values through a software supply chain. Combined with data collected from package management systems, our model indicates "hot spots" in the ecosystem of higher expected risk. We demonstrate this model using data collected from the Python Package Index (PyPI). Our results suggest that Zope and Plone related projects carry the highest risk of all PyPI packages because they are widely used and their core libraries are no longer maintained.

Index Terms—risk management, software dependencies, complex networks, software vulnerabilities, software security

Introduction

Systems that depend on complex software are open to many kinds of risk. One typical approach to software security that mitigates this risk is static analysis. We are developing novel methods to manage software risk through supply chain intelligence, with a focus on open source software ecosystems.

The Heartbleed bug in OpenSSL is an example of community failure and of how vulnerabilities in open source software can be a major security risk. [Wheeler2014] The recent failure of React, Babel, and many other npm packages due to the removal of one small dependency, `left-pad`, shows how dependencies can be a risk factor for production software. [Haney2016] These high profile examples, though quite different from each other, illustrate how software risk traverses the supply chain. As dependencies become more numerous and interlinked, the complexity of the system increases, as does the scope of risk management. Open source software projects make their source code and developer activity data openly available for analysis. This data can be used to mitigate software risk in ways that have not been explored.

* *Corresponding author: sb@ischool.berkeley.edu*
‡ *Ion Channel, ionchannel.io*
§ *UC Berkeley School of Information*
¶ *travis.pinney@ionchannel.io*
** *jc.herz@ionchannel.io*
‖ *kit.plummer@ionchannel.io*

With a small number of analytic assumptions about the propagation of vulnerability and exposure through the software dependency network, we have developed a model of ecosystem risk that predicts "hot spots" in need of more investment. In this paper, we demonstrate this model using real software dependency data extracted from PyPI using Ion Channel [IonChannel].

Prior work

[Verdon2004] outline the diversity of methods used for risk analysis in software design. Their emphasis is on architecture-level analysis and its iterative role in software development. Security is achieved through managing information flows through architecturally distinct tiers of trust. They argue for a team-based approach with diverse knowledge and experience because "risk analysis is not a science". Contrary to this, our work develops a scientific theory of risk analysis, building on work from computer science and other fields.

There is a long history of security achieved through static analysis of source code. [Wagner2000] points out that the dependency of modern Internet systems on legacy code and the sheer complexity of source code involved makes manual source code level auditing very difficult. While some complex projects are audited by large and dedicated communities, not all software systems are so gifted in human resources. Therefore, static analysis tools based on firm mathematical foundations are significant for providing computer security at scale.

[Wheeler2015] develops a risk index for determining which open source software projects need security investments. This work is part of the Linux Foundation (LF) Core Infrastructure Initiative (CII) and published by the Institute for Defense Analysis. This metric is based on their expertise in software development analytics and an extensive literature review of scholarly and commercial work on the subject. They then apply this metric to Debian packages and have successfully identified projects needing investment. This work is available on-line as the CII Census project [CensusProject].

While software security studies general focus on the possibility of technical failure of software systems, open source software exposes an additional risk of community failure. Development of a software project may cease before it reaches a state of usability and maturity. [Schweik2012] is a comprehensive study of the success and failure of open source projects based on large-scale analysis of SourceForge data, as well as survey and interview data. They define a successful project as one that performs a

useful function and has had at least three releases. They identify several key predictive factors to project success, including data that indicates usefulness (such as number of downloads), number of hours contributed to the project, and communicativeness of the project leader.

These precedents focus on individual software projects and their susceptiblity to technical and community failure. [Nagappan2005] and [Nagappan2007] look at dependency relationships between packages and specifically relative code churn (changes in lines of code) between dependent packages as a cause of system defects in Windows Server 2003.

We build on these approaches by considering security as a function of the entire software supply chain. This supply chain resembles a complex ecosystem more than a simple 'chain' or stack. We draw inspiration from a risk management strategy approaches used in another kind of complex system, namely disaster risk reduction and climate change adaptation research developed by Cardona [Cardona2012] and widely used by the World Bank's Global Facility for Disaster Risk Reduction among others [Yamin2013].

This framework evaluates the expected cost of low-probability events by distinguishing three factors of risk: hazards, exposure, and vulnerabilities. *Hazards* are potentially damaging factors from the environment. *Exposure* refers to the inventory of elements in places where hazards occur. *Vulnerabilities* are defined as the propensity of exposed elements to suffer adverse effects when impacted by a hazard. Expected risk is then straightforwardly calculated using the formula:

$$risk = hazard * vulnerability * exposure$$

We adapting this framework to cybersecurity in the software ecosystem. There are significant differences between modeling risk from natural hazards and modeling cybersecurity risk. Most notably, cybersecurity threats can be deliberately adversarial, detecting and exploiting specific weaknesses rather than presenting a general hazard. In this work we focus on the interplay between exposure and vulnerability in the software ecosystem and abstract away the specificity of a threat model. We see an analytic treatment of that interplay as a valuable step in tractable security analysis of the software supply chain.

Modeling Ecological Risk in Software

Software dependency and project risk

Some previous studies of software risk [Wheeler2015] have suffered from the ambiguity of how 'risk' is used in a software development context. Security research often contextualizes problems within a specific threat model. But for some applications, such as identifying software projects in need of additional investment in order to mitigate risk from generalized and potentially unknown threats, this kind of threat modeling is inappropriate. A general concern with supply chain security motivates a different approach.

If we break down the sources of risk and how these affect the need for security investments analytically, we can distinguish between several different factors:

- Vulnerability. A software project's vulnerability is its intrinsic susceptibility to attack. Common Vulnerability and Exposure (CVE) records are good examples of specific software vulnerabilities. But software's vulnerability can also be predicted from a general property, such as the language it's written in. (Some languages, such as C++, are harder to write in securely and therefore generally more vulnerable [Wheeler2015])
- Exposure. A software project's exposure is its extrinsic availability to attack. A direct network connection is a source of exposure.

Vulnerability and exposure are distinct elements of a software project's risk. Analyzing them separately and then combining them in a principled way gives us a better understanding of a project's risk.

Dependencies complicate the way we think about vulnerability and exposure. A software project doesn't just include the code in its own repository; it also includes the code of all of its dependencies, often tied to a specific version. Furthermore, a package does not need to be installed directly to be exposed--it can be installed as a dependency of another project, or as a transitive dependency. Based on these observations, we can articulate two heuristics for use of dependency topology in assessing project risk:

- If A depends on B, then a vulnerability in B implies a corresponding vulnerability in A.
- If A depends on B, then an exposure to A implies an exposure to B.

For example, if a web application (A) uses a generic web application framework (B), and that web application is installed and recieving web traffic, then there is an instance of the web framework installed and recieving web traffic. The framework is exposed through the web application. If there is a vulnerability in the web application framework (such as a susceptibility to SQL injection attacks), then the web application will inherit that vulnerability. There are exceptions to these rules. Developers of the web application (A) might recognize the vulnerability to SQL injection and fix the problem without pushing the change upstream (to B). Nevertheless, this is a principled analytic way of relating vulnerability, exposure, and software dependency that can be implemented as a heuristic and tested as a hypothesis.

The risk analysis framework described above is very general. Due to this generality, the framework suffers from the ambiguity of its terms. Depending on the application of this framework, "vulnerability" refers to literal software vulnerabilities such as would be reported in a CVE. When we analyze the software ecosystem as a supply chain, we are often concerned about higher level properties that serve as general proxies for whole classes of error or failure.

Robustness and resilience

We find the distinction between system *robustness* and system *resilience* helpful. We define the *robustness* of a system as its invulnerability to threats and hazards, as a function of its current state. We define the *resilience* of a system as its capacity to recover quickly from injury or failure. A mature, well-tested system will be robust. A system with an active community ready to respond to the discovered of a new exploit will be resilient.

