SciPy2017

Proceedings of the 16th Python in Science Conference

July 10 - July 16 • Austin, Texas

Katy Huff
David Lippa
Dillon Niederhut
M Pacer

PROCEEDINGS OF THE 16TH PYTHON IN SCIENCE CONFERENCE

Edited by Katy Huff, David Lippa, Dillon Niederhut, and M Pacer.

SciPy 2017
Austin, Texas
July 10 - July 16, 2017

ISSN:2575-9752
https://doi.org/10.25080/Majora-c9725a0-011

ORGANIZATION

Conference Chairs

PRABHU RAMACHANDRAN, Enthought Inc. & IIT Bombay
SERGE REY, Arizona State University

Program Chairs

LORENA BARBA, George Washington University
GIL FORSYTH, George Washington University

Communications

JUAN SHISHIDO, Twitter

Birds of a Feather

AASHISH CHAUDHARY, Kitware
MATT MCCORMICK, Kitware

Proceedings

KATY HUFF, University of Illinois
DAVID LIPPA, Amazon
DILLON NIEDERHUT, Enthought
M PACER, Berkeley Institute of Data Science

Financial Aid

SCOTT COLLIS, Argonne National Laboratory
ERIC MA, MIT

Tutorials

ALEXANDRE CHABOT-LECLERC, Enthought
MIKE HEARNE, USGS
BEN ROOT, Atmospheric and Environmental Research, Inc.

Sprints

JONATHAN ROCHER, KBI Biopharma
CHARLYE TRAN, Lewis University and The Recurse Center

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

David Lippa
Katy Huff
M Pacer
Dillon Niederhut
Chien-Hsiu Lee
Priya Sawant
Guy Tel-Zur
Nick Wan
Alejandro de la Vega
George Markomanolis
Matthew Rocklin
Oleksandr Pavlyk
Jean Bilheux
Brian McFee
Alejandro Weinstein
Stefan van der Walt
Li Wang
Serge Guelton
Kapil Saraswat
Thomas Arildsen
Paul Celicourt
Cyrus Harrison
Kyle Niemeyer
Sartaj Singh
Ankur Ankan
Nathan Goldbaum
Angelos Krypotos
Jaime Arias
Jonathan J. Helmus
Fernando Chirigati
Chris Calloway
Nicholas Malaya
Daniel Wheeler
Jeremiah Johnson
Yu Feng
Yingwei Yu
Manoj Pandey
James A. Bednar
Adrian Heilbut
Jason Vertrees
Horacio Vargas
Fatih Akici
Sebastian Benthall
Nirmal Sahuji

SCHOLARSHIP RECIPIENTS

WILL BARNES, Rice University
SCOTT COLE, University of California, San Diego
ROBERTO COLISTETE JR, UFES
BJORN DAHLGREN, KTH Royal Institute of Technology
FILIPE FERNANDES, SECOORA
ERIC MA, Massachusetts Institute of Technology
JEREMY MCGIBBON, University of Washington
HIMANSHU MISHRA, NetworkX and Timelab
DANIIL PAKHOMOV, Johns Hopkins University
ZACH SAILER, University of Oregon
CARL SAVAGE, School District 69 (Qualicum)
TOBIAS SCHRAINK, New York University
SCOTT SIEVERT, UWisconsin Madison
SARTAJ SINGH, India Institute of Technology, BHU

JUMP TRADING AND NUMFOCUS DIVERSITY SCHOLARSHIP RECIPIENTS

SELENA ALEMAN, University of Texas at Austin
CELIA CINTAS, CENPAT-CONICET
NATALIA CLEMENTI, The George Washington University
JOHANNA COHOON, The University of Texas at Austin
CHARLYE DELGADO, Lewis University
ALEXANDER IVANOV, The Institute for Information Transmission Problems of Russian Academy of Sciences
DANA MILLER, University of California, Berkeley
REBECCA MINICH, None
JULIETTE SEIVE, University of Texas at Austin
MALVIKA SHARAN, European Molecular Biology Laboratory (EMBL), Heidelberg
CHAYA STERN, Weill Cornell Graduate School
PAMELA WU, New York University
ISRAEL ZÚÑIGA DE LA MORA, Alturin

CONTENTS

SPORCO: A Python package for standard and convolutional sparse representations

Brendt Wohlberg[‡*]

Abstract—SParse Optimization Research COde (SPORCO) is an open-source Python package for solving optimization problems with sparsity-inducing regularization, consisting primarily of sparse coding and dictionary learning, for both standard and convolutional forms of sparse representation. In the current version, all optimization problems are solved within the Alternating Direction Method of Multipliers (ADMM) framework. SPORCO was developed for applications in signal and image processing, but is also expected to be useful for problems in computer vision, statistics, and machine learning.

Index Terms—sparse representations, convolutional sparse representations, sparse coding, convolutional sparse coding, dictionary learning, convolutional dictionary learning, alternating direction method of multipliers

Introduction

SPORCO is an open-source Python package for solving inverse problems with sparsity-inducing regularization [MBP14]. This type of regularization has become one of the leading techniques in signal and image processing, with applications including image denoising, inpainting, deconvolution, superresolution, and compressed sensing, to name only a few. It is also a prominent method in machine learning and computer vision, with applications including image classification, video background modeling, collaborative filtering, and genomic data analysis, and is widely used in statistics as a regression technique.

SPORCO was initially a Matlab library, but the implementation language was switched to Python for a number of reasons, including (i) the substantial cost of Matlab licenses (particularly in an environment that does not qualify for an academic discount), and the difficulty of running large scale experiments on multiple hosts with a limited supply of toolbox licenses, (ii) the greater maintainability and flexibility of the object-oriented design possible in Python, (iii) the flexibility provided by NumPy in indexing arrays of arbitrary numbers of dimensions (essentially impossible in Matlab), and (iv) the superiority of Python as a general-purpose programming language.

SPORCO supports a variety of inverse problems, including Total Variation [ROF92] [All92] denoising and deconvolution, and Robust PCA [CCS10], but the primary focus is on sparse coding and dictionary learning, for solving problems with sparse representations [MBP14]. Both standard and convolutional forms of sparse representations are supported. In the standard form the dictionary is a matrix, which limits the sizes of signals, images, etc. that can be directly represented; the usual strategy is to compute independent representations for a set of overlapping blocks, as illustrated in Figure 1. In the convolutional form [LS99][ZKTF10][Woh16d] the dictionary is a set of linear filters, making it feasible to directly represent an entire signal or image. The convolutional form is equivalent to sparse coding with a structured dictionary constructed from translations of a smaller generating dictionary, as illustrated in Figure 2. The support for the convolutional form is one of the major strengths of SPORCO since it is the only Python package to provide such a breadth of options for convolutional sparse coding and dictionary learning. Some features are not available in any other open-source package, including support for representation of multi-channel images (e.g. RGB color images) [Woh16b], and representation of arrays of

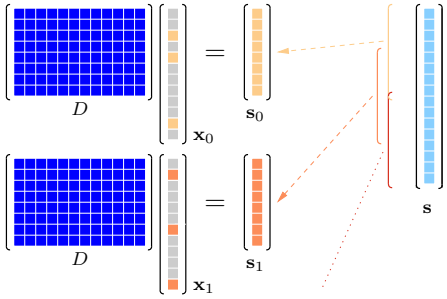


Fig. 1: *Independent sparse coding of overlapping blocks*

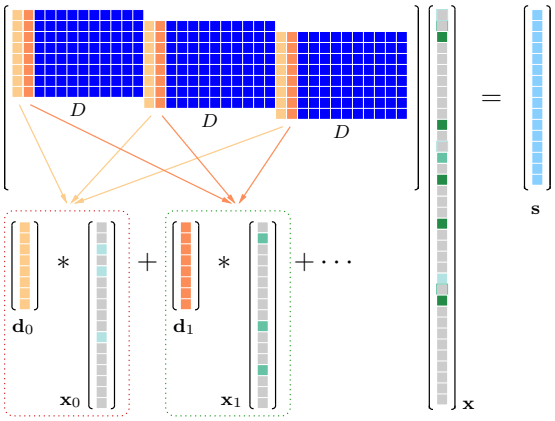


Fig. 2: *Convolutional sparse coding of an entire signal*

* *Corresponding author: brendt@ieee.org*
‡ *Theoretical Division, Los Alamos National Laboratory*

arbitrary numbers of dimensions, allowing application to one-dimensional signals, images, and video and volumetric data.

In the current version, all optimization problems are solved within the Alternating Direction Method of Multipliers (ADMM) [BPC+10] framework, which is implemented as flexible class hierarchy designed to minimize the additional code that has to be written to solve a specific problem. This design also simplifies the process of deriving algorithms for solving variants of existing problems, in some cases only requiring overriding one or two methods, involving a few additional lines of code.

The remainder of this paper provides a more detailed overview of the SPORCO library. A brief introduction to the ADMM optimization approach is followed by a discussion of the design of the classes that implement it. This is followed by a discussion of both standard and convolutional forms of sparse coding and dictionary learning, and some comments on the selection of parameters for the inverse problems supported by SPORCO. The next section addresses the installation of SPORCO, and is followed by some usage examples. The remaining sections consist of a discussion of the derivation of extensions of supported problems, a list of useful support modules in SPORCO, and closing remarks.

ADMM

The ADMM [BPC+10] framework addresses optimization problems of the form

$$\operatorname{argmin}_{\mathbf{x},\mathbf{y}} \; f(\mathbf{x}) + g(\mathbf{y}) \;\text{ such that }\; A\mathbf{x} + B\mathbf{y} = \mathbf{c} \;. \tag{1}$$

This general problem is solved by iterating over the following three update steps:

$$\begin{aligned}
\mathbf{x}^{(j+1)} &= \operatorname{argmin}_{\mathbf{x}} \; f(\mathbf{x}) + \frac{\rho}{2} \left\| A\mathbf{x} - \left(-B\mathbf{y}^{(j)} + \mathbf{c} - \mathbf{u}^{(j)} \right) \right\|_2^2 \\
\mathbf{y}^{(j+1)} &= \operatorname{argmin}_{\mathbf{y}} \; g(\mathbf{y}) + \frac{\rho}{2} \left\| B\mathbf{y} - \left(-A\mathbf{x}^{(j+1)} + \mathbf{c} - \mathbf{u}^{(j)} \right) \right\|_2^2 \\
\mathbf{u}^{(j+1)} &= \mathbf{u}^{(j)} + A\mathbf{x}^{(j+1)} + B\mathbf{y}^{(j+1)} - \mathbf{c}
\end{aligned}$$

which we will refer to as the **x**, **y**, and **u**, steps respectively.

The feasibility conditions (see Sec. 3.3 [BPC+10]) for the ADMM problem are

$$\begin{aligned}
& A\mathbf{x}^* + B\mathbf{y}^* - \mathbf{c} = 0 \\
& 0 \in \partial f(\mathbf{x}^*) + \rho^{-1} A^T \mathbf{u}^* \\
& 0 \in \partial g(\mathbf{u}^*) + \rho^{-1} B^T \mathbf{u}^* \;,
\end{aligned}$$

where ∂ denotes the subdifferential operator. It can be shown that the last feasibility condition above is always satisfied by the solution of the **y** step. The primal and dual residuals [BPC+10]

$$\begin{aligned}
\mathbf{r} &= A\mathbf{x}^{(j+1)} + B\mathbf{y}^{(j+1)} - \mathbf{c} \\
\mathbf{s} &= \rho A^T B(\mathbf{y}^{(j+1)} - \mathbf{y}^{(j)}) \;,
\end{aligned}$$

which can be derived from the feasibility conditions, provide a convenient measure of convergence, and can be used to define algorithm stopping criteria. The **u** step can be written in terms of the primal residual as

$$\mathbf{u}^{(j+1)} = \mathbf{u}^{(j)} + \mathbf{r}^{(j+1)} \;.$$

It is often preferable to use normalized versions of these residuals [Woh17], obtained by dividing the definitions above by their corresponding normalization factors

$$\begin{aligned}
r_{\mathrm{n}} &= \max(\|A\mathbf{x}^{(j+1)}\|_2, \|B\mathbf{y}^{(j+1)}\|_2, \|\mathbf{c}\|_2) \\
s_{\mathrm{n}} &= \rho \|A^T \mathbf{u}^{(j+1)}\|_2 \;.
\end{aligned}$$

These residuals can also be used in a heuristic scheme [Woh17] for selecting the critical *penalty parameter* ρ.

SPORCO ADMM Classes

SPORCO provides a flexible set of classes for solving problems within the ADMM framework. All ADMM algorithms are derived from class `admm.admm.ADMM`, which provides much of the infrastructure required for solving a problem, so that the user need only override methods that define the constraint components A, B, and $\mathbf{c}$, and that compute the **x** and **y** steps. This infrastructure includes the computation of the primal and dual residuals, which are used as convergence measures on which termination of the iterations can be based.

These residuals are also used within the heuristic scheme, referred to above for, automatically setting the penalty parameter. This scheme is controlled by the `AutoRho` entry in the algorithm options dictionary object that is used to specify algorithm options and parameters. For example, to enable or disable it, set `opt['AutoRho', 'Enabled']` to `True` or `False` respectively, where `opt` is an instance of `admm.admm.ADMM.Options` or one of its derived classes. It should be emphasized that this method is not always successful, and can result in oscillations or divergence of the optimization. The scheme is enabled by default for classes for which it is expected to give reasonable performance, and disabled for those for which it is not, but these default settings should not be considered to be particularly reliable, and the user is advised to explicitly select whether the method is enabled to disabled.

Additional class attributes and methods can be defined to customize the calculation of diagnostic information, such as the functional value, at each iteration. The SPORCO documentation includes a detailed description of the required and optional methods to be overridden in defining a class for solving a specific optimization problem.

The `admm.admm` module also includes classes that are derived from `admm.admm.ADMM` to specialize to less general cases; for example, class `admm.admm.ADMMEqual` assumes that $A = I$, $B = -I$, and $\mathbf{c} = \mathbf{0}$, which is a very frequently occurring case, allowing derived classes to avoid overriding methods that specify the constraint. The most complex partial specialization is `admm.admm.ADMMTwoBlockCnstrnt`, which implements the commonly-occurring ADMM problem form with a block-structured **y** variable,

$$\begin{aligned}
&\operatorname{argmin}_{\mathbf{x},\mathbf{y}_0,\mathbf{y}_1} \; f(\mathbf{x}) + g_0(\mathbf{y}_0) + g_0(\mathbf{y}_1) \\
&\text{such that } \begin{pmatrix} A_0 \\ A_1 \end{pmatrix} \mathbf{x} - \begin{pmatrix} \mathbf{y}_0 \\ \mathbf{y}_1 \end{pmatrix} = \begin{pmatrix} \mathbf{c}_0 \\ \mathbf{c}_1 \end{pmatrix} \;,
\end{aligned}$$

for solving problems that have the form

$$\operatorname{argmin}_{\mathbf{x}} \; f(\mathbf{x}) + g_0(A_0\mathbf{x}) + g_1(A_1\mathbf{x})$$

prior to variable splitting. The block components of the **y** variable are concatenated into a single NumPy array, with access to the individual components provided by methods `block_sep0` and `block_sep1`.

Defining new classes derived from `admm.admm.ADMM` or one of its partial specializations provides complete flexibility in constructing a new ADMM algorithm, while reducing the amount of code that has to be written compared with implementing the entire ADMM algorithm from scratch. When a new ADMM algorithm is closely related to an existing algorithm, it is often

much easier to derived the new class from that of the existing algorithm, as described in the section *Extending SPORCO*.