A system can be robust, or resilient, both, or neither. Robustness and resilience can be in tension with each other. For example, the more churn a software project is, measured as a function of the activity of the community and frequency of new commits, the more likely that it will be resilient, responding to new threat information. But it is also likely to be less robust, as new code might introduce new software flaws. [Nagappan2005] and

[Nagappan2007] find that relative code churn between dependent packages is a significant predictor of system defects.

We refer to a system that is not robust as *fragile*, and a system that is not resilient as *brittle*. Fragility and brittleness are two distinct and general ways in which a component of a software ecosystem might be vulnerable.

Computing fragility and exposure

Our risk analysis framework defines exposure and vulnerability as abstract components of risk that can be defined depending on the hazards and threats under consideration. In the example of this study, we will define these variables with an interest in the general prediction of robustness in widely used software. This sort of analysis would be useful in determining which software packages are in need of further investment in order to reduce risk globally.

In the following analysis, we will define *exposure* to be the number of times a package has been downloaded. We assume for the sake of this analysis that more widely downloaded software is more widely used and exposed to threats. This metadata is provided by PyPI for each package directly.

We will define vulnerability specifically in terms of software fragility, and make the assumption that frequently released software is less fragile. While it is true that sometimes a new software release can introduce new flaws into software, we assume that, on average, more releases mean a more active community, more robust development processes, and greater maturity in the project lifecycle. Specifically for the purpose of this study we will define

$$fragility(p) = \frac{1}{number\_of\_releases(p)}$$

In future work, we will revise and validate these metrics.

Implementation of risk computation

The risk analysis framework presented in the above section *Software dependency and project risk* is designed to be widely applicable, factoring risk into abstract *exposure* and *vulnerability* factors and then making minimal assumptions about how these factors propagate through the dependency graph.

In practice, the application of this framework will depend on the selection of package metadata used to measure exposure and vulnerability. Below is a Python implementation of efficient risk computation using a directed graph representation of package dependencies and NetworkX. [Hagberg2008] It imports data as a graph, where packages are nodes, directed edges indicate package dependencies, and relevant metadata are precomputed properties of the nodes. In this code, we use a precomputed 'fragility' metric as the vulnerability variable, and the number of unique downloads of each package as the exposure variable. Running this code imports the data from a Graph Exchange XML Format (GEXF) file, computes the ecosystem risk of each package, and exports the data to a different file.

```
import networkx as nx

G = nx.read_gexf('pkg.gexf')

# select proxy empirical variables for
# vulnerability and exposure

vulnerability_metric = 'fragility'
exposure_metric = 'downloads'

# efficiently compute ecosystem vulnerability
# and assign as attribute

ecosystem_vulnerability = {}

for i in nx.topological_sort(G,reverse=True):

    ecosystem_vulnerability[i] =
            G.node[i][vulnerability_metric]
            + sum([ecosystem_vulnerability[j]
                   for j in G.neighbors(i)])

nx.set_node_attributes(G,
                       'ecosystem_vulnerability',
                       ecosystem_vulnerability)

# efficiently compute ecosystem exposure
# and assign as attribute

ecosystem_exposure = {}

for i in nx.topological_sort(G):

     ecosystem_exposure[i] =
            G.node[i][exposure_metric]
            + sum([ecosystem_exposure[j]
                   for j in G.predecessors(i)])

nx.set_node_attributes(G,
                       'ecosystem_exposure',
                       ecosystem_exposure)

# efficiently compute ecosystem risk
# and assign as attribute

ecosystem_risk= {}

for i in nx.topological_sort(G):
    ecosystem_risk[i] =
            G.node[i]['ecosystem_vulnerability']
            * G.node[i]['ecosystem_exposure']

nx.write_gexf(G,'pkg-with-risk.gexf')
```

A significant problem with this implementation of risk calculation is that if node A is accessible to node B through multiple distinct paths, then the vulnerability (or exposure) of B will be counted towards A's ecosystem vulnerability (or exposure) once for each path. A superior version of this algorithm would ensure that each node was only counted once in ecosystem measurements. The version of the algorithm presented above uses a heuristic measure for performance reasons.

Removing cycles

The above algorithm has one very important limitation: it assumes that there are no cycles in the dependency graph. This property is necessary for the nodes to have a well-defined topological order. However, Python package dependencies do indeed include many cycles. An amusing example are the packages `chicken` and `egg`. We can adapt any directed cyclic graph into a directed acyclic graph simply by removing one edge from every cycle.

```
def remove_cycles(G):
    cycles = nx.simple_cycles(G)

    for c in cycles:
        try:
            if len(c) == 1:
                G.remove_edge(c[0],c[0])
            else:
                G.remove_edge(c[0],c[1])
        except:
            pass
```

One way to improve this algorithm would be to remove as few edges as possible in order to eliminate all cycles. Another way to improve this algorithm would be to adapt the heuristic assumptions that motivate this framework to make reasonable allowances for cycle dependencies. It is unknown how these changes will effect the results. We leave the elaboration of this algorithm for future work.

Data collection and publication

Data for this analysis comes from two sources. For package and release metadata, we used data requested from PyPI, the Python Package Index. This includes the publication date and number of unique downloads for each software release. We also downloaded each Python release and inspected it for the presence of a `setup.py` file. We then extracted package dependency information from `setup.py` through its `install_requires` field. This data is available in `.gexf` format [Benthall2016].

Python dependencies are determined through execution of Python install scripts. Therefore, our method of discovering package dependencies via static analysis of the source code does not capture all cases.

For each package, we consider dependencies to be the recursive union of all requirements for all releases. Specifically we collapse all releases of a package into a single node in the dependency graph. While this loses some of the available information, it is sufficient for this prelimenary analysis of the PyPI ecosystem.

Empirical and Modeling Results

Our data collection process created a network with 66,536 nodes and 72,939 edges. Over half of the nodes, 33,573, have no edge. This isolates them from the dependency network. Of the remaining 32,963, 31,473 belong to a single giant connected component. Complex networks often exhibit the preponderance of a single connected component like this.

Statistical properties of the software dependency network

The PyPI package dependency network resembles classical complex networks, with some notable departures.

A early claim in complex network theory by [Newman2002], [Newman2003] is that random complex networks will exhibit negative degree assortativity, and that social networks will exhibit positive degree assortativity due to homophily or other effects of group membership on network growth. [Noldus2015] notes that in directed graphs, there are four variations on the degree assortativity metric as for each pair of adjacent nodes one can consider each node's in-degree and out-degree. The degree assortativity metrics for the PyPI dependency graph are given in Table 1.

The PyPI package dependency network notably has *in-in* degree assortativity of 0.19, and *out-in* degree assortativity of -0.16. The *in-out* and *out-out* degree assortativities are both close to zero. We have constructed the graph with the semantics that an edge from A to B implies that A depends on B.

This is a strange structure because its assortativity measures defy the assortativity patterns seen in other complex networks. One reason is that there is much greater variation in out-degree than in in-degree. Table 2 shows the top ten most depended on packages. Table 3 shows the top ten packages with the most dependencies. Three packages, `requests`, `six`, and `django` have out-degree over 1000.

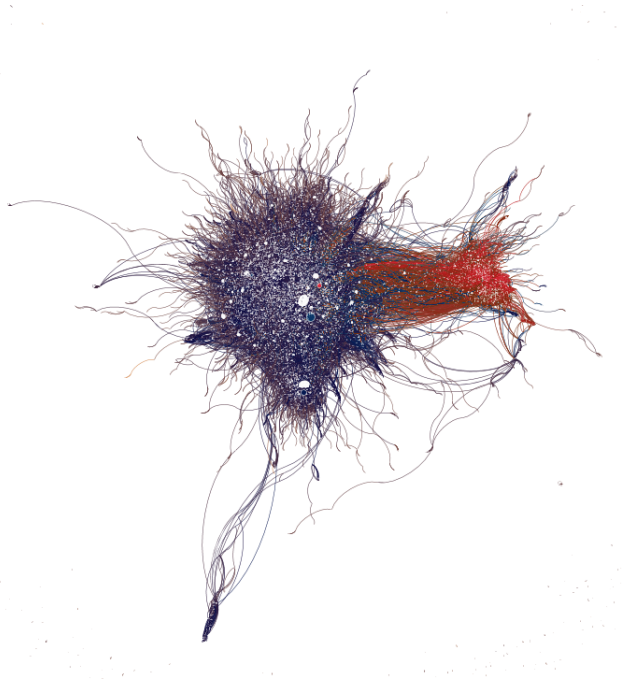

Fig. 1: *Visualization of PyPI dependency network. Annotated dependency graph (see* Implementation of risk computation*) was exported as .gexf, loaded into Gephi [Bastian2009], styled using Force Atlas layout, and colored by ecosystem risk property. This visualization does not include singleton nodes with zero degree, which are the vast majority of nodes. Node size is proportional to out degree. Nodes are colored by the log (base 10) of package ecosystem risk. Red nodes are higher risk. The large red cluster consists of projects related to the Zope web application server, including the Plone content management system.*

Metric	Value
in-in	0.19
in-out	0.05
out-in	−0.16
out-out	−0.04

TABLE 1: *Degree assortativity metrics for the PyPI dependency graph.*

Hot spot analysis

Our analysis suggests that the riskiest packages in the Python ecosystem are those that are part of the Zope web application server and the Plone content management system (CMS) built on it. The Zope community has declared that Zope is now a legacy system and does not recommend that developers use these projects. Therefore, our analytic findings are consistent with community and domain knowledge regarding the resilience of these communities. Despite these warnings, the Plone community is still active and many web sites may still depend on this legacy technology. This study motivates further work on the resilience of Zope to new security threats.

The security properties of Plone have been the subject of considerable informal debate. [Walsh2011] noted that Plone has an order of magnitude lower number of vulnerabilites reported in Mitre's Common Vulnerabilities and Exposures database compared to other popular CMSes like Joomla, Drupal, and Word-

Package	Out-Degree
requests	2125
six	1381
django	1174
pyyaml	775
zope.interface	663
lxml	619
flask	607
python-dateutil	599
zope.component	550
jinja2	507

TABLE 2: *Top ten most dependencies.*

Package	Out-Degree
plone	92
mypypi	53
invenio	52
ztfy.sendit	48
ztfy.blog	47
smartybot	47
icemac.addressbook	41
sentry	40
products.silva	38
ztfy.scheduler	37

TABLE 3: *Top ten packages by number of dependencies.*

press. This has lead Wikipedia [Wiki2016] to assert that Plone's security record is cause of its widespread adoption by government and non-government organizations. [Byrne2013] has challenged this conventional wisdom, noting that the high number of recorded vulnerabilites may just as likely be due to the much greater popularity of the other CMS's. That Drupal, Wordpress, and Joomla are all written in PHP is another confounding factor, as PHP may

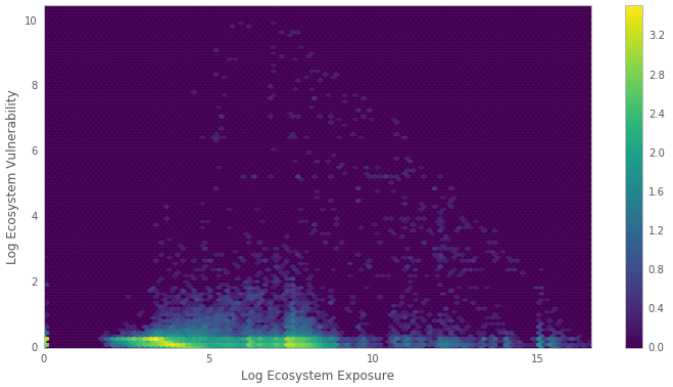

Fig. 2: *Hex plot of log vulnerability and log exposure of each package, with bin density scored on log scale. All logs are base 10. Exposure is more widely distributed than vulnerability. Vulnerability scores for the vast majority of packages are low. There is a fringe of packages that are either highly vulnerable, highly exposed, or both. There is a log-linear tradeoff between high vulnerability and high exposure. This is most likely due to the fact that ecosystem vulnerability and ecosystem exposure both depend on an package's position in the dependency network. Rendered with Matplotlib [Hunter2007].*

be a language prone to security problems. Drupal, Joomla, and Wordpress are beyond the scope of our study, which is concerned only with the PyPI ecosystem. In our risk analysis, Plone scores poorly compared to other Python web frameworks such as Django and Flask. We take this as an indication that beyond its scientific merits, our risk analysis method can provide actionable insights into security that are relevant to practicing software engineers.

We have also identified `six`, a Python 2 and Python 3 compatibility library, as an ecosystem risk hot spot. The second most depended on project in PyPI, `six` inherits its exposure from all of its downstream descendants. For this reason, it is important to ensure that `six` does not have any security-related flaws.

We must admit that there is another reason why the Plone ecosystem has score highly in software risk. The Zope and Plone packages are notably dense in their dependency connectivity. In the original dependency network, before cycles were excised from the graph by removing edges, many Zope and Plone packages were implicated in large cycles of mutual dependency. Even with many of these edges removed, it is possible that packages in this subsystem are more likely to be linked by multiple disitinct paths. With our present algorithm, this would result in some packages being double counted. Due to this technical complication, we must conclude that our results, though suggestive, are only tentative pending future work.

Discussion and future work

We have synthesized techniques from computer security and disaster risk reduction to develop a novel method of predicting risk in the software ecosystem. This fits within the broad scope of supply chain analysis, though we recognize that the software ecosystem as a whole is not merely a chain, but a complex network with a distinctive topology. We approach risk analysis as a science that employs static analysis techniques but also looks more broadly at developer communities and the rate and flow of their activities and communications. This paper proposes a framework of predicting risk in software infrastructure based on static analysis of package dependencies, metadata about downloads and release schedules, and minimal assumptions about the distribution of exposure and vulnerability in software. We have demonstrated the implications of this framework using the PyPI package ecosystem.

A major shortcoming of our analysis is the lack of validation against a gold standard data of *ground truth* regarding software risk. In future work, we will test this framework using other data sets, including data from project issue trackers (such as GitHub) and Common Vulnerabilities and Exposure (CVE) data. We anticipate that linking this data with package dependencies will require a non-trivial amount of work on entity resolution. It is an open question to what extent this framework is useful for assessing software robustness (absence of software errors that can be exploited, for example) and software resilience (capacity of software development communities to respond to known exploits).

There is also room to improve our data preprocessing in future work. For the work in this paper, Python dependencies were discovered using crude static analysis. We used a regular expression to parse each package's `setup.py` file. Python requirements are in fact determined upon package installation by executing Python code. We can get more accurate data by running the setup scripts and extracting requirements from the resulting Python objects.

We simplified the dependency graph by considering any requirement relation between any versions of two packages to be

	Log Eco. Risk	Log Eco. Vulnerability	Log Eco. Exposure	Fragility	Num. Releases	Downloads	In Degree	Out Degree
zope.app.publisher	17.54	6.95	10.59	0.04	26	232460	24	54
zope.app.form	17.54	6.89	10.64	0.04	26	265370	19	45
five.formlib	17.44	6.47	10.97	0.20	5	127280	13	10
plone	17.44	2.37	15.07	0.01	79	387614	96	152
zope.interface	17.42	10.48	6.94	0.03	31	8685819	0	841
zope2	17.41	6.40	11.01	0.03	32	241354	28	163
zope.traversing	17.32	8.40	8.92	0.04	28	367494	9	181
zope.schema	17.29	9.61	7.68	0.03	31	624429	4	399
zope.site	17.28	7.60	9.68	0.07	14	255063	9	72
zope.container	17.27	7.73	9.54	0.05	20	294873	20	119

TABLE 4: *Highest risk Python packages. All logs base 10.*

sufficient for an edge in the final graph. In reality, package requirements configurations often refer to specific versions or version ranges in their dependencies. In order to take this into account, we will need to reexamine our risk model and its assumptions about vulnerability and exposure propagation. A fully dynamic version of our risk model would also take into account how proxy variables such as number of unique downloads change between versions.