Sparse Coding

Sparse coding in SPORCO is based on the Basis Pursuit DeNoising (BPDN) problem [CDS98]

$$\text{argmin}_X \ (1/2)\|DX - S\|_F^2 + \lambda \|X\|_1 \ ,$$

were D is the dictionary, S is the signal matrix, each column of which is a distinct signal, X is the sparse representation, and λ is the regularization parameter controlling the sparsity of the solution. BPDN is solved via the equivalent ADMM problem

$$\text{argmin}_X \ (1/2)\|DX - S\|_F^2 + \lambda \|Y\|_1 \quad \text{such that} \quad X = Y \ .$$

This algorithm is effective because the Y step can be solved in closed form, and is computationally relatively cheap. The main computational cost is in solving the X step, which involves solving the potentially-large linear system

$$(D^T D + \rho I)X = D^T S + \rho(Y - U) \ .$$

SPORCO solves this system efficiently by precomputing an LU factorization of $(D^T D + \rho I)$ which enables a rapid direct-method solution at every iteration (see Sec. 4.2.3 in [BPC+10]). In addition, if $(DD^T + \rho I)$ is smaller than $(D^T D + \rho I)$, the matrix inversion lemma is used to reduce the size of the system that is actually solved (see Sec. 4.2.4 in [BPC+10]).

The solution of the BPDN problem is implemented by class `admm.bpdn.BPDN`. A number of variations on this problem are supported by other classes in module `admm.bpdn`.

Dictionary Learning

Dictionary learning is based on the problem

$$\text{argmin}_{D,X} \ (1/2)\|DX - S\|_F^2 + \lambda \|X\|_1 \ \text{ s.t } \ \|\mathbf{d}_m\|_2 = 1 \ ,$$

which is solved by alternating between a sparse coding stage, as above, and a constrained dictionary update obtained by solving the problem

$$\text{argmin}_D (1/2)\|DX - S\|_2^2 \ \text{ s.t } \ \|\mathbf{d}_m\|_2 = 1 \ .$$

This approach is implemented by class `admm.bpdndl.DictLearn`. An unusual feature of this dictionary learning algorithm is the adoption from convolutional dictionary learning [BEL13] [Woh16d] [GCW17] of the very effective strategy of alternating between a single step of each of the sparse coding and dictionary update algorithms. To the best of this author's knowledge, this strategy has not previously been applied to standard (non-convolutional) dictionary learning.

Convolutional Sparse Coding

Convolutional sparse coding (CSC) is based on a convolutional form of BPDN, referred to as Convolutional BPDN (CBPDN) [Woh16d]

$$\text{argmin}_{\{\mathbf{x}_m\}} \ \frac{1}{2} \left\| \sum_m \mathbf{d}_m * \mathbf{x}_m - \mathbf{s} \right\|_2^2 + \lambda \sum_m \|\mathbf{x}_m\|_1 \ ,$$

which is implemented by class `admm.cbpdn.ConvBPDN`. Module `admm.cbpdn` also contains a number of other classes implementing variations on this basic form. As in the case of standard BPDN, the main computational cost of this algorithm is in solving the $\mathbf{x}$ step, which can be solved very efficiently by exploiting the Sherman-Morrison formula [Woh14]. SPORCO provides support for solving the basic form above, as well as a number of variants, including one with a gradient penalty, and two different approaches for solving a variant with a spatial mask W [HHW15][Woh16a]

$$\text{argmin}_{\{\mathbf{x}_m\}} \ \frac{1}{2} \left\| W \left(\sum_m \mathbf{d}_m * \mathbf{x}_m - \mathbf{s} \right) \right\|_2^2 + \lambda \sum_m \|\mathbf{x}_m\|_1 \ .$$

SPORCO also supports two different methods for convolutional sparse coding of multi-channel (e.g. color) images [Woh16b]. The one represents a multi-channel input with channels $\mathbf{s}_c$ with single-channel dictionary filters $\mathbf{d}_m$ and multi-channel coefficient maps $\mathbf{x}_{c,m}$,

$$\text{argmin}_{\{\mathbf{x}_{c,m}\}} \ \frac{1}{2} \sum_c \left\| \sum_m \mathbf{d}_m * \mathbf{x}_{c,m} - \mathbf{s}_c \right\|_2^2 + \lambda \sum_c \sum_m \|\mathbf{x}_{c,m}\|_1 \ ,$$

and the other uses multi-channel dictionary filters $\mathbf{d}_{c,m}$ and single-channel coefficient maps $\mathbf{x}_m$,

$$\text{argmin}_{\{\mathbf{x}_m\}} \ \frac{1}{2} \sum_c \left\| \sum_m \mathbf{d}_{c,m} * \mathbf{x}_m - \mathbf{s}_c \right\|_2^2 + \lambda \sum_m \|\mathbf{x}_m\|_1 \ .$$

In the former case the representation of each channel is completely independent unless they are coupled via an $\ell_{2,1}$ norm term [Woh16b], which is supported by class `admm.cbpdn.ConvBPDNJoint`.

An important issue that has received surprisingly little attention in the literature is the need to explicitly consider the representation of the smooth/low frequency image component when constructing convolutional sparse representations. If this component is not properly taken into account, convolutional sparse representations tend to give poor results. As briefly mentioned in [Woh16d] (Sec. I), the simplest approach is to lowpass filter the image to be represented, computing the sparse representation on the highpass residual. In this approach the lowpass component forms part of the complete image representation, and should, of course, be added to the reconstruction from the sparse representation in order to reconstruct the image being represented. SPORCO supports this separation of an image into lowpass/highpass components via the function `util.tikhonov_filter`, which computes the lowpass component of $\mathbf{s}$ as the solution of the problem

$$\text{argmin}_{\mathbf{x}} \ (1/2) \|\mathbf{x} - \mathbf{s}\|_2^2 + (\lambda/2) \sum_i \|G_i \mathbf{x}\|_2^2 \ ,$$

where G_i is an operator computing the derivative along axis i of the array represented as vector $\mathbf{x}$, and λ is a parameter controlling the amount of smoothing. In some cases it is not feasible to handle the lowpass component via such a pre-processing strategy, making it necessary to include the lowpass component in the CSC optimization problem itself. The simplest approach to doing so is to append an impulse filter to the dictionary and include a gradient regularization term on corresponding coefficient map in the functional [Woh16c] (Sec. 3). This approach is supported by class `admm.cbpdn.ConvBPDNGradReg`, the use of which is demonstrated in section *Removal of Impulse Noise via CSC*.

Convolutional Dictionary Learning

Convolutional dictionary learning is based on the problem

$$\text{argmin}_{\{\mathbf{d}_m\},\{\mathbf{x}_{k,m}\}} \ \frac{1}{2} \sum_k \left\| \sum_m \mathbf{d}_m * \mathbf{x}_{k,m} - \mathbf{s}_k \right\|_2^2 + \lambda \sum_k \sum_m \|\mathbf{x}_{k,m}\|_1$$
$$\text{s.t } \ \mathbf{d}_m \in C \ ,$$

where C is the feasible set, consisting of filters with unit norm and constrained support [Woh16d]. It is solved by alternating between a convolutional sparse coding stage, as described in the previous section, and a constrained dictionary update obtained by solving the problem

$$\operatorname{argmin}_{\{\mathbf{d}_m\}} \frac{1}{2} \sum_k \left\| \sum_m \mathbf{d}_m * \mathbf{x}_{k,m} - \mathbf{s}_k \right\|_2^2 \text{ s.t. } \mathbf{d}_m \in C \ .$$

This approach is implemented by class `ConvBPDNDictLearn` in module `admm.cbpdndl`. Dictionary learning with a spatial mask W,

$$\operatorname{argmin}_{\{\mathbf{d}_m\},\{\mathbf{x}_{k,m}\}} \frac{1}{2} \sum_k \left\| W \left(\sum_m \mathbf{d}_m * \mathbf{x}_{k,m} - \mathbf{s}_k \right) \right\|_2^2 + \lambda \sum_k \sum_m \|\mathbf{x}_{k,m}\|_1 \quad \text{s.t } \mathbf{d}_m \in C$$

is also supported by class `ConvBPDNMaskDcplDictLearn` in module `admm.cbpdndl`.

Convolutional Representations

SPORCO convolutional representations are stored within NumPy arrays of `dimN` + 3 dimensions, where `dimN` is the number of spatial/temporal dimensions in the data to be represented. This value defaults to 2 (i.e. images), but can be set to any other reasonable value, such as 1 (i.e. one-dimensional signals) or 3 (video or volumetric data). The roles of the axes in these multi-dimensional arrays are required to follow a fixed order: first spatial/temporal axes, then an axis for multiple channels (singleton in the case of single-channel data), then an axis for multiple input signals (singleton in the case of only one input signal), and finally the axis corresponding to the index of the filters in the dictionary.

Sparse Coding

For the convenience of the user, the `D` (dictionary) and `S` (signal) arrays provided to the convolutional sparse coding classes need not follow this strict format, but they are internally reshaped to this format for computational efficiency. This internal reshaping is largely transparent to the user, but must be taken into account when passing weighting arrays to optimization classes (e.g. option `L1Weight` for class `admm.cbpdn.ConvBPDN`). When performing the reshaping into internal array layout, it is necessary to infer the intended roles of the axes of the input arrays, which is performed by class `admm.cbpdn.ConvRepIndexing` (this class is expected to be moved to a different module in a future version of SPORCO). The inference rules, which are described in detail in the documentation for class `admm.cbpdn.ConvRepIndexing`, are relatively complex, depending on both the number of dimensions in the `D` and `S` arrays, and on parameters `dimK` and `dimN`.

Dictionary Update

The handling of convolutional representations by the dictionary update classes in module `admm.ccmod` are similar to those for sparse coding, the primary difference being the the dictionary update classes expect that the sparse representation inputs `X` are already in the standard layout as described above since they are usually obtained as the output of one of the sparse coding classes, and therefore already have the required layout. The inference of internal dimensions for these classes is handled by class `admm.ccmod.ConvRepIndexing` (which is also expected to be moved to a different module in a future version of SPORCO).

Problem Parameters

Most of the inverse problems supported by SPORCO have at least one problem parameter (e.g. regularization parameter λ in the BPDN and CBPDN problems) that determines the balance between the different terms in the functional to be minimized. Of these, the only problem that has a relatively reliable default value for its parameter is RPCA (see class `admm.rpca.RobustPCA`). Most of the classes implementing BPDN and CBPDN problems do have default values for regularization parameter λ, but these defaults should not be expected to provide even close to optimal performance for specific applications, and may be removed in future versions.

SPORCO does not support any statistical parameter estimation techniques such as GCV [GHW79] or SURE [Ste81], but the grid search function `util.grid_search` can be very helpful in selecting problem parameters when a suitable data set with ground truth is available. This function efficiently evaluates a user-specified performance measure, in parallel, over a single- or multi-dimensional grid sampling the parameter space. Usage of this function is illustrated in the example scripts `examples/stdsparse/demo_bpdn.py` and `examples/stdsparse/demo_bpdnjnt.py`, which "cheat" by evaluating performance by using the ground truth for the actual problem being solved. In a more realistic setting, one would optimize the parameters using the ground truth for a separate set of data with the same properties as those of the data for the test problem.

Installing SPORCO

The primary requirements for SPORCO are Python itself (version 2.7 or 3.x), and modules numpy, scipy, future, pyfftw, and matplotlib. Module numexpr is not required, but some functions will be faster if it is installed. If module mpldatacursor is installed, `plot.plot` and `plot.imview` will support the data cursor that it provides. Additional information on the requirements are provided in the installation instructions.

SPORCO is available on GitHub and can be installed via `pip`:

```
pip install sporco
```

SPORCO can also be installed from source, either from the development version from GitHub, or from a release source package downloaded from PyPI.

To install the development version from GitHub do

```
git clone https://github.com/bwohlberg/sporco.git
```

followed by

```
cd sporco
python setup.py build
python setup.py test
python setup.py install
```

The install command will usually have to be performed with root permissions, e.g. on Ubuntu Linux

```
sudo python setup.py install
```

The procedure for installing from a source package downloaded from PyPI is similar.

A summary of the most significant changes between SPORCO releases can be found in the `CHANGES.rst` file. It is strongly recommended to consult this summary when updating from a previous version.

SPORCO includes a large number of usage examples, some of which make use of a set of standard test images, which can be installed using the `sporco_get_images` script. To download these images from the root directory of the source distribution (i.e. prior to installation) do

```
bin/sporco_get_images --libdest
```

after setting the `PYTHONPATH` environment variable to point to the root directory of the source distribution; for example, in a `bash` shell

```
export PYTHONPATH=$PYTHONPATH:`pwd`
```

from the root directory of the package. To download the images as part of a package that has already been installed, do

```
sporco_get_images --libdest
```

which will usually have to be performed with root privileges.

Using SPORCO

The simplest way to use SPORCO is to make use of one of the many existing classes for solving problems that are already supported, but SPORCO is also designed to be easy to extend to solve custom problems, in some cases requiring only a few lines of additional code to extend an existing class to solve a new problem. This latter, more advanced usage is described in the section *Extending SPORCO*.

Detailed documentation is available. The distribution includes a large number of example scripts and a selection of Jupyter notebook demos, which can be viewed online via nbviewer, or run interactively via mybinder.

A Simple Usage Example

Each optimization algorithm is implemented as a separate class. Solving a problem is straightforward, as illustrated in the following example, which assumes that we wish to solve the BPDN problem

$$\operatorname{argmin}_{\mathbf{x}} (1/2)\|D\mathbf{x} - \mathbf{s}\|_F^2 + \lambda \|\mathbf{x}\|_1$$

for a given dictionary D and signal vector $\mathbf{s}$, represented by NumPy arrays `D` and `s` respectively. After importing the appropriate modules

```
import numpy as np
from sporco.admm import bpdn
```

we construct a synthetic problem consisting of a random dictionary and a test signal that is generated so that it has a very sparse representation, `x0`, on that dictionary

```
np.random.seed(12345)
D = np.random.randn(8, 16)
x0 = np.zeros((16, 1))
x0[[3,11]] = np.random.randn(2,1)
s = D.dot(x0)
```

Now we create an object representing the desired algorithm options

```
opt = bpdn.BPDN.Options({'Verbose' : True,
                         'MaxMainIter' : 500,
                         'RelStopTol' : 1e-6})
```

initialize the solver object

```
lmbda = 1e-2
b = bpdn.BPDN(D, s, lmbda, opt)
```

and call the `solve` method

```
x = b.solve()
```

leaving the result in NumPy array `x`. Since the optimizer objects retain algorithm state, calling `solve` again gives a warm start on an additional set of iterations for solving the same problem (e.g. if the first solve terminated because it reached the maximum number of iterations, but the desired solution accuracy was not reached).