The research presented here deals exclusively with data about technical organization. However, as we expand into research into how software communities and their interactions are predictive of software risk, we must be mindful of ethical considerations. Though all the data we intend to use is public and more importantly known to be public in the context of software development, study of human subjects is nevertheless sensitive. Our research agenda depends critically on maintaining the trust of the developer communities we study. For this reason we are dedicated to ecosystems and software projects, which aggregate individual efforts, as the fundamental unit of analysis.

Acknowledgements

We gratefully acknowledge David Lippa, Kyle Niemeyer, and J. Edward Pickle for their helpful comments.

REFERENCES

[Bastian2009] Bastian, Mathieu, Sebastien Heymann, and Mathieu Jacomy. "Gephi: an open source software for exploring and manipulating networks." ICWSM 8 (2009): 361-362.

[Benthall2016] Sebastian Benthall. (2016). PyPI Packages Annotated. Zenodo. 10.5281/zenodo.57563

[Byrne2013] Byrne, Tony. "Is Plone Really More Secure Than Drupal and Joomla?" Web log post. Real Story Group. N.p., 11 Feb. 2013. Web. 23 June 2016.

[Clauset2007] A. Clauset, C.R. Shalizi, and M.E.J. Newman. Power-law distributions in empirical data. arXiv:0706.1062, June 2007.

[Mitzenmacher2003] Mitzenmacher, M. 2003. "A Brief History of Generative Models for Power Law and Lognormal Distributions." Internet Mathematics Vol. 1, No. 2: 226-251

[CensusProject] Census Project. (n.d.). Retrieved July 12, 2016, from https://www.coreinfrastructure.org/programs/census-project

[Cardona2012] Cardona, Omar-Daria, et al. "Determinants of risk: exposure and vulnerability." (2012).

[Girardot2013] O. Girardot. STATE OF THE PYTHON/PYPI DEPENDENCY GRAPH. 2013

[Hagberg2008] 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

[Haney2016] David Haney. 2016. "NPM & left-pad: Have We Forgotten How To Program?" http://www.haneycodes.net/npm-left-pad-have-we-forgotten-how-to-program/

[Hunter2007] Hunter, J. D. (2007). Matplotlib: A 2D graphics environment. Computing in science and engineering, 9(3), 90-95. http://dx.doi.org/10.5281/zenodo.44579

[IonChannel] (n.d.). Retrieved July 12, 2016, from http://ionchannel.io/

[LaBelle2004] N. LaBelle, E. Wallingford. 2004. Inter-package dependency networks in open-source software.

[Nagappan2005] Nagappan, N., & Ball, T. (2005, May). Use of relative code churn measures to predict system defect density. In Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005. (pp. 284-292). IEEE.

[Nagappan2007] Nagappan, N., & Ball, T. (2007). Explaining failures using software dependences and churn metrics. In Proceedings of the 1st International Symposium on Empirical Software Engineering and Measurement.

[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

[Schweik2012] C. Schweik and R. English. *Internet Success: A Study of Open-Source Software Commons*, The MIT Press. 2012

[Verdon2004] D. Verdon and G. McGraw, "Risk analysis in software design," in IEEE Security & Privacy, vol. 2, no. 4, pp. 79-84, July-Aug. 2004.

[Walsh2011] Walsh, M. (2011, March 11). Gov 2.0 guide to Plone. Retrieved June 23, 2016, from http://www.govfresh.com/2011/03/gov-2-0-guide-to-plone/

[Wagner2000] David A. Wagner. 2000. Static Analysis and Computer Security: New Techniques for Software Assurance. Ph.D. Dissertation. University of California, Berkeley. AAI3002306.

[Wiki2016] Plone (software). (2016, May 5). In Wikipedia, The Free Encyclopedia. Retrieved 18:20, June 23, 2016, from https://en.wikipedia.org/w/index.php?title=Plone_(software)&oldid=718838043

[Wheeler2014] Wheeler, David A. How to Prevent the next Heartbleed. 2014-10-20. http://www.dwheeler.com/essays/heartbleed.html

[Wheeler2015] D. Wheeler and S. Khakimov. *Open Source Security Census: Open Source Software Projects Needing Security Investments*, Institute for Defense Analysis. 2015

[Yamin2013] Yamin, Luis Eduardo; Ghesquiere, Francis; Cardona, Omar Dario; Ordaz, Mario Gustavo. 2013. Modelacion probabilista para la gestion del

riesgo de desastre. Washington DC ; World Bank. http://documents.worldbank.org/curated/en/2013/07/18100020/colombia-probabilistic-modeling-disaster-risk-management-modelacion-probabilista-para-la-gestion-del-riesgo-de-desastre

Launching Python Applications on Peta-scale Massively Parallel Systems

Yu Feng[‡§*], Nick Hand[‡]

https://youtu.be/CfrRDI71vTc

Abstract—We introduce a method to launch Python applications at near native speed on large high performance computing systems. The Python run-time and other dependencies are bundled and delivered to computing nodes via a broadcast operation. The interpreter is instructed to use the local version of the files on the computing node, removing the shared file system as a bottleneck during the application start-up. Our method can be added as a preamble to the traditional job script, improving the performance of user applications in a non-invasive way. Furthermore, we find it useful to implement a three-tier system for the supporting components of an application, reducing the overhead of runs during the development phase of an application. The method launches applications on Cray XC30 and Cray XT systems up to full machine capacity with an overhead of typically less than 2 minutes. We expect the method to be portable to similar applications in Julia or R. We also hope the three-tier system for the supporting components provides some insight for the container based solutions for launching applications in a development environment. We provide the full source code of an implementation of the method at *https://github.com/rainwoodman/python-mpi-bcast*. Now that large scale Python applications can launch extremely efficiently on state-of-the-art super-computing systems, it is time for the high performance computing community to seriously consider building complicated computational applications at large scale with Python.

Index Terms—Python, high performance computing, development environment, application

Introduction

The use of a scripting or interpreted programming language in high performance computing has the potential to go beyond post-processing and plotting results. Modern super-computers support dynamic linking and shared libraries, and thus, are capable of running the interpreters of a scripting programming language. Modern interpreters of scripting languages are equipped with the Just-In-Time (JIT) compilation technique that compiles the script in-time to achieve performances close to C or Fortran [LPS15], [BEKS14], [Aut06]. The Python programming language is of particular interest due to its large number of libraries and its wide user base. There are several Python bindings of the Message Passing Interface (MPI)[1] [DPKC11], [Mil02]. Bindings for higher level abstractions, e.g. [Spo12] also exist, allowing one to write complex parallel applications with MPI for simulations and data analysis.

However, it is still traditionally believed that Python does not coexist with large scale high performance computing. The major barrier is due to the slow and unpredictable amount of time required to launch Python applications on such systems. For example, it has been shown that the start-up time sometimes increases to hours for jobs with a relative small scale (a hundred ranks ..[#rank]). Some quantitative benchmarks can be see in [Fro13], [Lan12b].

The issue is an interplay between the current file system architecture on Peta-scale systems and the behavior of the Python interpreter. Peta-scale systems are typically equipped with a shared file system that is suitable for large band-width operations. The meta-data requests are handled by the so-called metadata servers, and usually, at most one master meta-data server serves all requests to a large branch of the file system; then, the data files are replicated to several data storage nodes [Sch03]. As an example, the Phase-I Cray XC 40 system Cori at NERSC is connected to 5 metadata servers (MDT) [NER15]. Because the file system is a shared resource with a limited throughput, it is relatively easy for an application to flood the file systems with requests and nearly bring an entire file system to a halt -- a phenomona most users to HPC systems are very familiar with.