Removal of Impulse Noise via CSC

We now consider a more detailed and realistic usage example, based on using CSC to remove impulse noise from a color image. First we need to import some modules, including `print_function` for Python 2/3 compatibility, NumPy, and a number of modules from SPORCO:

```
from __future__ import print_function

import numpy as np
from scipy.misc import imsave

from sporco import util
from sporco import plot
from sporco import metric
from sporco.admm import cbpdn
```

Boundary artifacts are handled by performing a symmetric extension on the image to be denoised and then cropping the result to the original image support. This approach is simpler than the boundary handling strategies described in [HHW15] and [Woh16a], and for many problems gives results of comparable quality. The functions defined here implement symmetric extension and cropping of images.

```
def pad(x, n=8):

  if x.ndim == 2:
      return np.pad(x, n, mode='symmetric')
  else:
      return np.pad(x, ((n, n), (n, n), (0, 0)),
                    mode='symmetric')


def crop(x, n=8):

  return x[n:-n, n:-n]
```

Now we load a reference image (see the discussion on the script for downloading standard test images in section *Installing SPORCO*), and corrupt it with 33% salt and pepper noise. (The call to `np.random.seed` ensures that the pseudo-random noise is reproducible.)

```
img = util.ExampleImages().image('standard',
      'monarch.png', zoom=0.5, scaled=True,
      idxexp=np.s_[:, 160:672])
np.random.seed(12345)
imgn = util.spnoise(img, 0.33)
```

We use a color dictionary, as described in [Woh16b]. The impulse denoising problem is solved by appending some additional filters to the learned dictionary `D0`, which is one of those distributed with SPORCO. The first of these additional components is a set of three impulse filters, one per color channel, that will represent the impulse noise, and the second is an identical set of impulse filters that will represent the low frequency image components when used together with a gradient penalty on the coefficient maps, as discussed below.

```
D0 = util.convdicts()['RGB:8x8x3x64']
Di = np.zeros(D0.shape[0:2] + (3, 3))
np.fill_diagonal(Di[0, 0], 1.0)
D = np.concatenate((Di, Di, D0), axis=3)
```

The problem is solved using class `ConvBPDNGradReg` in module `admm.cbpdn`, which implements the form of CBPDN with an additional gradient regularization term,

$$\operatorname{argmin}_{\{\mathbf{x}_m\}} \frac{1}{2} \left\| \sum_m \mathbf{d}_m * \mathbf{x}_m - \mathbf{s} \right\|_2^2 + \lambda \sum_m \|\mathbf{x}_m\|_1 + \frac{\mu}{2} \sum_i \sum_m \|G_i \mathbf{x}_m\|_2^2$$

where G_i is an operator computing the derivative along index i, as described in [Woh16c]. The regularization parameters for the ℓ_1 and gradient terms are `lmbda` and `mu` respectively. Setting correct weighting arrays for these regularization terms is critical to obtaining good performance. For the ℓ_1 norm, the weights on the filters that are intended to represent the impulse noise are tuned to an appropriate value for the impulse noise density (this value sets the relative cost of representing an image feature by one of the impulses or by one of the filters in the learned dictionary), the weights on the filters that are intended to represent low frequency components are set to zero (we only want them penalized by the gradient term), and the weights of the remaining filters are set to unity. For the gradient penalty, all weights are set to zero except for those corresponding to the filters intended to represent low frequency components, which are set to unity.

```
lmbda = 2.8e-2
mu = 3e-1
w1 = np.ones((1, 1, 1, 1, D.shape[-1]))
w1[..., 0:3] = 0.33
w1[..., 3:6] = 0.0
wg = np.zeros((D.shape[-1]))
wg[..., 3:6] = 1.0
opt = cbpdn.ConvBPDNGradReg.Options(
       {'Verbose': True, 'MaxMainIter': 100,
        'RelStopTol': 5e-3, 'AuxVarObj': False,
        'L1Weight': w1, 'GradWeight': wg})
```

Now we initialize the `cbpdn.ConvBPDNGradReg` object and call the `solve` method.

```
b = cbpdn.ConvBPDNGradReg(D, pad(imgn), lmbda, mu,
                          opt=opt, dimK=0)
X = b.solve()
```

The denoised estimate of the image is just the reconstruction from all coefficient maps except those that represent the impulse noise, which is why we subtract the slice of `X` corresponding the impulse noise representing filters from the result of `reconstruct`.

```
imgdp = b.reconstruct().squeeze() \
        - X[..., 0, 0:3].squeeze()
imgd = crop(imgdp)
```

Now we print the PSNR of the noisy and denoised images, and display the reference, noisy, and denoised images. These images are shown in Figures 3, 4, and 5 respectively.

```
print('%.3f dB   %.3f dB' % (sm.psnr(img, imgn),
      sm.psnr(img, imgd)))

fig = plot.figure(figsize=(21, 7))
plot.subplot(1,3,1)
plot.imview(img, fgrf=fig, title='Reference')
plot.subplot(1,3,2)
plot.imview(imgn, fgrf=fig, title='Noisy')
plot.subplot(1,3,3)
plot.imview(imgd, fgrf=fig, title='CSC Result')
fig.show()
```

Fig. 3: *Reference image*

Fig. 4: *Noisy image*

Finally, we save the low frequency image component estimate as an NPZ file, for use in a subsequent example.

```
imglp = X[..., 0, 3:6].squeeze()
np.savez('implslpc.npz', imglp=imglp)
```

Extending SPORCO

We illustrate the ease of extending or modifying existing algorithms in SPORCO by constructing an alternative approach to removing impulse noise via CSC. The previous method gave good results, but the weight on the filter representing the impulse noise is an additional parameter that has to be tuned. This parameter can be avoided by switching to an ℓ_1 data fidelity term instead of including dictionary filters to represent the impulse noise, as in the problem [Woh16c]

$$\operatorname{argmin}_{\{\mathbf{x}_m\}} \left\| \sum_m \mathbf{d}_m * \mathbf{x}_m - \mathbf{s} \right\|_1 + \lambda \sum_m \|\mathbf{x}_m\|_1 \;. \tag{2}$$

Ideally we would also include a gradient penalty term to assist in the representation of the low frequency image component. While this relatively straightforward, it is a bit more complex to

Fig. 5: *Denoised image (first method)*

implement, and is omitted from this example. Instead of including a representation of the low frequency image component within the optimization, we use the low frequency component estimated by the previous example, subtracting it from the signal passed to the CSC algorithm, and adding it back to the solution of this algorithm.

An algorithm for the problem in Equation (2) is not included in SPORCO, but there is an existing algorithm that can easily be adapted. CBPDN with mask decoupling, with mask array W,

$$\operatorname{argmin}_{\{\mathbf{x}_m\}} \frac{1}{2} \left\| W \left(\sum_m \mathbf{d}_m * \mathbf{x}_m - \mathbf{s} \right) \right\|_2^2 + \lambda \sum_m \|\mathbf{x}_m\|_1 \;, \qquad (3)$$

is solved via the ADMM problem

$$\begin{aligned} &\operatorname{argmin}_{\mathbf{x},\mathbf{y}_0,\mathbf{y}_1} (1/2)\|W\mathbf{y}_0\|_2^2 + \lambda \|\mathbf{y}_1\|_1 \\ &\text{such that } \begin{pmatrix} D \\ I \end{pmatrix} \mathbf{x} - \begin{pmatrix} \mathbf{y}_0 \\ \mathbf{y}_1 \end{pmatrix} = \begin{pmatrix} \mathbf{s} \\ \mathbf{0} \end{pmatrix} \;, \end{aligned} \qquad (4)$$

where $\mathbf{x} = \begin{pmatrix} \mathbf{x}_0^T & \mathbf{x}_1^T & \dots \end{pmatrix}^T$ and $D\mathbf{x} = \sum_m \mathbf{d}_m * \mathbf{x}_m$. We can express Equation (2) using the same variable splitting, as

$$\begin{aligned} &\operatorname{argmin}_{\mathbf{x},\mathbf{y}_0,\mathbf{y}_1} \|W\mathbf{y}_0\|_1 + \lambda \|\mathbf{y}_1\|_1 \\ &\text{such that } \begin{pmatrix} D \\ I \end{pmatrix} \mathbf{x} - \begin{pmatrix} \mathbf{y}_0 \\ \mathbf{y}_1 \end{pmatrix} = \begin{pmatrix} \mathbf{s} \\ \mathbf{0} \end{pmatrix} \;. \end{aligned} \qquad (5)$$

(We don't need the W for the immediate problem at hand, but there isn't a good reason for discarding it.) Since Equation (5) has no $f(\mathbf{x})$ term (see Equation (1)), and has the same constraint as Equation (4), the $\mathbf{x}$ and $\mathbf{u}$ steps for these two problems are the same. The $\mathbf{y}$ step for Equation (4) decomposes into the two independent subproblems

$$\begin{aligned} \mathbf{y}_0^{(j+1)} &= \operatorname{argmin}_{\mathbf{y}_0} \frac{1}{2} \|W\mathbf{y}_0\|_2^2 + \frac{\rho}{2} \left\| \mathbf{y}_0 - (D\mathbf{x}^{(j+1)} - \mathbf{s} + \mathbf{u}_0^{(j)}) \right\|_2^2 \\ \mathbf{y}_1^{(j+1)} &= \operatorname{argmin}_{\mathbf{y}_1} \lambda \|\mathbf{y}_1\|_1 + \frac{\rho}{2} \left\| \mathbf{y}_1 - (\mathbf{x}^{(j+1)} + \mathbf{u}_1^{(j)}) \right\|_2^2 . \end{aligned}$$

The only difference between the ADMM algorithms for Equations (4) and (5) is in the $\mathbf{y}_0$ subproblem, which becomes

$$\mathbf{y}_0^{(j+1)} = \operatorname{argmin}_{\mathbf{y}_0} \|W\mathbf{y}_0\|_1 + \frac{\rho}{2} \left\| \mathbf{y}_0 - (D\mathbf{x}^{(j+1)} - \mathbf{s} + \mathbf{u}_0^{(j)}) \right\|_2^2 .$$

Therefore, the only modifications we expect to make to the class implementing the problem in Equation (3) are changing the computation of the functional value, and part of the $\mathbf{y}$ step.

We turn now to the implementation for this example. The module import statements and definitions of functions `pad` and `crop` are the same as for the example in section *Removal of Impulse Noise via CSC*, and are not repeated here. Our main task is to modify `cbpdn.ConvBPDNMaskDcpl`, the class for solving the problem in Equation (3), to replace the ℓ_2 norm data fidelity term with an ℓ_1 norm. The $\mathbf{y}$ step of this class is

```
def ystep(self):
    AXU = self.AX + self.U
    Y0 = (self.rho*(self.block_sep0(AXU) - self.S)) \
         / (self.W**2 + self.rho)
    Y1 = sl.shrink1(self.block_sep1(AXU),
                    (self.lmbda/self.rho)*self.wl1)
    self.Y = self.block_cat(Y0, Y1)

    super(ConvBPDNMaskDcpl, self).ystep()
```

where the `Y0` and `Y1` blocks of `Y` respectively represent $\mathbf{y}_0$ and $\mathbf{y}_1$ in Equation (5). All we need do to change the data fidelity term to an ℓ_1 norm is to modify the calculation of `Y0` to be a soft thresholding instead of the calculation derived from the existing ℓ_2 norm. We also need to override method `obfn_g0` so that the functional values are calculated correctly, taking into account the change of the data fidelity term. We end up with a definition of our class solving Equation (2) consisting of only a few lines of additional code

```
class ConvRepL1L1(cbpdn.ConvBPDNMaskDcpl):

  def ystep(self):

    AXU = self.AX + self.U
    Y0 = sl.shrink1(self.block_sep0(AXU) - self.S,
                    (1.0/self.rho)*self.W)
    Y1 = sl.shrink1(self.block_sep1(AXU),
                    (self.lmbda/self.rho)*self.wl1)
    self.Y = self.block_cat(Y0, Y1)

    super(cbpdn.ConvBPDNMaskDcpl, self).ystep()


  def obfn_g0(self, Y0):

    return np.sum(np.abs(self.W *
                         self.obfn_g0var()))
```

To solve the impulse denoising problem we load the reference image and dictionary, and construct the test image as before. We also need to load the low frequency component saved by the previous example

```
imglp = np.load('implslpc.npz')['imglp']
```

Now we initialize an instance of our new class, solve, and reconstruct the denoised estimate

```
lmbda = 3.0
b = ConvRepL1L1(D, pad(imgn) - imglp, lmbda,
                opt=opt, dimK=0)
X = b.solve()
imgdp = b.reconstruct().squeeze() + imglp
imgd = crop(imgdp)
```

The resulting denoised image is displayed in Figure 6.

Support Functions and Classes

In addition to the main set of classes for solving inverse problems, SPORCO provides a number of supporting functions and classes, within the following modules:

Fig. 6: *Denoised image (second method)*

- `util`: Various utility functions and classes, including a parallel-processing grid search for parameter optimization, access to a set of pre-learned convolutional dictionaries, and access to a set of example images.
- `plot`: Functions for plotting graphs or 3D surfaces and visualizing images, providing simplified access to Matplotlib functionality.
- `linalg`: Linear algebra and related functions, including solvers for specific forms of linear system and filters for computing image gradients.
- `metric`: Image quality metrics including standard metrics such as MSE, SNR, and PSNR.
- `cdict`: A constrained dictionary class that constrains the allowed dict keys, and also initializes the dict with default content on instantiation. All of the inverse problem algorithm options classes are derived from this class.

Conclusion

SPORCO is an actively maintained and thoroughly documented open source Python package for computing with sparse representations. While the primary design goal is ease of use and flexibility with respect to extensions of the supported algorithms, it is also intended to be computationally efficient and able to solve at least medium-scale problems. Standard sparse representations are supported, but the main focus is on convolutional sparse representations, for which SPORCO provides a wider range of features than any other publicly available library. The set of ADMM classes on which the optimization algorithms are based is also potentially useful for a much broader range of convex optimization problems.

Acknowledgment

Development of SPORCO was supported by the U.S. Department of Energy through the LANL/LDRD Program.

References

[All92] Stefano Alliney. Digital filters as absolute norm regularizers. *IEEE Transactions on Signal Processing*, 40(6):1548–1562, June 1992. doi:10.1109/78.139258.

[BEL13] Hilton Bristow, Anders Eriksson, and Simon Lucey. Fast convolutional sparse coding. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 391–398, June 2013. doi:10.1109/CVPR.2013.57.

[BPC+10] Stephen Boyd, Neal Parikh, Eric Chu, Borja Peleato, and Jonathan Eckstein. Distributed optimization and statistical learning via the alternating direction method of multipliers. *Foundations and Trends in Machine Learning*, 3(1):1–122, 2010. doi:10.1561/2200000016.

[CCS10] Jian-Feng Cai, Emmanuel J. Candès, and Zuowei Shen. A singular value thresholding algorithm for matrix completion. *SIAM Journal on Optimization*, 20(4):1956–1982, 2010. doi:10.1137/080738970.

[CDS98] Scott Shaobing Chen, David L. Donoho, and Michael A. Saunders. Atomic decomposition by basis pursuit. *SIAM Journal on Scientific Computing*, 20(1):33–61, 1998. doi:10.1137/S1064827596304010.

[GCW17] Cristina Garcia-Cardona and Brendt Wohlberg. Subproblem coupling in convolutional dictionary learning. In *Proceedings of IEEE International Conference on Image Processing (ICIP)*, September 2017. Accepted for presentation.

[GHW79] Gene H. Golub, Michael Heath, and Grace Wahba. Generalized cross-validation as a method for choosing a good ridge parameter. *Technometrics*, 21(2):215–223, May 1979. doi:10.1080/00401706.1979.10489751.

[HHW15] Felix Heide, Wolfgang Heidrich, and Gordon Wetzstein. Fast and flexible convolutional sparse coding. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 5135–5143, 2015. doi:10.1109/CVPR.2015.7299149.

[LS99] Michael S. Lewicki and Terrence J. Sejnowski. Coding time-varying signals using sparse, shift-invariant representations. In *Advances in Neural Information Processing Systems*, volume 11, pages 730–736, 1999.

[MBP14] Julien Mairal, Francis Bach, and Jean Ponce. Sparse modeling for image and vision processing. *Foundations and Trends in Computer Graphics and Vision*, 8(2-3):85–283, 2014. doi:10.1561/0600000058.

[ROF92] Leonid I. Rudin, Stanley Osher, and Emad Fatemi. Nonlinear total variation based noise removal algorithms. *Physica D: Nonlinear Phenomena*, 60(1–4):259–268, 1992. doi:10.1016/0167-2789(92)90242-F.

[Ste81] Charles M. Stein. Estimation of the mean of a multivariate normal distribution. *The Annals of Statistics*, pages 1135–1151, November 1981.

[Woh14] Brendt Wohlberg. Efficient convolutional sparse coding. In *Proceedings of IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, pages 7173–7177, May 2014. doi:10.1109/ICASSP.2014.6854992.

[Woh16a] Brendt Wohlberg. Boundary handling for convolutional sparse representations. In *Proceedings of IEEE International Conference on Image Processing (ICIP)*, September 2016. doi:10.1109/ICIP.2016.7532675.

[Woh16b] Brendt Wohlberg. Convolutional sparse representation of color images. In *Proceedings of the IEEE Southwest Symposium on Image Analysis and Interpretation (SSIAI)*, pages 57–60, March 2016. doi:10.1109/SSIAI.2016.7459174.

[Woh16c] Brendt Wohlberg. Convolutional sparse representations as an image model for impulse noise restoration. In *Proceedings of the IEEE Image, Video, and Multidimensional Signal Processing Workshop (IVMSP)*, July 2016. doi:10.1109/IVMSPW.2016.7528229.

[Woh16d] Brendt Wohlberg. Efficient algorithms for convolutional sparse representations. *IEEE Transactions on Image Processing*, 25(1):301–315, January 2016. doi:10.1109/TIP.2015.2495260.

[Woh17] Brendt Wohlberg. ADMM penalty parameter selection by residual balancing, 2017. arXiv:1704.06209.

[ZKTF10] Matthew D. Zeiler, Dilip Krishnan, Graham W. Taylor, and Rob Fergus. Deconvolutional networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2528–2535, June 2010. doi:10.1109/cvpr.2010.5539957.

Software Transactional Memory in Pure Python

Dillon Niederhut‡*

Abstract—There has been a growing interest in programming models for concurrency. Strategies for dealing with shared data amongst parallel threads of execution include immutable (as in Erlang) and locked (as in Python) data structures. A third option exists, called transactional memory (as in Haskell), which includes thread-local journaling for operations on objects which are both mutable and globally shared. Here, we present TraM, a pure Python implementation of the TL2 algorithm for software transactional memory.

Index Terms—concurrency, threading, transactional memory

Introduction

Methods for sharing resources between multiple processes have been of academic interest for quite some time [Lamport_1978]. Recently, the need for handling coincident events in client-server interactions and the increasing scale of easily available data, especially in combination with the reduced momentum in increasing the clock speed of CPUs, have promoted discussions of concurrent software architecture [Lamport_et_al_1997]. In an ideal world, a computationally intensive task could split its work across the cores of a CPU in a way that does not require changes to the structure of the task itself. However, sharing work or any other kind of data in a concurrent system removes the guarantee that events occur in a strict linear order, which in turn disrupts the atomicity and consistency inherent to single threads of control. To see concretely how this might become a problem, consider the example below, in which several Python threads are attempting to increment a global counter[1] .

```
from threading import Thread
import time
import random

unsafe_number = 0

def unsafe_example():
    wait = random.random() / 10000
    global unsafe_number
    value = unsafe_number + 1
    time.sleep(wait)
    unsafe_number = value
```

In this particular example, we are forcing Python to behave asynchronously by inserting sleeping calls, which allow the interpreter to interrupt the execution of our "unsafe example", and give control in the interpreter to another thread. This creates the opportunity for inconsistent data, as the value of the "unsafe number" might have changed between the time a thread reads it, and the time a thread overwrites it. Thus, in the code below, we would expect to see consistent output of the number 10, but in practice will see something smaller, depending on the length of the wait and the architecture of the system running the example[2].

```
if __name__ == '__main__':

    thread_list = []
    for _ in range(10):
        thread = Thread(target=unsafe_example)
        thread_list.append(thread)
    for thread in thread_list:
        thread.start()
    for thread in thread_list:
        thread.join()

    print(unsafe_number)
```

```
$ python run_test.py
$ 5
$ python run_test.py
$ 4
$ python run_test.py
$ 6
```

Models for handling shared data in memory, specifically, have included restricting data structures into being immutable, or restricting access into those data structures with locking mechanisms. The former solution is disadvantaged by the CPU and memory costs of redundant data copies, while the latter suffers from deadlocks and leaky abstractions [Peyton_2002].

A third approach involves the use of local operation journals that are validated before any data is modified in place [Le_et_al_2016]. The strategy is similar to that used in database transactions and self-correcting file systems, where atomicity, consistency, and durability are enforced in part by maintaining a history of changes that have been made to the copy of data in memory but not yet persisted to the copy of data on disk. Within a concurrent software application, each thread of control can keep a similar history of proposed changes, and only modify the data object shared across all threads once that journal of changes has been approved. This strategy, where incremental changes in each thread are applied all at once to shared, mutable structures, is called software transactional memory (STM).

Software Transactional Memory

A specific implementation of STM, called Transactional Locking Version II (TL2) was recently proposed which avoids most of the copy-based and lock-based errors, along with the temporary unsafety characteristic of earlier STM algorithms, by versioning its data [Dice_et_al_2006]. Briefly, the algorithm works by setting

* Corresponding author: dillon.niederhut@gmail.com
‡ Enthought

up a local journal for each thread, where proposed modifications to shared data are kept. If no other thread has modified the original data structures during the time needed to calculate the proposed changes, those changes are swapped in memory for the old version of the internal data.

Under work loads that are predominantly read operations, TL2 outperforms lock-based strategies because it employs non-blocking reads. Under workloads that are dominated by writes to shared data structures, TL2 outperforms immutable strategies in that is possible to only copy pieces of a structure. The actual performance gain varies based on workload characteristics and number of CPUs, but a comparison against a coarse-grained POSIX mutex strategy shows gains of more than an order of magnitude; and, comparisons against previous implementations of STM are faster by constant factors roughly between 2 and 5 [Dice_et_al_2006].

The Python Implementation

The TraM package (available at https://github.com/deniederhut/tram) attempts to recreate the TL2 algorithm for transactional memory pythonically, and is not a one-for-one transliteration of the original Java implementation. The chief difference is that it does not use a global counter whose state is maintained by primitives in the language, but is instead using the system clock. This comes with the additional cost of making system calls, but prevents us from the necessity of building a concurrency strategy inside our concurrency strategy, since the clock state must be shared across all threads.

The algorithm starts by entering a retry loop, that will attempt to conduct the transaction a limited number of times before raising an exception. Ideally, this number is large enough that the retry limit would only be reached in the event of a system failure.

```
def transaction(self, *instance_list, write_action,
                read_action=None):
    """Conduct threadsafe operation"""
    if read_action is None:
        read_action = self.read
    retries = self.retries
    time.sleep(self.sleep) # for safety tests
    while retries:
        with self:
            read_list = read_action(instance_list)
            self.write(write_action(instance_list,
                       read_list))
            self.sequence_lock(instance_list)
            time.sleep(self.sleep) #
            try:
                self.validate()
                time.sleep(self.sleep) #
                self.commit()
            except ValidationError:
                pass
            except SuccessError:
                break
            finally:
                self.sequence_unlock(instance_list)
        self.decrement_retries()
```

It then creates two thread local logs. In our Python implementation, this occurs inside of a context manager.

```
def __enter__(self):
    """initialize local logs"""
    self.read_log = []
    self.write_log = []
```

It then reads local copies of data into its read log, and writes proposed changes into its write log. The algorithm itself is agnostic to what the reading and writing operations actually do.

```
def write(self, pair_list):
    """Write instance-value pairs to write log"""
    for instance, value in pair_list:
        self.write_log.append(
            Record(instance, value, time.time())
        )
```

This makes it easy to extend TraM's threadsafe objects by writing decorated, transactional methods.

```
def __iadd__(self, other):
    @atomic
    def fun(data, *args, **kwargs):
        return data + other
    do = Action()
    do.transaction(self, write_action=fun)
    return self
```

The algorithm then compares the version numbers of the original objects against the local data to see if they have been updated.

```
def validate(self):
    """Raise exception if any instance reads are
    no longer valid
    """
    for record in self.read_log:
        if record.instance.version > record.version:
            raise ValidationError
```

If not, a lock is acquired only long enough to accomplish two instructions: pointing the global data structure to the locally modified data; and, updating the version number.

```
def commit(self):
    """Commit write log to memory"""
    for record in self.write_log:
        record.instance.data = record.value
        record.instance.version = record.version
    raise SuccessError
```

If the read log is not validated, the entire operation is aborted and restarted. This suggests that the worst case scenario for TL2 is when several threads are attempting to write to a single object, as the invalidated threads will waste resources cycling through the retry loop.

Using a similar safety test, we can see that the TraM Int object correctly handles separate threads attempting to update its internal data, even when the actions performed by each thread cannot be guaranteed to be atomic themselves.

```
from tram import Int

def safe_example():
    global safe_number
    safe_number += 1

if __name__ == '__main__':

    thread_list = []
    for _ in range(10):
        thread = Thread(target=safe_example)
        thread_list.append(thread)
    for thread in thread_list:
        thread.start()
    for thread in thread_list:
        thread.join()

    print(safe_number)
```

```
$ python run_test.py
$ 10
$ python run_test.py
$ 10
$ python run_test.py
$ 10
```

Future Directions

This implementation of TL2 is specifically limited by implementation details of CPython, namely the global interpreter lock (GIL), which ensures that all actions are executed in a linear order given a single Python interpreter. Python's libraries for concurrent operations, including threading and the more modern async*s, are still executed within a single interpreter and are therefore under control of the GIL. Python's library for multiple OS threads, multiprocessing, will perform operations in parallel, but has a small number of data strucutures that are capable of being shared.

In our motivating example, we have tricked the interpreter into behaving as if this is not the case. While it is probably not a good idea to encourage software developers to play fast and loose with concurrency, there is a lot to be said for compartmentalizing the complexity of shared data into the shared data structures themselves. Concurrent programs are notoriously difficult to debug, and part of that complexity has to do with objects leaking their safety abstraction into the procedures trying to use them.

However, the work on creating a transactional branch of PyPy shows that there is some interest in concurrent applications for Python. PyPy's implementation of STM is currently based on a global processing queue, modeled after the threading module, with the transactional algorithms written in C [Meier_et_al_2014]. We hope that presenting an additional abstraction for composing transactional objects will encourage the exploration of STM specifically and concurrency generally, in the python community. Even if this does not occur, seeing the algorithm written out in a read-friendly language may serve as an education tool, especially as a starting point for creating a more clever version of the implementation itself.

As an algorithm for threadsafe objects, TL2 itself has two major limitations. The first, mentioned above, is that the algorithm depends on a version clock which is used to create a post-hoc, partial synchronization of procedures. In the original implementation, this is a shared, global, mutable counter, which is incremented every time any object is updated. In this implementation, it is the system clock, which is shared but no longer mutable by structures inside the algorithm. Both strategies have drawbacks.

The second major limitation is that attaching versions to objects works fine for *updating* data, but not for *deleting* the object. In garbage collected languages like Java and Python, we can rely on the runtime to keep track of whether those objects are still needed, and can remove them only after their last reference. Any implementation in a language which without automated memory management will need its own solution to the deletion of versioned data to avoid memory leaks.

References

[Dice_et_al_2006] Dice, D., Shalev, O., & Shavit, N. (2006). Transactional locking II. In *International Symposium on Distributed Computing* (pp. 194-208). Springer Berlin Heidelberg. https://doi.org/10.1007/11864219_14.

[Lamport_1978] Lamport, L. (1978). Time, clocks and the ordering of events in a distributed system. In *Communications of the ACM, 21.* (pp. 558-565).

[Le_et_al_2016] Le, M., Yates, R., & Fluet, M. (2016). Revisiting software transactional memory in Haskell. https://doi.org/10.1145/2976002.2976020. In *Proceedings of the 9th International Symposium on Haskell* (pp. 105-113). ACM.

[Meier_et_al_2014] Meier, R., & Rigo, A. (2014). A way forward in parallelising dynamic languages. In *Proceedings of the 9th International Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems PLE*. ACM. https://doi.org/10.1145/2633301.2633305.

[Peyton_2002] Peyton Jones, S. (2002). Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. In *Engineering theories of software construction* (pp. 47-96).

[Lamport_et_al_1997] Shavit, N. & Touitou, D. (1997). Software transactional memory. *Distributed Computing, 10.* (pp. 99-116). http://doi.org/10.1007/s004460050028.

1. Code has been modified from the original to avoid overfull hbox per the proceedings requirements

2. The order of magnitude for the wait time was chosen by experimentation to produce results between 3 and 7 on a 2.7GHz Intel Core i5.

BespON: Extensible config files with multiline strings, lossless round-tripping, and hex floats

Geoffrey M. Poore‡*

Abstract—BespON is a human-editable data format focused on expressive syntax, lossless round-tripping, and advanced features for scientific and technical tasks. Nested data structures can be represented concisely without multiple levels of either brackets or significant whitespace. The open-source Python implementation of BespON can modify data values while otherwise perfectly preserving config file layout, including comments. BespON also provides doc comments that can be preserved through arbitrary data modification. Additional features include integers (binary, octal, decimal, and hex), floats (decimal and hex, including Infinity and NaN), multiline string literals that only preserve indentation relative to delimiters, and an extensible design that can support user-defined data types.

Index Terms—configuration, data serialization, data interchange

Introduction

Many software projects need a human-editable data format, often for configuration purposes. For Python programs, INI-style files, JSON [JSON], and YAML [YAML] are popular choices. More recently, TOML [TOML] has become an alternative. While these different formats have their strengths, they also have some significant weaknesses when it comes to scientific and technical computing.

This paper introduces BespON [BespON], a new human-editable data format focused on scientific and technical features, and the `bespon` package for Python [pkg:bespon]. An overview of INI-style files, JSON, YAML, and TOML provides the motivation for BespON as well as context for its particular feature set.

Though this overview focuses on the features of each format, it also considers Python support for round-tripping—for loading data, potentially modifying it, and then saving it. While round-tripping will not lose data, it will typically lose comments and fail to preserve data ordering and formatting. Since comments and layout can be important in the context of configuration, some libraries provide special support for preserving them under round-tripping. That allows manual editing to be avoided while still minimizing the differences introduced by modifying data.

INI-style formats

Python's `configparser` module [py:configparser] supports a simple config format similar to Microsoft Windows INI files. For example:

```
[key]
subkey = value
```

Because all values are interpreted as strings, any typed values must be retrieved from the parsed data using getter functions that perform type conversion, introducing significant potential for ambiguity. Multiline string values that preserve newlines are permitted, but all indentation and all trailing whitespace (including a final newline) is stripped, so storing precise chunks of text for tasks such as templating is difficult. Another issue is that the format is not strictly specified, so that the Python 2 and Python 3 versions of the package are not fully compatible. This was a primary reason for `configparser` being rejected in PEP 518 [PEP518] as a possible format for storing Python build system requirements.

A more powerful and sophisticated INI-style format is provided by the `configobj` package [pkg:configobj]. All values are still strings as with `configparser`, but the package also provides a validator that allows the required format of a config file to be specified, along with type conversions for each data element. Multiline triple-quoted string literals are supported, though they are somewhat limited since they lack backslash-escapes and thus cannot contain triple-quoted strings or represent special characters using escape sequences. One particularly nice feature of `configobj` is round-trip support. Data can be loaded, modified, and saved, while preserving the order of values and retaining comments.

JSON

JSON [JSON] was designed as a lightweight interchange format. Its focus on a small number of common data types has enabled broad cross-language support, while its simple syntax is amenable to fast parsing. With JSON syntax, the earlier example data becomes:

```
{"key": {"subkey": "value"}}
```

Only dicts, lists, strings, numbers (floats), booleans, and null (`None`) are supported, so binary data and other unsupported types must be dealt with in an ad-hoc manner. As in the INI-style formats, dict keys can only be strings.

For configuration purposes, JSON has disadvantages. It lacks comments. Comments are not necessary in the common case of exchanging JSON data between machines, but in human-edited configuration data, they can be very useful. Similarly, JSON's brackets, braces, and quotation marks are sometimes criticized as verbose for human editing (for example, [PEP518]). For scientific

* *Corresponding author: gpoore@uu.edu*
‡ *Union University*

and technical tasks, JSON's lack of an integer type and of floating-point Infinity and NaN can be an issue. In fact, Python's standard library JSON implementation [py:json] explicitly does not comply with the JSON specification by adding extensions for integer, Infinity, and NaN support, and enabling these by default. Another drawback is that a string in JSON must be on a single line; there are no multiline string literals.

JSON's simplicity and limitations are an advantage when it comes to round-tripping data. Since there are no comments, a primary source of complexity is avoided altogether. Since there is only a single possible representation of most data if whitespace is ignored, lossless round-tripping primarily amounts to preserving indentation, line break locations, and the exact manner in which numerical values are represented.