Unfortunately, the Python interpreter is such an application, as has been repeatedly demonstrated in previous studies [Lan12a], [Lan12b], [Fro13], [ERSM11]. During start-up, a Python application will generate thousands of file system requests to locate and import files for scripts and compiled extension modules. We demonstrate the extent of the problematic behavior in Figure 1, where we measure the number of file system requests associated with several fairly commonly used Python packages on a typical system (Anaconda 2 and 3 in this case). The measurement is performed with `strace -ff -e file`. For either Python 2 or Python 3, the number of file system operations increases linearly with the number of entries in `sys.path` (controlled by the `PYTHONPATH` environment variable). Importing the scipy package with 10 additional paths requires 5,000+ operations on Python 2 and 2,000 operations on Python 3. Extrapolating to 1,000 instances or MPI ranks, the number of requests reaches 2 ~ 5 million. On a system that can handle 10,000 file system requests

* *Corresponding author: yfeng1@berkeley.edu*
‡ *Berkeley Center for Cosmological Physics, University of California, Berkeley CA 94720*
§ *Berkeley Institute for Data Science, University of California, Berkeley CA 94720*

1. Message Passing Interface (MPI) is the standard programming model on high performance computing. For readers that are unfamiliar with such topics, we recommend [Qui03] for an introduction to parallel programming and MPI.
2. A rank is defined as one of the concurrent processes of the application.

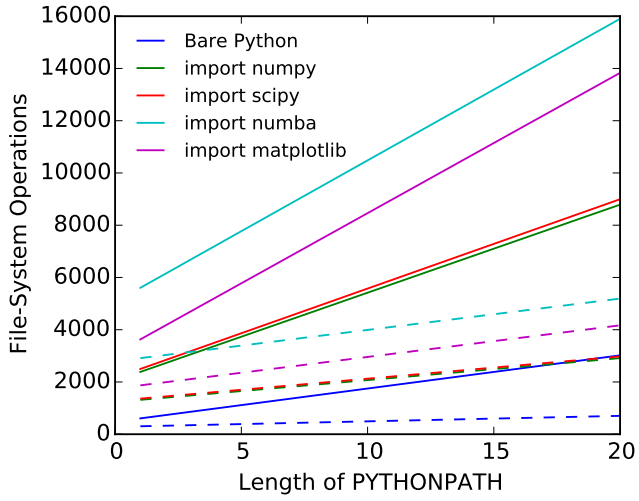

Fig. 1: *Number of file system requests during Python start-up. Solid lines: Python 2. Dashed lines: Python 3. We increase the number of entries in* `PYTHONPATH` *to simulate the number of packages installed in user directory or loaded via* `modules` *command commonly used on HPC systems.*

per second, consuming these requests takes 200 ~ 500 seconds of the full capacity of the entire system. Furthermore, the application becomes extremely sensitive to the load on the shared file system: when the file system is heavily loaded, the application will start extremely slowly.

It is worth pointing out that although the number of requests per rank can be significantly reduced, the total number of requests still increases linearly with the number of MPI ranks, and will become a burden at sufficiently large scale. For example, due to improvements in the importing system, the number of requests per rank is reduced by 50% in Python 3 as compared to Python 2 (seen in Figure 1). Therefore, a plain Python 3 application will handle twice as many ranks as Python 2 does.

In this paper, we present a solution (which we name `python-mpi-bcast`) that addresses the start-up speed without introducing a burden on the users. We have been using this method to launch data analysis applications in computational cosmology (e.g. [FH16]) at National Energy Research Scientific Computing Center (NERSC).

In Section 2, we collect and overview the previous solutions developed over the years. In Section 3, we describe our solution python-mpi-bcast. In Section 4, we discuss the management of the life-cycles of components. In Section 5, we demonstrate the cleanness of python-mpi-bcast with an example script. We conclude this paper and discuss possible extensions of this work in Section 6.

Previous Solutions

Given the importance and wide-adoption of the Python programming language, the application launch time issue has been investigated by several authors. We briefly review them in this section. These solutions either do not fully solve the problem or introduce a burden on the users to maintain the dependency packages.

The application delivery mechanism on a super-computer can deliver the full binary executable to the computing nodes. In fact, older systems can only deliver one statically linked executable file to the computing nodes during the job launch. The support of dynamic libraries on Cray systems was once very limited [ZDA+12] -- a significant amount of work has been invested to solve this limitation in the context of shared library objects (e.g. [AA14]).

One can take advantage of the standard delivery mechanism and launch the application at an optimal speed, by bundling the entire support system of the Python application as one statically compiled executable. [Fro13], [PM12] both fall into this category. We also note that the yt-project has adopted some similar approaches for their applications [TSO+11]. While being a plausible solution, the technical barrier of this approach is very high. Statically compiled Python is not widely used by the mainstream community, and special expertise is required to patch and incorporate every dependency package for individual applications. Although the steps are documented very well, the effort is beyond the knowledge of a typical Python developer.

Fortunately, in recent years the support for dynamic libraries on high performance computing systems has significantly improved, as super-computing vendors began to embrace a wider user base for general, data-intensive analysis. On these platforms, the main bottleneck has shifted from the lack of support for dynamic libaries to the vast number of meta-data requests to import the full python runtime library.

A particularly interesting approach is to eliminate the meta-data requests altogether via caching. Caching can happen at the user level or operation system level. On the user level, mpiimport [Lan12b] and Scalable Python cite:*scalablepython* attempt to cache the meta-data requests with an import hook. After the hooks are enabled, the user application are supposed to run as is. Unfortunately, these methods are not as fully opaque as they appear to be. With import hooks, because the meta-data requests are cached, they have to be calculated by the root rank first. Therefore, an implicit synchronization constraint is imposed in order to ensure the cache is evaluated before the requests from the non-root ranks. All of the import operations must be made either collectively or un-collectively at the same time. We find that the collective importing scheme breaks site.py in the Python standard library and the un-collective importing scheme breaks most MPI-enabled scripts. At the system level, users can file a ticket to mark a branch of the file system as immutable, allowing the computing nodes to cache the requests locally. This requires special requirements from the administrators, and in practice the relief has been limited.

Finally, one can locally mount a full application image on the computing node via a container-based solution [JCGB15]. The loopback mount adds a layer of caching to reduce the number of requests to the global file system. The drawback of the container-based solution is due to the requirement that the entire application is built as one image. Each time the application code is modified, the entire image needs to be re-generated before the job is ready to run. On super computing systems, it takes a long (and fluctuating) amount of time to build a non-trivial software package. Some of our support libraries (e.g. pfft-python) usually takes 10 to 20 minutes to rebuild from scratch. This waiting time can become an additional burden during code development. Furthermore, the user may need special privileges on the computing nodes in order to mount the images, requiring changes in the system security policy that can be challenging to implement for administration reasons; though we note that shifter has solved this problem at NERSC.

Our Solution: python-mpi-bcast

In this section, we show that the shared file system bottleneck can be solved with a much simpler approach that maintains a high level of compatibility with the main stream usage of the Python programming language.

Compatibility is maintained if one uses the vanilla C implementation of Python without any modifications to the import mechanism. A large number of file system requests during application start-up will be made, but we will reroute all shared file system requests to local file systems on the computing nodes, away from the limited shared file-system.

This is possible because the package search path of a Python interpreter is fully customizable via a few environment variables, a feature widely used in the community to provide support for 'environments' [LMR15], [Con15]. With python-mpi-bcast, we make use of this built-in relocation mechanism to fully bypass the scalability bottleneck of the shared file system. We note that none of the previous solutions make extensive use of this feature.

Because all file operations for importing packages are local after the re-routing, the start-up time of a Python application becomes identical to that of a single rank, regardless of the number of ranks used.

The only additional cost of our approach results from the delivery of the packages to the local file systems. In order to efficiently deliver the packages, we bundle the packages into tar files. The MPI broadcast function is used for the delivery. The tar files are uncompressed automatically with the tool `bcast.c` that could be linked into a static executable.

We will describe the steps in the following subsections:

1) Create bundles for dependencies and the application.
2) Deliver the bundles via broadcasting. The destination shall be a local file system on the computing nodes. (e.g. /dev/shm or /tmp)
3) Reroute Python search path (including shared library search path) to the delivery destination, bypassing the shared file system.
4) Start the Python application the usual way.

Creating bundles

We define a bundle as a compressed tar file that contains the full file system branch of a package or several packages, starting from the relative Python home directory. Three examples are:

1) The bundle file of a conda environment consists of all files in the bin, lib, include, and share directories of the environment. We provide a script (tar-anaconda.sh) for generating such a bundle from a conda environment. The size of a bundle for a typical conda environment is close to 300 MB.

2) The bundle file of a PIP installed package consists of all files installed by the `pip` command. We provide a wrapper command `bundle-pip` for generating a single bundle from a list of PIP packages.

3) The bundle file of basic system libraries includes those shared library files that are loaded by the dynamic linker for the Python interpreter. We provide three sample job scripts to generate these bundles for three Cray systems: XC30, XC40, and XT. The system bundle addresses the shared library bottleneck investigated in [ZDA+12] (DLFM) but without requiring an additional wrapper of the system dynamic linker.

The bundles only need to be updated when the dependencies of an application are updated.

Variable	Action
`PYTHONHOME`	Set to broadcast destination
`PYTHONPATH`	Purge
`PYTHONUSERBASE`	Purge
`LD_LIBRARY_PATH`	Prepended by `/lib` of the broadcast destination

TABLE 1: *Environment Variable used in* `python-mpi-bcast`

Delivery via broadcasting

Before launching the user application, the bundles built in the previous step must be delivered to the computing nodes -- we provide a tool for this task. On Cray systems, we make use of the memory file system mounted at `/dev/shm`. On a system with local scratch, `/tmp` may be used as well, although this has not been tested.

We use the broadcast function of MPI for the delivery. The tool first elects one rank per node to receive and deploy the bundles to a local storage space. The bundle is then uncompressed by the elected rank per computing node.

The new files are marked globally writable. Therefore, even if some of the files are not properly purged from a node, they can be overwritten by a different user when the same node is allocated to a new job. We note that this may pose a security risk in shared systems.

When several bundles are broadcast in the same job, the later ones will overwrite the former ones. This overwriting mechanism provides a way to deliver updates as additional bundles.

We also register an exit handler to the job script that purges the local files to free up the local file system. This step is necessary on systems where the local storage space is not purged after a job is completed.

Rerouting file system requests

We list the environment variables that are relevant to the relocation in Table 1. After the relocation, all of the file system requests (meta-data and data) are rerouted to the packages in the local file system. As a result, the start-up time of the interpreter drops to that of a single rank.

We note that the variable PYTHONUSERBASE is less well-known, documented only in the `site` package, but not in the Python command-line help or man pages. If the variable is not set, Python will search for packages from the user's home directory `$HOME/.local/`. Unfortunately, the home file-system is typically the slowest one in a Peta-scale system. This directory is not part of the application, therefore we purge this variable by setting it to an invalid location on the local file system, the root of the broadcast destination. We also purge `PYTHONPATH` in the same manner, since all packages are located at the same place. The variable `PYTHONPATH` can be very long on systems where each Python package is provided as an individual module of the `modules` system. This negtively impacts the performance of launching Python applications, as we see in Figure 1, which clearly shows that the length of `PYTHONPATH` has a huge impact on the number of file system operations that occur during start-up.

Launching the Python application

We launch the Python application via the standard python-mpi wrapper provided by `mpi4py`. We emphasize that no modifica-

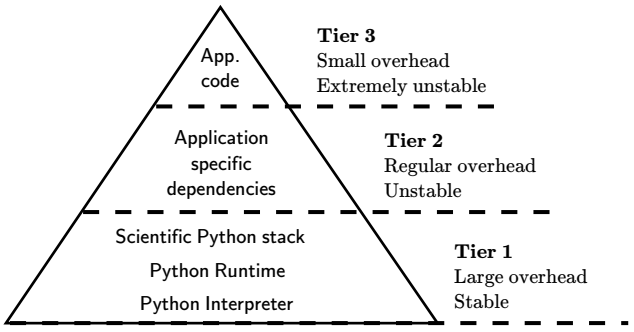

Fig. 2: *Three tiers of bundles. The most stable component (bottom of the pyramid, Tier 1) takes the most effort to build. The least stable component (top of the pyramid, Tier 3), takes the least effort to bundle. The split into three tiers allows the developers to save time in maintaining the bundles.*

tions to the python-mpi wrapper or to the user application are needed in our approach.

It is important to be aware that Python prepends the parent directory of the start-up script to the search path. If the start-up script of the application resides on a shared file system, the access to this directory will slow down the application launch. As an alternative, the application script (along with the full directory tree) can also be bundled and delivered via python-mpi-bcast before the launch. This is demonstrated in the example in Section 5, and we will discuss this case in more detail in the next section.

On a Cray system, the Python interpreter (usually `python-mpi`) must reside in a location that is accessible by the job manager node, because it will be delivered via the standard application launch process.

Three-tiers of bundles

Building bundles takes time and shifts the focus of the developer from application development to interfacing with the system. We therefore recommend to organize the components of an application into a three-tier system to minimize the redundant efforts required to to create bundles. The three-tier system is illustrated in Figure 2, and we describe the rationale and definitions in the following sections.

Tier 1 components

Tier 1 components consist of the Python interpreter, standard runtime libraries, and stable dependencies (dependencies that changes infrequently, for example, numpy, scipy, mpi4py, h5py). On a conda based Python distribution, the Tier 1 components map to the packages included in a conda environment. These components provide a basic Python computing environment, take the most time to install, yet barely change during the life-cycle of a project. Most super-computing facilities already maintain some form of these packages with the `modules` system, e.g. NCSA has a comprehensive set of Python packages [Ms14], and NERSC has the anaconda 2 and 3 based Python distribution.

It is straightforward to create bundles of these pre-installed components. We provide the `bundle` command with python-mpi-bcast for creating a bundle from a pre-installed 'modules' path. It is a good practice to create one bundle for each 'modules' path. The process can be time consuming, even though it does not invole compiling any source code packages. For example, creating a Tier 1 bundle from a full binary anaconda installation typically takes 5 minutes at NERSC facilities.

Tier 2 components

Tier 2 components consist of unstable dependencies of the application. These include packages used or developed specifically for the particular application, which are usually neither part of the conda distribution nor deployed at the computing system by the facility. Tier 2 components update frequently during the life-cycle of a project.

The difference in update-frequency means that Tier 2 components should not be bundled with the Tier 1 components. Since Tier 2 components are usually much smaller and thus faster to bundle than Tier 1 components, bundling them separately reduces the overhead for running and testing the application live at the supercomputing facility.

We provide a pip wrapper script `bundle-pip` with python-mpi-bcast to build bundles for the Tier 2 components. A good practice is to create a single bundle for all of the Tier 2 components with one invocation to the tar-pip.sh wrapper.

Tier 3 components

Tier 3 components are the application itself and other non-package dependencies. These include the main script and files in the same directory as the main script. The Tier 3 components change most frequently among the three tiers during the life cycle of a project. As Tier 3 components mature and receive less frequent changes they should be migrated into Tier 2, following the usual software refactoring practices.

We implement two strategies for Tier 3 components. The *simple* strategy is to leave these files at the original location in the shared file system. In this case, Python will prepend the parent directory of the main script to the search path, not fully bypassing the shared file system. We find that the extra cost due to this additional search is usually small. However, when the system becomes highly congested (an ironic example is when another user attempts to start a large Python job without using our solution), the start-up time can observe a significant slow down.

A consistently reliable start-up time is obtained if Tier 3 components are also bundled and delivered to the local file system (*mirror* strategy). The location of the main script in the job script should be modified to reflect this change. Because the Tier 3 components are the most lightweight, typically consisting of only a few files, a good practice is to create the bundle automatically in the job script, without requiring the developer to manually create a bundle before every job submission. We provide a helper command *mirror* that implementes the strategy. The *mirror* strategy is demonstrated in the next section with examples.