YAML

YAML [YAML] was designed as a general serialization format. It can create a text-based representation of essentially arbitrary data structures, including some programming language-specific types. As a result, it supports integers (decimal, octal, hex), Infinity and NaN floating-point values, Base64-encoded binary data, and a variety of other data types. It also allows non-string dict keys. Its use of significant whitespace to avoid JSON's brackets and braces is reminiscent of Python's own avoidance of braces. In YAML syntax, the example data could be represented without quotation marks or braces:

```
key:
    subkey: value
```

The serialization capabilities of YAML can actually be a disadvantage by blurring the distinction between data and executable code. PyYAML [pkg:PyYAML], perhaps the most common Python YAML implementation, can execute arbitrary code during deserialization unless the special `yaml.safe_load()` function is used. For example, during YAML loading it is possible to run the default Python and include its `--help` output:

```
>>> yaml.load("""
help: !!python/object/apply:subprocess.check_output
      [['python', '--help']]
""")
```

YAML libraries in other languages can exhibit similar behavior by default; YAML deserialization was the source of a major security vulnerability in Ruby on Rails in 2013 [RoR].

YAML has been criticized for its complexity (for example, [PEP518] and [TOML]). This is partially due to the comparatively long YAML specification and the plethora of features it defines. For instance, most characters are allowed unquoted, but in a context-dependent manner. When YAML loads `"a#comment"`, it returns the string `a#comment`, but add a space before the `#`, and this becomes the string `a` followed by a line comment. Similarly, Python's `None` may be represented as `null`, `Null`, `NULL`, `~`, or as an empty value (for example, `"k:"` is identical to `"k: null"`). Some YAML issues were resolved in the transition from the version 1.1 specification (2005) to version 1.2 (2009). Among other things, the treatment of `Yes`, `No`, `On`, `Off`, and their lowercase and titlecase variants as boolean values was removed. However, since PyYAML is still based on the version 1.1 specification, the impact of version 1.2 for Python users has been minimal, at least until the `ruamel.yaml` package [pkg:ruamel.yaml] defaulted to the version 1.2 specification in 2016.

YAML does provide multiline string literals. For example:

```
key: |
    a multiline string
    in which line breaks are preserved
```

The multiline string begins on the line after the pipe `|`, and contains all text indented relative to the parent node (`key` in this case). This is a simple and efficient approach with minimal syntax for short snippets of text. It can become complex, however, if whitespace or indentation are important. Since the multiline string has no explicit ending delimiter, by default all trailing whitespace except for the final line break is stripped. This may be customized by using `|-` (remove all trailing whitespace, including the last line break) or `|+` (keep all trailing whitespace). Unfortunately, the `|+` case means that the string content depends on the relative positive of the next data element (or the end of the file, if the string is not followed by anything). Similarly, there are complications if all lines of the string contain leading whitespace or if the first line of the string is indented relative to subsequent lines. In such cases, the pipe must be followed immediately by an integer that specifies the indentation of the string relative to the parent node (`key` in the example).

All line breaks in multiline strings are normalized to line feeds (`\n`). Because backslash-escapes are not allowed in multiline strings, there is no way to wrap long lines, to specify other line break characters explicitly, or to use code points that are prohibited as literals in YAML files (for example, most control characters).

PyYAML provides no round-tripping support. The `ruamel.yaml` package does provide round-trip features. It can maintain comments, key ordering, and most styling so long as dict keys and list values are not deleted. While it supports modifying dict and list values, it does not provide built-in support for renaming dict keys.

TOML

TOML [TOML] is a more recent INI-inspired format. In TOML, the example data could be represented as:

```
[key]
subkey = "value"
```

TOML supports dicts (only with string keys), lists (only with all elements of the same type), strings, floats, integers, and booleans, plus date and time data. There are multiline string literals, both raw (delimited by `'''`) and with backslash-escapes (delimited by `"""`). Though these are very similar to Python multiline strings, they do have the difference that a line feed (`\n`) *immediately* following the opening delimiter is stripped, while it is retained otherwise, even if only preceded by a space.

String keys may be unquoted if they match the pattern for an ASCII identifier, and sections support what might be called "key paths." This allows nested data to be represented in a very compact manner without either brackets and braces or significant indentation. For example:

```
[key.subkey]
subsubkey = "value"
```

would be equivalent to the JSON

```
{"key": {"subkey": {"subsubkey": "value"}}}
```

TOML aims to be obvious, minimal, and more formally standardized than typical INI-style formats. In many ways it succeeds. It is used by Rust's Cargo package manager [Cargo] and in May 2016 was accepted as the future format for storing Python build system dependencies in PEP 518 [PEP518].

For scientific and technical tasks, TOML has some drawbacks. While there are integers, only decimal integers are supported. Decimal floats are supported, but with the notable exception of Infinity and NaN. Unlike YAML, multiline strings cannot be indented for clarity, because any indentation becomes part of the literal string content. There is no built-in support for any form of encoded binary data, and no extension mechanism for unsupported data types. These limitations may make sense in a format whose expanded acronym contains "obvious" and "minimal," but they do make TOML less appropriate for some projects.

In addition to these issues, some current features have the potential to be confusing. Inline dicts of the form

```
{"key" = "value"}
```

are supported, but they are not permitted to break over multiple lines. Meanwhile, inline lists *are* permitted to span multiple lines. When unquoted `true` appears as a dict key, it is a string, because only strings are allowed as keys. However, when it appears as a value, it is boolean true. Thus, `true = true` is a mapping of a string to a boolean.

Two of the more popular TOML implementations for Python are the `toml` package [pkg:toml] and the `pytoml` package [pkg:pytoml], which is being used in PEP 518. Currently, neither provides any round-trip support.

Introducing BespON

"BespON" is short for *Bespoken*, or custom-made, *Object Notation*. It originally grew out of a need for a config format with a `key=value` syntax that also offers excellent multiline string support. I am the creator of PythonTeX [PythonTeX], which allows executable code in Python and several other programming languages to be embedded within LaTeX documents. Future PythonTeX-related software will need a LaTeX-style `key=value` syntax for configuration. Because PythonTeX involves a significant amount of templating with Python code, a config format with multiline strings with obvious indentation would also be very useful. Later, BespON was influenced by some of my other software projects and by my work as a physics professor. This resulted in a focus on features related to scientific and technical computing.

- Integers, with binary, octal, and hexadecimal integers in addition to decimal integers.
- Full floating-point support including Infinity and NaN, and support for hexedecimal floating-point numbers.
- Multiline strings designed with templating and similar tasks in mind.
- A binary data type.
- Support for lossless round-tripping including comment preservation, at least when data is only modified.
- An extensible design that can allow for user-defined data types.

The `bespon` package for Python [pkg:bespon] was first released in April 2017, after over a year of development. It is used in all examples below. Like Python's `json` module [py:json], `bespon` provides `load()` and `loads()` functions for loading data from file-like objects or strings, and `dump()` and `dumps()` functions for dumping data to file-like objects or strings. `bespon` is compatible with Python 2.7 and 3.3+.

None and booleans

Python's `None` and boolean values are represented in BespON as `none`, `true`, and `false`. As in JSON and TOML, all keywords are lowercase. For example:

```
>>> import bespon
>>> bespon.loads("[none, true, false]")
[None, True, False]
```

Numbers

Integers

BespON supports binary, octal, decimal, and hexadecimal integers. Non-decimal integers use `0b`, `0o`, and `0x` base prefixes. Underscores are allowed between adjacent digits and after a base prefix, as in numbers in Python 3.6+ [PEP515]. For example:

```
>>> bespon.loads("[0b_1, 0o_7, 1_0, 0x_f]")
[1, 7, 10, 15]
```

Floats

Decimal and hexadecimal floating point numbers are supported, with underscores as in integers. Decimal numbers use `e` or `E` for the exponent, while hex use `p` or `P`, just as in Python float literals [py:stdtypes]. Infinity and NaN are represented as `inf` and `nan`.

```
>>> bespon.loads("[inf, nan, 2.3_4e1, 0x5_6.a_fp-8]")
[inf, nan, 23.4, 0.3386077880859375]
```

The support for hexadecimal floating-point numbers is particularly important in scientific and technical computing. Dumping and then loading a floating-point value in decimal form will typically involve small rounding errors [py:stdtypes]. The hex representation of a float allows the value to be represented exactly, since both the in-memory and serialized representation use base 2. This allows BespON files to be used in fully reproducible floating-point calculations. When the `bespon` package dumps data, the `hex_floats` keyword argument may be used to specify that all floats be saved in hex form.

Strings

BespON provides both inline strings, which do not preserve literal line breaks, and multiline strings, which do.

Raw and escaped versions of both are provided. Raw strings preserve all content exactly. Escaped strings allow code points to be represented with backslash-escapes. BespON supports Python-style `\xhh`, `\uhhhh`, and `\Uhhhhhhhh` escapes using hex digits `h`, as well as standard shorthand escapes like `\r` and `\n`. It also supports escapes of the form `\u{h...h}` containing 1 to 6 hex digits, as used in Rust [rs:tokens] and some other languages.

In addition, single-word identifier-style strings are allowed unquoted.

Inline strings

Raw inline strings are delimited by a single backtick `` ` ``, double backticks `` `` ``, triple backticks `` ``` ``, or a longer sequence that is a multiple of three. This syntax is inspired by [Markdown]; the case of single backticks is similar to Go's raw strings [Go]. A raw inline string may contain any sequence of backticks that is either longer or shorter than its delimiters. If the first non-space character in a raw string is a backtick, then the first space is stripped; similarly,

if the last non-space character is a backtick, then the last space is stripped. This allows, for example, the sequence `` ` ``` ` `` to represent the literal triple backticks ```, with no leading or trailing spaces.

The overall result is a raw string syntax that can enclose essentially arbitrary content while only requiring string modification (adding a leading or trailing space) in one edge case. Other common raw string syntaxes avoid any string modification, but either cannot enclose arbitrary content or require multiple different delimiting characters. For example, Python does not allow `r"\"`. It does allow `r"""\"""`, but this is not a complete string representing the backslash; rather, it is the start of a raw string that will contain the literal sequence `\"""` and requires `"""` as a closing delimiter [py:lexical]. Meanwhile, Rust represents the literal backslash as `r#"\"#` in raw string syntax, while literal `\#` would require `r##"\#"##` [rs:tokens].

Escaped inline strings are delimited by single quotation characters, either a single quote `'` or double quote `"`. These end at the first unescaped delimiting character. Escaped inline strings may also be delimited by triple quotation mark sequences `'''` or `"""`, or longer sequences that are a multiple of three. In these cases, any shorter or longer sequence of the delimiting character is allowed unescaped. This is similar to the raw string case, but with backslash-escapes.

Inline strings may be wrapped over multiple lines, in a manner similar to YAML. This allows BespON data containing long, single-line strings to be embedded within a LaTeX, Markdown, or other document without requiring either lines longer than 80 characters or the use of multiline strings with newline escapes. When an inline string is wrapped over multiple line, each line break is replaced with a space unless it is preceded by a code point with the Unicode `White_Space` property [UAX44], in which case it is stripped. For example:

```
>>> bespon.loads("""
'inline value
 that wraps'
""")
'inline value that wraps'
```

When an inline string is wrapped, the second line and all subsequent lines must have the same indentation.

Multiline strings

Multiline strings also come in raw and escaped forms. Syntax is influenced by heredocs in shells and languages like Ruby [rb:literals]. The content of a multiline string begins on the line *after* the opening delimiter, and ends on the line *before* the closing delimiter. All line breaks are preserved as literal line feeds (`\n`); even if BespON data is loaded from a file using Windows line endings `\r\n`, newlines are always normalized to `\n`. The opening delimiter consists of a pipe `|` followed immediately by a sequence of single quotes `'`, double quotes `"`, or backticks `` ` `` whose length is a multiple of three. Any longer or shorter sequence of quote/backtick characters is allowed to appear literally within the string without escaping. The quote/backtick determines whether backslash-escapes are enabled, following the rules for inline strings. The closing delimiter is the same as the opening delimiter with a slash `/` appended to the end. This enables opening and closing delimiters to be distinguished easily even in the absence of syntax highlighting, which is convenient when working with long multiline strings.

In a multiline string, total indentation is not preserved. Rather, indentation is only kept relative to the delimiters. For example:

```
>>> bespon.loads("""
  |'''
   first line
    second line
  |'''/
""")
' first line\n  second line\n'
```

This allows the overall multiline string to be indented for clarity, without the indentation becoming part of the literal string content.

Unquoted strings

BespON also allows unquoted strings. By default, only ASCII identifier-style strings are allowed. These must match the regular expression:

```
_*[A-Za-z][0-9A-Z_a-z]*
```

There is the additional restriction that no unquoted string may match a keyword (`none`, `true`, `false`, `inf`, `nan`) or related reserved word when lowercased. This prevents an unintentional miscapitalization like `FALSE` from becoming a string and then yielding true in a boolean test.

Unquoted strings that match a Unicode identifier pattern essentially the same as that in Python 3.0+ [PEP3131] may optionally be enabled. These are not used by default because they introduce potential usability and security issues. For instance, boolean false is represented as `false`. When unquoted Unicode identifier-style strings are enabled, the final `e` could be replaced with the lookalike code point `\u0435`, CYRILLIC SMALL LETTER IE. This would represent a string rather than a boolean, and any boolean tests would return true since the string is not empty.

Lists

Lists are supported using an indentation-based syntax similar to YAML as well as a bracket-delimited inline syntax like JSON or TOML.

In an indentation-style list, each list element begins with an asterisk `*` followed by the element content. For example:

```
>>> bespon.loads("""
* first
* second
* third
""")
['first', 'second', 'third']
```

Any indentation before or after the asterisk may use spaces or tabs, although spaces are preferred. In determining indentation levels and comparing indentation levels, a tab is never treated as identical to some number of spaces. An object that is indented relative to its parent object must share its parent object's indentation exactly. This guarantees that in the event that tabs and spaces are mixed, relative indentation will always be preserved.

In an inline list, the list is delimited by square brackets `[]`, and list elements are separated by commas. A comma is permitted after the last list element (dangling comma), unlike JSON:

```
>>> bespon.loads("[first, second, third,]")
['first', 'second', 'third']
```

An inline list may span multiple lines, as long as everything it contains and the closing bracket are indented at least as much as

the line on which the list begins. When inline lists are nested, the required indentation for all of the lists is simply that of the outermost list.

Dicts

Dicts also come in an indentation-based form similar to YAML as well as a brace-delimited inline syntax like JSON or TOML.

In an indentation-style list, keys and values are separated by an equals sign, as in INI-style formats and TOML. For example:

```
>>> bespon.loads("""
key =
    subkey = value
""")
{'key': {'subkey': 'value'}}
```

The rules for indentation are the same as for lists. A dict value that is a string or collection may span multiple lines, but it must always have at least as much indentation as its key if it starts on the same line as the key, or more indentation if it starts on a line after the key. This may be demonstrated with a multiline string:

```
>>> bespon.loads("""
key = |```
   first line
    second line
  |```/
""")
{'key': ' first line\n  second line\n'}
```

Because the multiline string starts on the same line as `key`, the opening and closing delimiters are not required to have the same indentation, and the indentation of the string content is relative to the closing deliter.

In an inline dict, the dict is delimited by curly braces `{}`, and key-value pairs are separated by commas:

```
>>> bespon.loads("""
{key = {subkey = value}}
""")
{'key': {'subkey': 'value'}}
```

As with inline lists, a dangling comma is permitted, as is spanning multiple lines so long as all content is indented at least as much as the line on which the dict begins. When inline dicts are nested, the required indentation for all of the dicts is simply that of the outermost dict.

Dicts support `none`, `true`, `false`, integers, and strings as keys. Floats are not supported as keys by default, since this could produce unexpected results due to rounding.