Example Scripts

Generic Cray Systems

In this section, we show an example SLURM job script on a Cray XC 30 system. The script demonstrates the non-invasive nature of our method. After the bundles are built, a few extra lines are added to the job script to enable python-mpi-bcast and deliver the three tiers of components. The user application does not need to be specifically modified for python-mpi-bcast. We emphasize that the job script runs in the user's security context, without any special requirements from the facility.

```
# Script without NERSC integration
# Modify and adapt to use on a general
# HPC system

#! /bin/bash
#SBATCH -n 2048
#SBATCH -p debug

export PBCAST=/usr/common/contrib/bccp/python-mpi-bcast

source $PBCAST/activate.sh \
    /dev/shm/local "srun -n 1024"

# Tier 1 : anaconda
bcast -v $PBCAST/2.7-anaconda.tar.gz \
         $HOME/fitsio-0.9.8.tar.gz

# Tier 2 : commonly used packages
# e.g. installed in $PYTHONUSERBASE
bcast-userbase

# Tier 3 : User application
mirror /home/mytestapp/ \
    testapp bin

# Launch
time srun -n 1024 python-mpi
    /dev/shm/local/bin/main.py
```

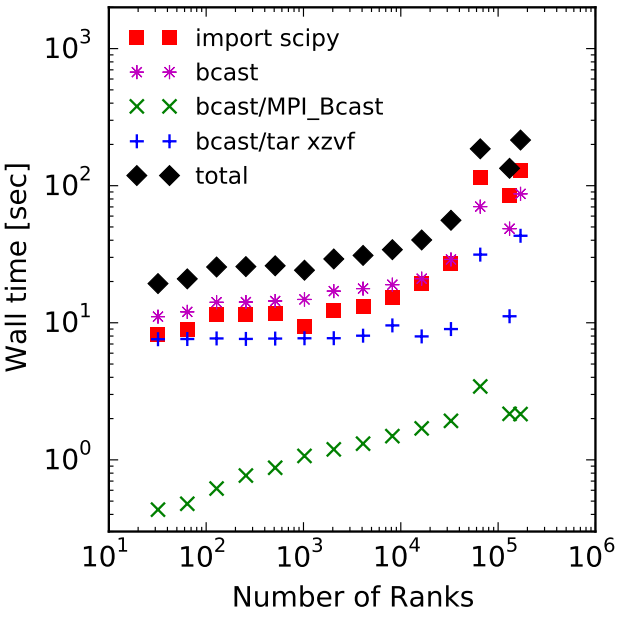

Fig. 3: *Time measurements of python-mpi-bcast on Edison, a Cray XC 30 system at NERSC. We perform tests launching a dummy Python 2 application (that imports scipy) with up to 127,440 MPI ranks. The total time in the bcast job step is shown in stars. The two major time consuming components of bcast, the call to* `MPI_Bcast` *(×) and the call to the* `tar` *command, are also shown (+). Note that large jobs incur a large overhead in the job step such that the sum of the latter differs from the job step times. The total time of the job step that launches the dummy application is shown in squares. The total time of both job steps is shown in diamonds.*

Integration with NERSC Facilities

On the NERSC systems where `python-mpi-bcast` was originally developed, we also provide a default installation of `python-mpi-bcast` that is integrated with the `modules` system and the Anaconda based Python installations. The full integration source code is hosted together in the main python-mpi-bcast repository and can be easily adapted to other systems.

The following script provides an example for using `python-mpi-bcast` in a pre-configured system. Note that the Python runtime environment (along with shared libraries from the Cray Linux Environment) are automatically delivered. The impact on the user application is limited to two lines in the job script: one line for enabling python-mpi-bcast and the other line to mirror the application to a local file system with the `mirror` command.

```
#! /bin/bash
#SBATCH -N 2048
#SBATCH -p debug

# select the Python environment
module load python/3.4-anaconda

# NERSC integration
PBCAST=/usr/common/contrib/bccp/python-mpi-bcast
source $PBCAST/nersc/activate.sh

# Directly deliver the user application
mirror /home/mytestapp/ \
    testapp bin

# launch the mirrored application
time srun -n 1024 python-mpi \
    /dev/shm/local/bin/main.py
```

Benchmark and Performance

In Figure 3 and 4, we show the measurement of wall clock time of python-mpi-bcast for a dummy Python 2 application on the Cray XC30 system Edison at NERSC and the Cray XT system BlueWaters at NCSA. The dummy application imports the scipy package on all ranks before exiting. We point out that in the benchmark it is important to import Python packages as done in a real application, because most of the metadata requests are to locate the Python scripts of packages rather than dynamic libraries associated with extension modules. Therefore, a benchmark based on performance of `simulating` dynamic libraries [LAdS+14] does not properly represent the true launch time of a realistic Python application. We do not perform another set of benchmarks for Python 3, but note that the stream-lined import system in Python 3 could perform better than Python 2. [van02]

The job includes two steps: the first involves the statically linked bcast program that delivers the bundles to the computing nodes (which does not involve Python), and the second launches the Python application.

The bcast step consists of two major components, a call to `MPI_Bcast` and a call to `libarchive`[Tim09] to inflate the tar ball. We observe that the scaling in the `MPI_Bcast` function is consistent with the expected $O[\log N]$ scaling of a broadcast algorithm. The call to inflate the tar ball remains roughly constant, but shows fluctuations for larger runs on the XC30 system. This is likely because the job has hit a few nodes that are in a non-optimal state, which is a common effect in jobs running near the capacity of the system.

As a further evidence, the fluctuation in the large jobs correlates with an increase in the time spent in the 'tar' stage of the bcast time step, as seen by comparing the tests with 49,152 ranks (2048 nodes), 98,304 ranks(4096 nodes), and 127,440 ranks (5310 nodes).

The time spent in the Python application (second job step) increases slowly as well, but the increase becomes more significant

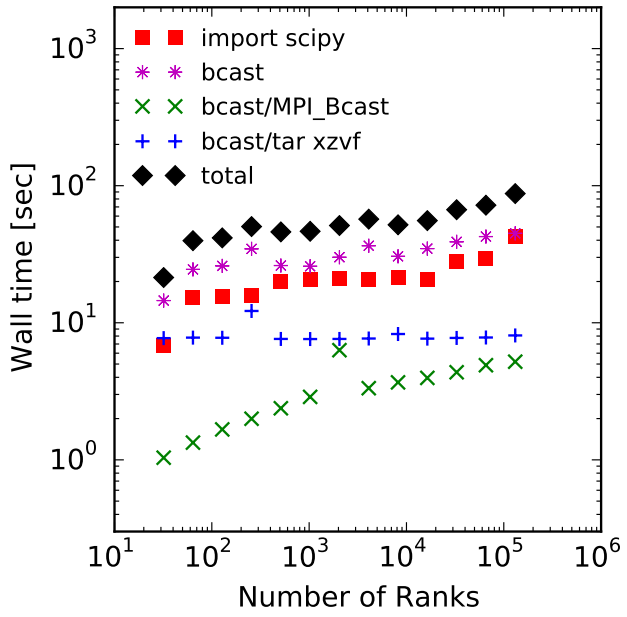

Fig. 4: *Time measurements of python-mpi-bcast on BlueWaters. a Cray XT system at NCSA. We perform tests launching a dummy Python 2 application (that imports scipy) with up to 127,440 MPI ranks. The total time in the bcast job step is shown in stars. The two major time consuming components of bcast, the call to* `MPI_Bcast` *(×) and the call to the 'tar' command, are also shown (+). Note that large jobs incur a large overhead in the job step such that the sum of the latter differs from the job step times. The total time of the job step that launches the dummy application is shown in squares. The total time of both job steps is shown in diamonds.*

as the size of the job approaches the capacity of the system. An additional cause of the increase can be attributed to the remaining few requests to the shared file system for unbundled shared libraries and Python configuration files that are not rerouted. For example, the configuration of mpi4py package is hard coded on the shared file system.

For jobs with less than 1024 nodes, the timing is close to 1 minute. In any case, the largest test on Edison that employs 127,440 MPI ranks (5310 nodes), spent 4 minutes in total for launching the application. We note that the slightly smaller job that employs 98,304 ranks (4096 nodes) spent less than 2 minutes in total.