Key paths and sections

The indentation-based syntax for dicts involves increasing levels of indentation, while the inline syntax involves accumulating layers of braces. BespON provides a key-path syntax that allows this to be avoided in some cases. A nested dict can be created with a series of unquoted, period-separated keys. For example:

```
>>> bespon.loads("""
key.subkey.subsubkey = value
""")
{'key': {'subkey': {'subsubkey': 'value'}}}
```

Key path are scoped, so that once the indentation or brace level of the top of the key path is closed, no dicts created by the key path can be modified. Consider a nested dict three levels deep, with the lowest level accessed via key paths:

```
>>> bespon.loads("""
key =
    subkey.a = value1
    subkey.b = value2
""")
{'key': {'subkey': {'a': 'value1', 'b': 'value2'}}}
```

Key paths starting with `subkey` can be used multiple times at the indentation level where `subkey` is first used. Using `subkey.c` at this level would be valid. However, returning to the indentation level of `key` and attempting to use `key.subkey.c` would result in a scope error. Scoping ensures that all data defined via key paths with common nodes remains relatively localized.

Key paths can also be used in sections similar to INI-style formats and TOML. A section consists of a pipe followed immediately by three equals signs (or a longer series that is a multiple of three), followed by a key path. Everything until the next section definition will be placed under the section key path. For example:

```
>>> bespon.loads("""
|=== key.subkey
subsubkey = value
""")
{'key': {'subkey': {'subsubkey': 'value'}}}
```

This allows both indentation and layers of braces to be avoided, while not requiring the constant repetition of the complete path to the data that is being defined (`key.subkey` in this case).

Instead of ending a section by starting a new section, it is also possible to return to the top level of the data structure using an end delimiter of the form `|===/` (with the same number of equals signs as the opening section delimiter).

Tags

All of the data types discussed so far are implicitly typed; there is no explicit type declaration. BespON provides a tag syntax that allows for explicit typing and some other features. This may be illustrated with the `bytes` type, which can be applied to strings to create byte strings (Python `bytes`):

```
>>> bespon.loads("""
(bytes)> "A string in binary"
""")
b'A string in binary'
```

Similarly, there is a `base16` type and a `base64` type:

```
>>> bespon.loads("""
(base16)> "01 89 ab cd ef"
""")
b'\x01\x89\xab\xcd\xef'
>>> bespon.loads("""
(base64)> "U29tZSBCYXNlNjQgdGV4dA=="
""")
b'Some Base64 text'
```

When applied to strings, tags also support keyword arguments `indent` and `newline`. `indent` is used to specify a combination of spaces and tabs by which all lines in a string should be indented to produce the final string. `newline` takes any code point sequence considered a newline in the Unicode standard [UnicodeNL], or the empty string, and replaces all literal line breaks with the specified sequence. This simplifies the use of literal newlines other than the default line feed (`\n`). When `newline` is applied to a byte string, only newline sequences in the ASCII range are permitted.

```
>>> bespon.loads(r"""
(bytes, indent=' ', newline='\r\n')>
|'''
A string in binary
with a break
|'''/
""")
b' A string in binary\r\n with a break\r\n'
```

Aliases and inheritance

For configuration purposes, it would be convenient to have some form of inheritance, so that settings do not need to be duplicated in multiple dicts. The tag `label` keyword argument allows lists, list elements, dicts, and dict values to be labeled. Then they can be referenced later using aliases, which consist of a dollar sign `$` followed by the label name. Aliases form the basis for inheritance.

Dicts support two keywords for inheritance. `init` is used to specify one or more dicts with which to initialize a new dict. The keys supplied by these dicts must not be overwritten by the keys put into the new dict directly. Meanwhile, `default` is used to specify one or more dicts whose keys are added to the new dict after `init` and after values that are added directly. `default` keys are only added if they do not exist; they are fallback values.

```
>>> d = bespon.loads("""
initial =
    (dict, label=init)>
    first = a
default =
    (dict, label=def)>
    last = z
    k = default_v
settings =
    (dict, init=$init, default=$def)>
    k = v
""")
>>> d['settings']
{'first': 'a', 'k': 'v', 'last': 'z'}
```

If there multiple values for `init` or `default`, these could be provided in an inline list of aliases:

```
[$alias1, $alias2, ...]
```

In similar manner, `init` can be used to specify initial elements in a list, and `extend` to add elements at the end. Other features that make use of aliases are under development.

Immutability, confusability, and other considerations

BespON and the `bespon` package contain several features designed to enhance usability and prevent confusion.

Nested collections more than 100 levels deep are prohibited by default. In such cases, the `bespon` package raises a nesting depth error. This reduces the potential for runaway parsing.

BespON requires that dict keys be unique; keys are never overwritten. Similarly, there is no way to set and then modify list elements. In contrast, the JSON specification only specifies that keys "SHOULD be unique" [JSON]. Python's JSON module [py:json] allows duplicate keys, with later keys overwriting earlier ones. Although YAML [YAML] specifies that keys are unique, in practice PyYaml [pkg:PyYAML] allows duplicate keys, with later keys overwriting earlier ones. TOML [TOML] also specifies unique keys, and this is enforced by the `toml` [pkg:toml] and `pytoml` [pkg:pytoml] packages.

When the last line of an inline or unquoted string contains one or more Unicode code points with `Bidi_Class` R or AL (right-to-left languages) [UAX9], by default no other data objects or comments are allowed on the line on which the string ends. This prevents a right-to-left code point from interacting with following code points to produce ambiguous visual layout as a result of the Unicode bidirectional algorithm [UAX9] that is implemented in much text editing software. Consider an indentation-based dict mapping Hebrew letters to integers (valid BespON):

```
"א" =
    1
"ב" =
    2
```

There is no ambiguity in that case. Now consider the same data, but represented with an inline dict (still valid BespON):

```
{"\u05D0" = 1, "\u05D1" = 2}
```

There is still no ambiguity, but the meaning is less clear due to the Unicode escapes. If the literal letters are substituted, this is the rendering in most text editors (now invalid BespON):

```
{"2 = "ב" ,1 = "א}
```

Because the quotation marks, integers, comma, and equals signs have no strong left-to-right directionality, everything after the first quotation mark until the final curly brace is visually laid out from right to left. When the data is loaded, though, it will produce the correct mapping, since loading depends on the logical order of the code points rather than their visual rendering. By default, BespON prevents the potential for confusion as a result of this logical-visual mismatch, by prohibiting data objects or comments from immediately following an inline or unquoted string with one or more right-to-left code points in its last line. For the same reason, code points with the property `Bidi_Control` [UAX9] are prohibited from appearing literally in BespON data; they can only be produced via backslash-escapes.

Round-tripping

BespON has been designed with round-tripping in mind. Currently, the `bespon` package supports replacing keys and values in data. For example:

```
>>> ast = bespon.loads_roundtrip_ast("""
key.subkey.first = 123   # Comment
key.subkey.second = 0b1101
key.subkey.third = `literal \string`
""")
>>> ast.replace_key(['key', 'subkey'], 'sk')
>>> ast.replace_val(['key', 'sk', 'second'], 7)
>>> ast.replace_val(['key', 'sk', 'third'],
                    '\\another \\literal')
>>> ast.replace_key(['key', 'sk', 'third'], 'fourth')
>>> print(ast.dumps())

key.sk.first = 123   # Comment
key.sk.second = 0b111
key.sk.fourth = `\another \literal`
```

This illustrates several features of the round-trip capabilities.

- Comments, layout, and key ordering are preserved exactly.
- Key renaming works even with key paths, when a given key name appears in multiple locations.
- When a number is modified, the new value is expressed in the same base as the old value by default.
- When a quoted string is modified, the new value is quoted in the same style as the old value (at least when practical).

- As soon as a key is modified, the new key must be used for further modifications. The old key is invalid.

In the future, the `bespon` package will add additional round-trip capabilities beyond replacing keys and values. One of the challenges in round-tripping data is dealing with comments. BespON supports standard line comments of the form `#comment`. While these can survive round-tripping when data is added or deleted, dealing with them in those cases is difficult, because line comments are not uniquely associated with individual data objects. To provide an alternative, BespON defines a doc comment that is uniquely associated with individual data objects. Each data object may have at most a single doc comment. The syntax is inspired by string and section syntax, involving three hash symbols (or a multiple of three). Both inline and multiline doc comments are defined, and must come immediately before the data with which they are associated (or immediately before its tag, for tagged data):

```
key1 = ### inline doc comment for value 1 ###
       value1
key2 = |###
       multiline doc comment

       for value2
       |###/
       value2
```

Because doc comments are uniquely associated with individual data elements, they will make possible essentially arbitrary manipulation of data while retaining all relevant comments.

Performance

Since the beginning, performance has been a concern for BespON. The `bespon` package is pure Python. YAML's history suggested that this could be a significant obstacle to performance. PyYAML [pkg:PyYAML] can be much slower than Python's `json` module [py:json] for loading equivalent data, in part because the JSON module is implemented in C while the default PyYAML is pure Python. PyYAML can be distributed with LibYAML [LibYAML], a C implementation of YAML 1.1, which provides a significant performance improvement.

So far, `bespon` performance is promising. The package uses `__slots__` and avoids global variables extensively, but otherwise optimizations are purely algorithmic. In spite of this, under CPython it can be only about 50% slower than PyYAML with LibYAML. Under PyPy [PyPy], the alternative Python implementation with a just-in-time (JIT) compiler, `bespon` can be within an order of magnitude of `json`'s CPython speed.

Figure 1 shows an example of performance in loading data. This was generated with the BespON Python benchmarking code [bespon:benchmark]. A sample BespON data set was assembled using the template below (whitespace reformatted to fit column width), substituting the template field `{num}` for integers in `range(1000)` and then concatenating the results.

```
key{num} =
  first_subkey{num} =
    "Some text that goes on for a while {num}"
  second_subkey{num} =
    "Some more text that also goes on and on {num}"
  third_subkey{num} =
    * "first list item {num}"
    * "second list item {num}"
    * "third list item {num}"
```

Analogous data sets were generated for JSON, YAML, and TOML, using the closest available syntax. Python's `json` module and the PyYAML, `toml`, `pytoml`, and `bespon` packages were then used to load their corresponding data from strings 10 times. Load times were measured with Python's `timeit` module [py:timeit], and the minimum time for each package was recorded and plotted in the figure.

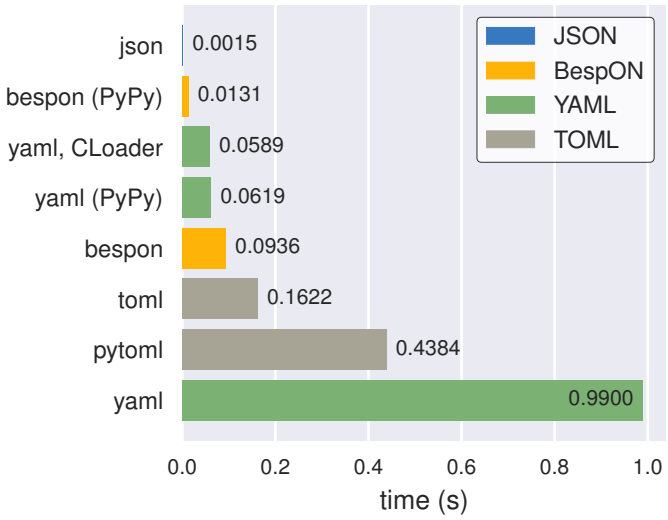


Fig. 1: *Performance of Python's `json` module and the PyYAML, `toml`, `pytoml`, and `bespon` packages in loading sample data. All tests were performed under Ubuntu 16.04. All tests used Anaconda Python 3.6.1 (64-bit) except those designated with "PyPy," which used PyPy3.5 5.7.1 (64-bit). PyYAML was tested with its C library implementation (CLoader) when available.*

An extended example

All examples shown so far have been short snippets loaded from Python strings using `bespon.loads()`. Any of those examples could instead have been saved in a text file, say `data.bespon`, and loaded as

```
with open('data.bespon', encoding='utf8') as f:
    data = bespon.load(f)
```

A longer example of a BespON file that could be loaded in this manner is shown below. It illustrates most BespON features.

```
# Line comments can be round-tripped if data
# elements are only modified, not added or removed.

### This doc comment can always be round-tripped.###
# Only one doc comment is allowed per data element.
# The doc comment above belongs to the key below.
"key (\x5C escapes)" = 'value (\u{5C} escapes)'

`key (no \ escapes)` = ``value (no `\` escapes)``

# Unquoted ASCII identifier-style strings.
unquoted_key = unquoted_value

# Trailing commas are fine.
inline_dict = {key1 = value1, key2 = value2,}

# Decimal, hex, octal, and binary integers.
inline_list_of_ints = [1, 0x12, 0o755, 0b1010]

list_of_floats =
    * 1.2e3
    * -inf      # Infinity and NaN are supported.
    * 0x4.3p2   # Hex floats to avoid rounding.
```

```
wrapped_string = """String with no whitespace
    lines, with line breaks converted to spaces,
    and "quotes" allowed by delimiters."""

multiline_raw_string = |```
      Linebreaks are kept (as '\n') and leading
      indentation is preserved relative to
      delimiters (which are on lines by themselves).
    |```/

multiline_escaped_string = |"""
    The same idea as the raw multiline string,
    but with backslash-escapes.
    |"""/

typed_string = (bytes)> "byte string"

# Key-path style; same as "key1 = {key2 = true}"
key1.key2 = true

# Same as "section = {subsection = {key = value}}"
|=== section.subsection
key = value
|===/  # Back to root level.  Can be omitted
       # if sections never return to root.
```

Conclusion

BespON and the `bespon` package remain under development.

The `bespon` package is largely complete as far as loading and dumping data are concerned. The standard, default data types discussed above are fully supported, and it is already possible to enable a limited selection of optional types.

The primary focus of future `bespon` development will be on improving round-tripping capabilities. Eventually, it will also be possible to enable optional user-defined data types with the tag syntax.

BespON as a configuration format will primarily be refined in the future through the creation of a more formal specification. The Python implementation is written in such a way that a significant portion of the grammar already exists in the form of Python template strings, from which it is converted into functions and regular expressions. A more formal specification will bring the possibility of implementations in additional languages.

Working with BespON will also be improved through additional revision of the programming language-agnostic test suite [bespon:test] and the syntax highlighting extension for Microsoft Visual Studio Code [bespon:vscode]. The language-agnostic test suite is a set of BespON data files containing hundreds of snippets of BespON that is designed to test implementations for conformance. It is used for testing the Python implementation before each release. The VS Code syntax highlighting extension provides a TextMate grammar [TextMate] for BespON, so it can provide a basis for BespON support in other text editors in the future.

References

[BespON] G. Poore. "BespON – Bespoken Object Notation," https://bespon.org/.

[bespon:benchmark] G. Poore. "Benchmark BespON in Python," https://github.com/bespon/bespon_python_benchmark.

[bespon:test] G. Poore. "Language-agnostic tests for BespON," https://github.com/bespon/bespon_tests.

[bespon:vscode] G. Poore. "BespON syntax highlighting for VS Code," https://github.com/bespon/bespon_vscode.

[Cargo] "CARGO: packages for Rust," https://crates.io/.

[Go] "The Go Programming Language Specification," November 18, 2016, https://golang.org/ref/spec.

[JSON] T. Bray. "The JavaScript Object Notation (JSON) Data Interchange Format," https://tools.ietf.org/html/rfc7159.

[LibYAML] "LibYAML," http://pyyaml.org/wiki/LibYAML.

[Markdown] J. Gruber. "Markdown: Syntax," https://daringfireball.net/projects/markdown/syntax.

[PEP515] G. Brandl, S. Storchaka. "PEP 515 -- Underscores in Numeric Literals," https://www.python.org/dev/peps/pep-0515/.

[PEP518] B. Cannon, N. Smith, D. Stufft. "PEP 518 -- Specifying Minimum Build System Requirements for Python Projects," https://www.python.org/dev/peps/pep-0518/.

[PEP3131] M. von Löwis. "PEP 3131 -- Supporting Non-ASCII Identifiers," https://www.python.org/dev/peps/pep-3131/.

[pkg:bespon] G. Poore, "bespon package for Python," https://github.com/gpoore/bespon_py.

[pkg:configobj] M. Foord, N. Larosa, R. Dennis, E. Courtwright. "Welcome to configobj's documentation!" http://configobj.readthedocs.io/en/latest/index.html.

[pkg:pytoml] "pytoml," https://github.com/avakar/pytoml.

[pkg:PyYAML] "PyYAML Documentation," http://pyyaml.org/wiki/PyYAMLDocumentation.

[pkg:ruamel.yaml] A. van der Neut. "ruamel.yaml," http://yaml.readthedocs.io/en/latest/index.html.

[pkg:toml] "TOML: Python module which parses and emits TOML," https://github.com/uiri/toml.

[PythonTeX] G. Poore. "PythonTeX: reproducible documents with LaTeX, Python, and more," *Computational Science & Discovery* 8 (2015) 014010, http://stacks.iop.org/1749-4699/8/i=1/a=014010.

[py:configparser] Python Software Foundation. "configparser — Configuration file parser", Apr 09, 2017, https://docs.python.org/3.6/library/configparser.html.

[py:json] Python Software Foundation. "json — JSON encoder and decoder," May 27, 2017, https://docs.python.org/3/library/json.html.

[py:lexical] Python Software Foundation. "Lexical analysis," Mar 26, 2017, https://docs.python.org/3/reference/lexical_analysis.html.

[py:stdtypes] Python Software Foundation. "Built-in Types," May 16, 2017, https://docs.python.org/3/library/stdtypes.html.

[py:timeit] Python Software Foundation. "`timeit` — Measure execution time of small code snippets," Mar 26, 2017, https://docs.python.org/3/library/timeit.html.

[PyPy] "Welcome to PyPy," http://pypy.org/.

[rb:literals] "Literals," https://ruby-doc.org/core-2.4.1/doc/syntax/literals_rdoc.html.

[RoR] A. Patterson. "Multiple vulnerabilities in parameter parsing in Action Pack (CVE-2013-0156)," https://groups.google.com/forum/#!topic/rubyonrails-security/61bkgvnSGTQ/discussion.

[rs:tokens] The Rust Project Developers. "Tokens," https://doc.rust-lang.org/reference/tokens.html.

[TextMate] MacroMates Ltd. "Language Grammars," https://manual.macromates.com/en/language_grammars.

[TOML] T. Preston-Werner. "TOML: Tom's Obvious, Minimal Language, v0.4.0," https://github.com/toml-lang/toml/.

[UAX9] M. Davis, A. Lanin, and A. Glass. "Unicode Standard Annex #9: UNICODE BIDIRECTIONAL ALGORITHM," http://unicode.org/reports/tr9/.

[UAX44] Unicode, Inc., ed. M. Davis, L. Iancu, and K. Whistler. "Unicode Standard Annex #44: UNICODE CHARACTER DATABASE," http://unicode.org/reports/tr44/.

[UnicodeNL] The Unicode Consortium. *The Unicode Standard, Version 9.0.0*, chapter 5.8, "Newline Guidelines," http://www.unicode.org/versions/Unicode9.0.0/.

[YAML] O. Ben-Kiki, C. Evans, I. döt Net. "YAML Ain't Markup Language (YAML) Version 1.2, 3rd Edition, Patched at 2009-10-01," http://www.yaml.org/spec/1.2/spec.html.

LabbookDB: A Wet-Work-Tracking Database Application Framework

Horea-Ioan Ioanas[‡*], Bechara John Saab[§], Markus Rudin[‡]

https://youtu.be/yDBu_wSyw-g

Abstract—LabbookDB is a relational database application framework for life sciences—providing an extendable schema and functions to conveniently add and retrieve information, and generate summaries. The core concept of LabbookDB is that wet work metadata commonly tracked in lab books or spreadsheets is more efficiently and more reliably stored in a relational database, and more flexibly queried. We overcome the flexibility limitations of designed-for-analysis spreadsheets and databases by building our schema around atomized physical object interactions in the laboratory (and providing plotting- and/or analysis-ready dataframes as a compatibility layer). We keep our database schema more easily extendable and adaptable by using joined table inheritance to manage polymorphic objects and their relationships. LabbookDB thus provides a wet work metadata storage model excellently suited for explorative ex-post reporting and analysis, as well as a potential infrastructure for automated wet work tracking.

Index Terms—laboratory notebook, labbook, wet work, record keeping, internet of things, reports, life science, biology, neuroscience, behaviour, relational database, normalization, SQL

Introduction

The laboratory notebook (more commonly, lab book) is a long-standing multi-purpose record—serving as a primary data trace, as a calendar, diary, legal document, memory aid, organizer, timetable, and also proposed as a rapid science communication medium [Bra07]. It is of notable popularity in the natural sciences, especially in the life sciences—where research largely consists of "wet work" (i.e. real-world manipulation), which generally leaves no data trace unless explicitly recorded. With the advent of electronic data acquisition and storage, however, the lab book has increasingly lost significance as a repository for actual data, and has transformed into a metadata record. Notably, the modern lab book has become a general repository of information, for which simple array formats (e.g. tables, spreadsheets, or data matrices) do not provide an adequate input and/or storage format.

Some scientists and science service providers seek to emulate the seemingly convenient lab book format in the electronic medium—even providing support for sketching and doodling (e.g. eLabFTW [CNM12]). Storing information in free-text or pictorial form, however, exacerbates the incompatibility with electronic data analysis and reporting (which commonly requires consistent array formats). This approach, rather than merely retarding information flow by increasing the need for manual lookup and input, can also increase the incidence of biased evaluation—most easily as a consequence of notes being more often or more attentively consulted, and judged by varied but not explicitly documented standards, depending on the expectations of the researcher.

Conversely, researchers often force multidimensional and relationship-rich experimental metadata into the familiar and analysis-apt spreadsheet format. Under a list-like model, however, relationships become spread over many cell combinations while remaining untracked. This leads to information replication in multiple entries, which in turn renders e.g. the task of updating the correspondence between related cells non-trivial. These issues are known as information redundancy and update anomalies, respectively—and are prone to damage data integrity over time. The temptation also arises to truncate input to only what is considered essential at the time of the experiment. This runs the risk of omitting information which may have been easily recorded (even automatically) given a proper data structure, and which may become crucial for closer ex-post deliberation of results.

The crux of the issue, which neither of these approaches adequately addresses, is to store experimental metadata in a fashion which befits its relationship-rich nature, while providing array-formatted data output for analysis, and spreadsheet-formatted data for human inspection. Solutions which provide such functionality for a comprehensive experimental environment are few, and commonly proprietary and enterprise oriented (e.g. iRATS, REDCap [HTT+09]). One notable exception is MouseDB [Bri11], a database application framework built around mouse tracking. This package is considerably more mature than our present endeavour, yet more closely intended as a lab management tool rather than a general lab book replacement. It makes a number of differing structure choices, but given the permissive license (BSD [Ini99]) of both projects, it is conceivable for functionalities from one to be merged into another in the future.

The need for a wet work metadata system providing a better internal data model and consistently structured outputs, is compounded by the fact that such a system may also be better suited for (semi)automatic record keeping. Rudimentary semiautomatic tracking (via barcode-scanning) is already available for at least one commercial platform (iRATS), and the concept is likely to become of even more interest, as the internet of things reaches the laboratory. This makes a well-formed open source relational

* *Corresponding author: ioanas@biomed.ee.ethz.ch*
‡ *Institute for Biomedical Engineering, ETH and University of Zurich*
§ *Preclinical Laboratory for Translational Research into Affective Disorders, DPPP, Psychiatric Hospital, University of Zurich*

schema of object interactions accurately representing the physical world pivotal in the development of laboratory record keeping.

Methods

Database Management System

In order to cast complex laboratory metadata into tractable relationships with high enough entry numbers for statistical analysis, as well as in order to reduce data redundancy and the risk of introducing anomalies, we opt for a relational database management system, as interfaced with via SQLAlchemy. The scalability and input flexibility advantages of noSQL databases do not apply well to the content at hand, as experimental metadata is small, reliable, and slowly obtained enough to make scalability a secondary concern and schema quality and consistency a principal concern. Our robust but easily extendable schema design encapsulates contributors' wet work procedural knowledge, and is valuable in excess of creating an efficient storage model; as well-chosen predefined attributes facilitate reproducibility and encourage standardization in reporting and comparability across experiments.

Database Schema Design

The current database schema was generated from numerous bona fide spreadsheet formats used at the Psychiatry University Clinic, ETH, and University of Zurich. Iteratively, these spreadsheets are being normalized to first, second, third, and fourth normal forms (eliminating multivalued attributes, partial dependencies, transitive dependencies, and multivalued dependencies, respectively) [Cod74]. As the database schema of the current release (`0.0.1`) consists of over 40 tables, and is expected to expand as more facets of wet work are tracked, ensuring that relationships are well-formed will remain an ongoing process. The perpetually non-definitive nature of the database schema is also conditioned by the continuous emergence of new wet work methods.

Record Keeping and Structure Migration

We use version tracking via Git to provide both a verifiable primary input record, and the possibility to correct entries (e.g. typos) in order to facilitate later database usage in analysis. Version tracking of databases, however, is rendered difficult by their binary format. To mitigate this issue, as well as the aforementioned continuous structure update requirement, we track modular Python function calls which use the LabbookDB input application programming interface (API) to generate a database—instead of the database itself. We refer to this repository of Python function calls as the "source code" of the database.

Input Design

The LabbookDB input API consists of Python functions which interface with SQLAlchemy, and accept dictionary and string parameters for new entry specification and existing entry identification, respectively. These Python functions are wrapped for command line availability via `argh`—as sub-commands under the master command `LDB` in order to conserve executable namespace. Dictionaries are passed to the command line surrounded by simple quotes, and a LabbookDB-specific syntax was developed to make entry identification considerably shorter than standard SQL (though only arguably more readable).

Output Design

Outputs include simple human-readable command line reports and spreadsheets, `.pdf` protocols, introspective graphs, and dataframes. Dataframe output is designed to support both the Pandas `DataFrame` format and export as `.csv`. The dataframe conventions are kept simple and are perfectly understood by BehavioPy [Chr16], a collection of plotting functions originally developed as part of LabbookDB, but now branched off for more general usage. The formatting of command line reports is built by concatenating `__str__` methods of queryable objects and their immediate relationships, and is based on the most common use cases for rapid monitoring. Contingent on the availability of object-specific formatting guidelines, an interface is available for generating human-readable, itemized `.pdf` protocols.

Scope

To accommodate for a developing schema, reduce dependencies, and reduce usage difficulty, we opt to showcase LabbookDB as a personal database system, using SQLite as an engine. As such, the database is stored locally, managed without a client-server model, and accessed without the need for authentication. The scope thus extends to maximally a few users, which trust each other with full access. This is an appropriate scope for most research groups. Additionally, this design choice enables single researchers or clusters of researchers within a larger group to autonomously try out, test, contribute to, or adopt LabbookDB without significant overhead or the need for a larger institutional commitment.

Quality Control

LabbookDB provides an outline for unit testing which ships in the form of a submodule. Currently this is populated with a small number of simple example tests for low-level functionality, and is intended to grow as individual code units become more hardened. Additionally, we provide extensive integration testing which assures that the higher-level functionality of LabbookDB remains consistent, and that databases can be regenerated from updated source code as needed. The ever-increasing data required for extensive integration testing is distributed independently of LabbookDB and PIP, in a separate Git repository named Demolog [Chr17b]. Both unit and integration tests are currently run continuously with TravisCI.

Development Model

The database schema draws from ongoing input, testing, and the wet work experience of many researchers associated with the Institute of Biomedical Engineering and the Animal Imaging Center at the ETH and University of Zurich. The development team currently consists of one programmer (corresponding author), who will maintain and actively develop LabbookDB at least until 2019—independently of community involvement. Beyond that time point development may become contingent on the established impact of the project, including number of contributors, academic recognition of the metadata management system, adoption in the scientific Python or biomedical community, or the prospect of developing commercial tools to leverage the open source schema and API.

Documentation

Project documentation is published via Read the Docs, and contains a general project description, alongside installation instructions and a browsable listing of the API. The documentation model is based primarily on docstrings, but also contains example functions and example input stored in the corresponding submodule. A number of fully reproducible minimal input (working with the Demolog data only) versions of these functions are also presented in this paper.

Capabilities

The aforementioned integration testing data reposited as Demolog [Chr17b] demonstrates the capabilities of this first LabbookDB release in a concise fashion. Contingent on the presence of LabbookDB `0.0.1` [Chr17a] and its dependencies on the system, an example database can be built—and correspondingly described subsequent entries can be executed locally. To set up the example database, the following should be run from the terminal:

```
mkdir ~/src
cd ~/src
git clone https://bitbucket.org/TheChymera/demolog
cd demolog/from_python_code
./generate_db.py
mkdir ~/syncdata
cp meta.db ~/syncdata
```

Note that, for the examples to work, it is mandatory to create the `src` and `syncdata` directories under the user's home path.

Entry Insertion and Update

The Python API allows for clearly laid out entry insertion, via the `add_generic()` function:

```
add_generic(db_location, parameters={
    "CATEGORY":"Animal",
    "sex":"m",
    "ear_punches":"L",
    "license":"666/2013",
    "birth_date":"2016,7,21",
    "external_ids":[
        {"CATEGORY":"AnimalExternalIdentifier",
            "database":"ETH/AIC",
            "identifier":"5682",
            },
        {"CATEGORY":"AnimalExternalIdentifier",
            "database":"UZH/iRATS",
            "identifier":"M2889"
            },
        ],
    "genotypes":["Genotype:code.datg"],
    })
```

Technically, all entries could be created in such a fashion. However, in order to better organize logging (e.g. quarterly, as in the Demolog submodules), we provide an additional function for entry update. Instead of editing the original animal input file to set e.g. the death date, the animal entry can be updated via a separate function call:

```
append_parameter(db_location,
    entry_identification="Animal:external_ids."
        "AnimalExternalIdentifier:database."
        "ETH/AIC&#&identifier.5682",
    parameters={
        "death_date":"2017,5,13,17,25",
        "death_reason":"end of experiment",
        }
    )
```

In this example an existing entry is selected in a compact fashion using custom LabbookDB syntax.

Compact Syntax for Entry Selection

In order to compactly identify related for data input, we have developed a custom LabbookDB syntax. This syntax is automatically parsed by the `labbookdb.db.add.get_related_ids()` function, which is called internally by input functions. Notably, understanding of this syntax is not required in order to use reporting functions, and plenty of examples of its usage for input can be seen in Demolog.

Custom LabbookDB syntax is not written as a wrapper for SQL, but rather specifically designed to satisfy LabbookDB entry selection use cases in a minimum number of characters. This is primarily provided to facilitate database manipulation from the command line, though it also aids in making database source code more clearly laid out

Consider the string used to identify the entry to be updated in the previous code snippet (split to fit document formatting):

```
"Animal:external_ids.AnimalExternalIdentifier:datab"
"ase.ETH/AIC&#&identifier.5682"
```

Under the custom LabbookDB syntax, the selection string always starts with the entry's object name (in the string at hand, `Animal`). The object name is separated from the name of the attribute to be matched by a colon, and the attribute name is separated from the value identifying the existing entry by a period. The value can be either a string, or—if the string contains a colon—it is presumed to be another object (which is then selected using the same syntax). Multiple matching constraints can be specified, by separating them via double ampersands. Inserting one or multiple hashtags in between the ampersands indicates at what level the additional constraint is to be applied. In the current example, two ampersands separated by one hashtag mean that an `AnimalExternalIdentifier` object is matched contingent on a `database` attribute value of `"ETH/AIC"` and an `identifier` attribute value of `"5682"`. Had the ampersands not been separated by a hashtag, the expression would have prompted LabbookDB to apply the additional `identifier` attribute constraint not to the `AnimalExternalIdentifier` object, but one level higher, to the `Animal` object.