Conclusions

We introduce `python-mpi-bcast`, a solution to start native Python applications on large, high-performance computing systems.

We summarize and review a set of previous solutions developed over the years and with varying usage in the community. Their limitations in terms of practical usability and efficiency are discussed.

Our solution `python-mpi-bcast` does not suffer from any of the drawbacks of previous solutions. Using our tool, the runtime environment of the Python application on Peta-scale systems is fully compatible with the the mainstream Python environment. The entire solution can be added as a preamble to a user job script to enhance the speed and reliability of launching Python applications on any scales, from a single rank to thousands of ranks.

Our solution makes use of the established infrastructure of the mainstream Python community to reroute support packages of an application from the shared file system to local file systems per node via bundles. The solution is compatible with Python 2 and 3 at the same time. Almost all accesses to the shared file system are eliminated, which avoids the main bottleneck typically encountered during the start-up stage of a Python application. We have performed tests up to 127,440 ranks on a Cray XC 30 system (limited by the available cores on the Edison system at NERSC) and on a Cray XT system BlueWaters at NCSA. There is no fundamental reason that the method does not scale to even larger jobs, given that the only non-local operation is a broadcast operation.

We introduce a three-tier bundling system that reflects the evolutionary nature of an application. Different components of an application are bundled separately, reducing the preparation overhead for launching an application during the development stage. The three-tier system is an improvement from the all-in-one approaches such as [Fro13] or [JCGB15]. We in fact advocate adopting a similar system in general-purpose, images-based application deployment infrastructure (e.g. in cloud computing). We note that a large burden from the users can be further removed if the computing facilities maintain the Tier 1 bundle(s) in parallel with their existing `modules` system. Further integration into the job system is also possible to provide a fully opaque user experience.

Finally, with few modifications, `python-mpi-bcast` can be easily generalized to support applications written in other interpreted languages such as Julia and R. In addition, we highly welcome reimplementing the stratagies documented in the paper as an extension of the Conda package distribution system, and provide the full source code of `python-mpi-bcast` at https://github.com/rainwoodman/python-mpi-bcast.

Given that large-scale Python applications can be launched extremely efficiently on state-of-the-art super-computing systems, it is the time for the high-performance computing community to begin serious development of complex computational applications at large scale with Python.

Acknowledgment

The original purpose of this work was to improve the data analysis flow of cosmological simulations. The work is developed on the Edison system and Cori Phase I system at National Energy Research Super-computing Center (NERSC), under allocations for the Baryon Oscillation Spectroscopic Survey (BOSS) program and the Berkeley Institute for Data Science (BIDS) program. We also performed benchmark on the Blue Waters system at National Center for Super-computing Applications (NCSA) as part of the NSF Peta-apps program (NSF OCI-0749212) for the BlueTides simulation. The authors thank Zhao Zhang of Berkeley Institute of Data Science, Fernando Perez of Berkeley Institute of Data Science, Martin White of Berkeley Center for Cosmology, Rollin Thomas of Lawrence Berkeley National Lab, Aron Ahmadia of Continuum Analysis Inc., for insightful discussions over the topic.

References

[AA14] William Scullin Aron Ahmadia, Jed Brown. Joint anl/ksl collaboration on collective file system module (with glibc dynamic

library interception), 2014. URL: https://github.com/ahmadia/collfs.

[Aut06] V8 Project Authors. V8 javascript engine, 2006. URL: https://chromium.googlesource.com/v8/v8.git.

[BEKS14] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah. Julia: A Fresh Approach to Numerical Computing. *ArXiv e-prints*, November 2014. arXiv:1411.1607.

[Con15] Continuum Analytics, Inc. Conda, 2015. URL: http://conda.pydata.org/docs/.

[DPKC11] Lisandro D. Dalcin, Rodrigo R. Paz, Pablo A. Kler, and Alejandro Cosimo. Parallel distributed computing using python. *Advances in Water Resources*, 34(9):1124 – 1139, 2011. New Computational Methods and Software Tools. URL: http://www.sciencedirect.com/science/article/pii/S0309170811000777, doi:http://dx.doi.org/10.1016/j.advwatres.2011.04.013.

[ERSM11] Jussi Enkovaara, Nichols A. Romero, Sameer Shende, and Jens J. Mortensen. Gpaw - massively parallel electronic structure calculations with python-based software. *Procedia Computer Science*, 4:17 – 25, 2011. Proceedings of the International Conference on Computational Science, ICCS 2011. URL: http://www.sciencedirect.com/science/article/pii/S1877050911000615, doi:http://dx.doi.org/10.1016/j.procs.2011.04.003.

[FH16] Yu Feng and Nick Hand, 2016. URL: https://github.com/bccp/nbodykit.

[Fro13] Bradley M. Froehle, 2013. URL: https://github.com/bfroehle/slither.

[JCGB15] Doug Jacobsen, Shane Canon, Lisa Gerhardt, and Deborah Bard. Shifter: Bringing linux containers to hpc, 2015. URL: https://www.nersc.gov/research-and-development/user-defined-image.

[LAdS+14] Gregory L. Lee, Dong H. Ahn, Bronis R. de Supinski, John Gyllenhaal, and Patrick Miller. The python dynamic benchmark, 2014. URL: https://codesign.llnl.gov/pynamic.php.

[Lan12a] Asher Langton. Email communication to the mpi4py forum., 2012. URL: https://groups.google.com/forum/#!topic/mpi4py/h_GDdAUcviw.

[Lan12b] Asher Langton. An mpi-aware import module for python, 2012. URL: https://github.com/langton/MPI_Import.

[LMR15] Jannis Leidel, Carl Meyer, and Brian Rosner. Virtual python environment builder, 2015. URL: https://pypi.python.org/pypi/virtualenv.

[LPS15] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. Numba: A llvm-based python jit compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, LLVM '15, pages 7:1–7:6, New York, NY, USA, 2015. ACM. URL: http://doi.acm.org/10.1145/2833157.2833162, doi:10.1145/2833157.2833162.

[Mil02] Patrick Miller. pympi – an introduction to parallel python using mpi. 2002.

[Ms14] Colin MacLean and NCSA staff. bwpy: Python on bluewaters, 2014. URL: https://bluewaters.ncsa.illinois.edu/python.

[NER15] NERSC. Cori system specifications, 2015. URL: https://www.nersc.gov/users/computational-systems/cori/cori-phase-i/.

[PM12] Bradley M. Froehle Pat Marion, Aron Ahmadia. Import without a filesystem: scientific python built-in with static linking and frozen modules, 2012. URL: http://conference.scipy.org/scipy2013/presentation_detail.php?id=129.

[Qui03] Michael J. Quinn. *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill Education Group, 2003.

[Sch03] Philip Schwan. Lustre: Building a File System for 1,000-node Clusters. In *Proceedings of the 2003 Linux Symposium*, July 2003.

[Spo12] William F. Spotz. Pytrilinos: Recent advances in the python interface to trilinos. *Sci. Program.*, 20(3):311–325, July 2012. URL: http://dx.doi.org/10.1155/2012/965812, doi:10.1155/2012/965812.

[Tim09] Tim Kientzle and contributors. libarchive: Multi-format archive and compression library. http://libarchive.org, 2009.

[TSO+11] M. J. Turk, B. D. Smith, J. S. Oishi, S. Skory, S. W. Skillman, T. Abel, and M. L. Norman. yt: A Multi-code Analysis Toolkit for Astrophysical Simulation Data. *Astrophysical Journal Supplement*, 192:9–+, January 2011. arXiv:1011.3514, doi:10.1088/0067-0049/192/1/9.

[van02] van Rossum, Just and Moore, Paul . Pep 302 – new import hooks. 2002. URL: https://www.python.org/dev/peps/pep-0302/.

[ZDA+12] Zhengji Zhao, Mike Davis, Katie Antypas, Yushu Yao, Rei Lee, and Tina Butler. Shared library performance on hopper. *Cray User Group*, 2012. URL: https://cug.org/proceedings/attendee_program_cug2012/includes/files/pap124.pdf.