Command Line Reporting

Quick reports can be generated directly via the command line, e.g. in order to get the most relevant aspects of an animal at a glance. The following code should be executable locally in the terminal, contingent on LabbookDB example database availability:

```
LDB animal-info -p ~/syncdata/meta.db 5682 ETH/AIC
```

The code should return an overview similar to the following, directly in the terminal:

```
Animal(id: 15, sex: m, ear_punches: L):
   license: 666/2013
   birth:   2016-07-21
   death:   2017-05-13 (end of experiment)
   external_ids:    5682(ETH/AIC), M2889(UZH/iRATS)
   genotypes:       DAT-cre(tg)
   cage_stays:
      cage 31, starting 2016-12-06
      cage 37, starting 2017-01-10
   operations:
      Operation(2017-03-04 10:30:00: virus_injection)
      Operation(2017-03-20 13:00:00: optic_implant)
   treatments:
   measurements:
      Weight(2016-12-22 13:35:00, weight: 29.6g)
      Weight(2017-03-30 11:48:00, weight: 30.2g)
      fMRI(2016-12-22 13:35:49, temp: 35.0)
```

```
fMRI(2017-03-30 11:48:52, temp: 35.7)
Weight(2017-04-11 12:33:00, weight: 29.2g)
fMRI(2017-04-11 12:03:58, temp: 34.8)
Weight(2017-05-13 16:53:00, weight: 29.2g)
```

Human Readable Spreadsheets

LabbookDB can join tables from the database in order to construct comprehensive human-readable spreadsheet overviews. Storing information in a well-formed relational structure allows for versatile and complex reporting formats. In the following model, for instance, the "responsive functional measurements" column is computed automatically from the number of fMRI measurements and the number of occurrences of the `"ICA failed to indicate response to stimulus"` irregularity on these measurements.

Contingent on the presence of LabbookDB and the example database, the following lines of code should generate a dataframe formatted in the same fashion as Table 1, and return it directly in the terminal, or save it in `.html` format, respectively:

```
LDB animals-info ~/syncdata/meta.db
LDB animals-info ~/syncdata/meta.db -s overview
```

An example of the `.html` output can be seen in the Demolog repository under the `outputs` directory.

Printable Protocol Output

LabbookDB can create `.pdf` outputs to serve as portable step-by-step instructions suitable for computer-independent usage. This capability, paired with the database storage of e.g. protocol parameters means that one can store and assign very many protocol variants internally (with a minuscule storage footprint), and conveniently print out a preferred protocol for collaborators, technicians, or students, without encumbering their workflow with any unneeded complexity. The feature can be accessed from the `labbookdb.report.examples` module. The following code should be executable locally, contingent on LabbookDB and example database availability:

```
from labbookdb.report.examples import protocol

class_name = "DNAExtractionProtocol"
code = "EPDqEP"
protocol("~/syncdata/meta.db", class_name, code)
```

This should create a `DNAExtractionProtocol_EPDqEP.pdf` file identical to the one tracked in Demolog.

Introspection

LabbookDB ships with a module which generates graphical representations of the complex relational structures implemented in the package. The feature is provided by the `labbookdb.introspection.schema` module. The following code should be executable locally, contingent on LabbookDB availability:

```
from labbookdb.introspection.schema import generate

extent=[
    "Animal",
    "FMRIMeasurement",
    "OpenFieldTestMeasurement",
    "WeightMeasurement",
    ]
save_plot = "~/measurements_schema.pdf"
generate(extent, save_plot=save_plot)
```

This example should generate Figure 1 in `.pdf` format (though `.png` is also supported).

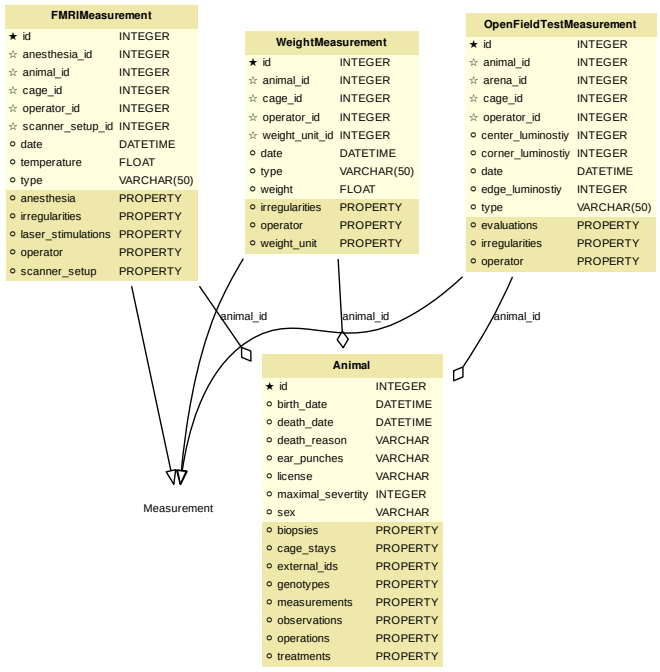


Fig. 1: *LabbookDB schema section, illustrating the polymorphic relationship between Animal objects and different Measurement variants.*

Polymorphic Mapping and Schema Extension

In current research, it is common to subject animals to experimental procedures which are similar in kind, but which can be split into categories with vastly different attributes. Prime examples of such procedures are `Measurements` and `Operations`. In Figure 1 we present how LabbookDB uses SQLAlchemy's joined table inheritance to link different measurement types to the `measurements` attribute of the `Animal` class. Attributes common to all measurement types are stored on the `measurements` table, as are relationships common to multiple measurements (e.g. the relationship to the `Animal` class, instantiated in the `animal_id` attribute).

One of the foremost requirements for a relational database application to become a general purpose lab book replacement is an easily extendable schema. The `Measurement` and `Operation` base classes demonstrate how inheritance and polymorphic mapping can help extend the schema to cover new types of work without changing existing classes. Polymorphism can be extended to more classes, to further propagate this feature. For instance, all measurement subjects in LabbookDB databases are currently recorded as `Animal` objects. This is adequate for most rodents, however it remains inadequate for e.g. human subjects. The issue would best be resolved by creating a `Subject` class, with attributes (including relationships) common to multiple types of subjects, and then creating derived classes, such as `HumanSubject` or `MouseSubject` to track more specific attributes. `Measurement` and `Operation` assignments would be seamlessly transferable, as relationships between objects derived from the `Subject` base class and e.g. the `Operation` base class would be polymorphic.

Animal_id	ETH/AIC	UZH/iRATS	Genotype_code	Animal_death_date	responsive functional measurements
45	6258	M5458	datg	2017-04-20 18:30:00	0/0
44	6262	M4836	eptg	None	2/2
43	6261	M4835	eptg	2017-04-09 18:35:00	0/0
42	6256	M4729	epwt	None	0/0
41	6255	M4728	eptg	None	2/2

TABLE 1: *Example of a human-readable overview spreadsheet generated via the LabbookDB command line functionality.*

Atomized Relationships

We use the expression "atomized relationships" to refer to the finest grained representation of a relationship which can feasibly be observed in the real world. In more common relational model terms, higher atomization would correspond to higher normal forms—though we prefer this separate nomenclature to emphasize the preferential consideration of physical interactions, with an outlook towards more easily automated wet work tracking. Similarly to higher normal forms, increasingly atomized relationships give rise to an increasingly complex relational structure of objects with decreasing numbers of attributes. LabbookDB embraces the complexity thus generated and the flexibility and exploratory power it facilitates. Database interaction in LabbookDB is by design programmatic, an thus ease of human readability of the raw relational structure is only of subordinate concern to reporting flexibility.

An example of relationship atomization is showcased in Figure 2. Here the commonplace one-to-many association between `Cage` and `Animal` objects is replaced by a `CageStay` junction table highlighting the fact that the relationship between `Cage` and `Animal` is bounded by time, and that while it is many-to-one at any one time point, in the overarching record it is, in fact, many-to-many. This structure allows animals to share a cage for a given time frame, and to be moved across cages independently—reflecting the physical reality in animal housing facilities. This complexity is seamlessly handled by LabbookDB reporting functions, as seen e.g. in the command line reporting example previously presented.

Conversely, atomization can result in a somewhat simpler schema, as higher level phenomena may turn out to be special cases of atomized interactions. By design (and in contrast to the MouseDB implementation), we would not track breeding cages as a separate entity, as the housing relationships are not distinct from those tracked by the `CageStay` object. A separate object may rather be introduced for breeding events—which need not overlap perfectly with breeding cages.

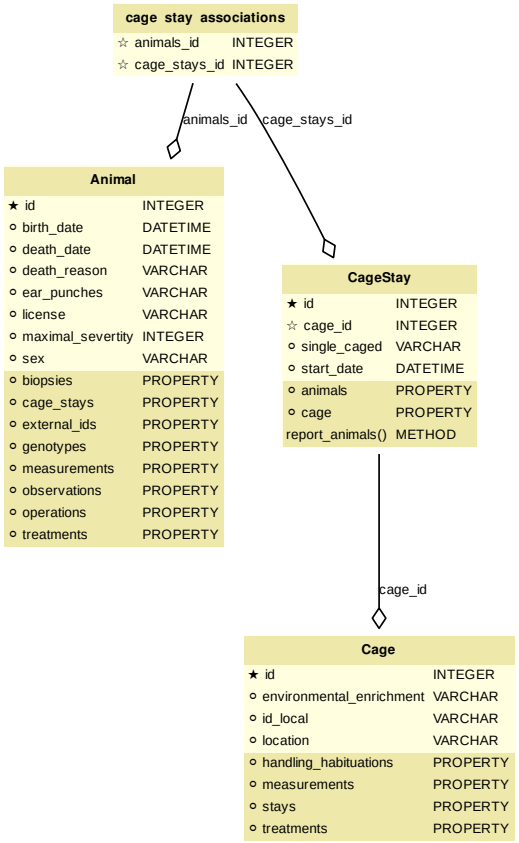


Fig. 2: *LabbookDB schema section, illustrating a more complex and accurate representation of the relational structure linking animals and cages in the housing facility.*

Irregularity and Free Text Management

The atomized schema seeks to introduce structure wherever possible, but also provides a bare minimum set of free-text fields, to record uncategorizable occurrences. Irregular events associated with e.g. `Measurement` or `Operation` instances are stored in the `irregularities` table, and linked by a many-to-many relationship to the respective objects. This not only promotes irregularity re-use, but also facilitates rudimentary manual pattern discovery, and the organic design of new objects within the schema.

Irregular events can also be recorded outside of predetermined interventions, via `Observation` objects. These objects have their own date attribute, alongside free-text attributes, and a `value` attribute, to more appropriately record a quantifiable trait in the observation.

Plotting via BehavioPy

LabbookDB provides a number of powerful data selection and processing functions, which produce consistently structured dataframes that seamlessly integrate with the BehavioPy [Chr16] plotting API. The forced swim test, for instance, is a preclinically highly relevant behavioural assay [PDCB05], which LabbookDB can document and evaluate. The following example code should be executable locally, contingent on LabbookDB, example database, and example data (included in Demolog) availability:

ID	Immobility Ratio	Interval [1 min]	Treatment
28	0.2635	3	Control
28	0.1440	2	Control
30	0.6813	3	Control
1	0.6251	6	Fluoxetine
32	0.6695	5	Fluoxetine
2	0.6498	6	Fluoxetine

TABLE 2: *Example of LabbookDB processed data output for the forced swim test. The format precisely matches the requirements of BehavioPy plotting functions.*

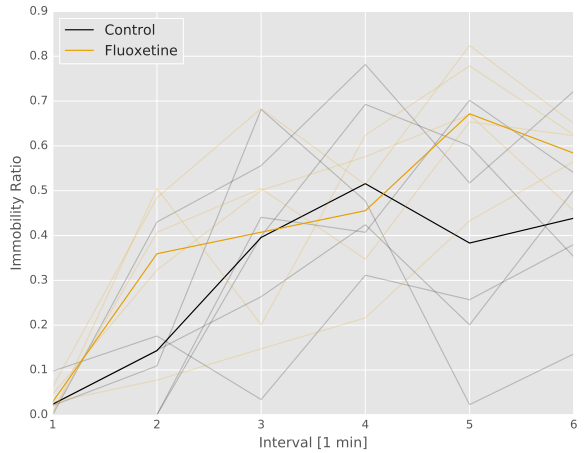


Fig. 3: *Timecourse plot of the forced swim test performed on mice in different treatment groups—automatically generated by LabbookDB, using plotting bindings from BehavioPy.*

```
import matplotlib.pyplot as plt
from labbookdb.report.behaviour import forced_swim

start_dates = ["2017,1,31,22,0","2016,11,24,21,30"]
forced_swim("~/syncdata/meta.db", "tsplot",
    treatment_start_dates=start_dates
    save_df="~/fst_df.csv")
plt.show()
```

The above code prompts LabbookDB to traverse the complex relational structure depicted in Figure 4, in order to join the values relevant to evaluation of the forced swim test. `Animal` objects are joined to `Treatment.code` values via their relationships to `Cage` and `CageStay` objects. This relational structure is determined by the administration of drinking water treatments at the cage level, and thus their contingence on the presence of animals in cages at the time of the treatment. Further, `Evaluation.path` values are joined to `Animal` objects (via their respective relationships to `Measurement` objects) in order to determine where the forced swim test evaluation data is stored for every animal. Subsequently, the annotated event tracking data is processed into desired length time bins (here, 1 minute), and immobility ratios are calculated per bin. Finally, the data is cast into a consistent and easily readable dataframe (formatted in the same fashion as Table 2) which can be both saved to disk, or passed to the appropriate BehavioPy plotting function, to produce Figure 3.

Discussion and Outlook

Record Keeping

Version tracking of database generation source code adequately addresses the main record keeping challenges at this stage of the project. Additionally, it has a number of secondary benefits, such as providing comprehensive and up-to-date usage examples. Not least of all, this method provides a very robust backup—as the database can always be rebuilt from scratch. A very significant drawback of this approach, however, is poor scalability.

As the amount of metadata reposited in a LabbookDB database increases, the time needed for database re-generation may reach unacceptable levels. Disk space usage, while of secondary concern, may also become an issue. Going forward, better solutions for record keeping should be implemented.

Of available options we would preferentially consider input code tracking (if possible in a form which is compatible with incremental execution) rather than output code tracking (e.g. in the form of data dumps). This is chiefly because output code tracking would be dependent not only of the data being tracked, but also of the version of LabbookDB used for database creation—ideally these versioning schemes would not have to become convoluted.

Structure Migration

The long-term unsustainability of database source code tracking also means that a more automated means of structure migration should be developed, so that LabbookDB databases can be recast from older relational structures into improved and extended newer structures—instead of relying on source code editing and regeneration from scratch. Possibly, this could be handled by shipping an update script with every release—though it would be preferable if this could be done in a more dynamic, rolling release fashion.

Data Input

Data input via sequential Python function calls requires a significant amount of boilerplate code, and appears very intransparent for users unaccustomed to the Python syntax. It also requires interfacing with an editor, minding syntax and formatting conventions, and browsing directory trees for the appropriate file in which to reposit the function calls.

While LabbookDB provides a command line interface to input the exact same data with the exact same dictionary and string conventions with arguably less boilerplate code, this input format has not been implemented for the full database generation source code. The main concern precluding this implementation is that the syntax, though simplified form standard SQL, is not nearly simple enough to be relied on for the robustness of thousands of manual input statements generated on-site.

A better approach may be to design automated recording workflows, which prompt the researcher for values only, while applying structure internally, based on a number of templates. Another possibility would be to write a parser for spreadsheets, which applies known LabbookDB input structures, and translates them into the internal relational representation. This second approach would also benefit from the fact that spreadsheets are already a very popular way in which researchers record their metadata—and could give LabbookDB the capability to import large numbers of old records, with comparatively little manual intervention.

Not least of all, the ideal outlook for LabbookDB is to automatically handle as much of the data input process as possible,

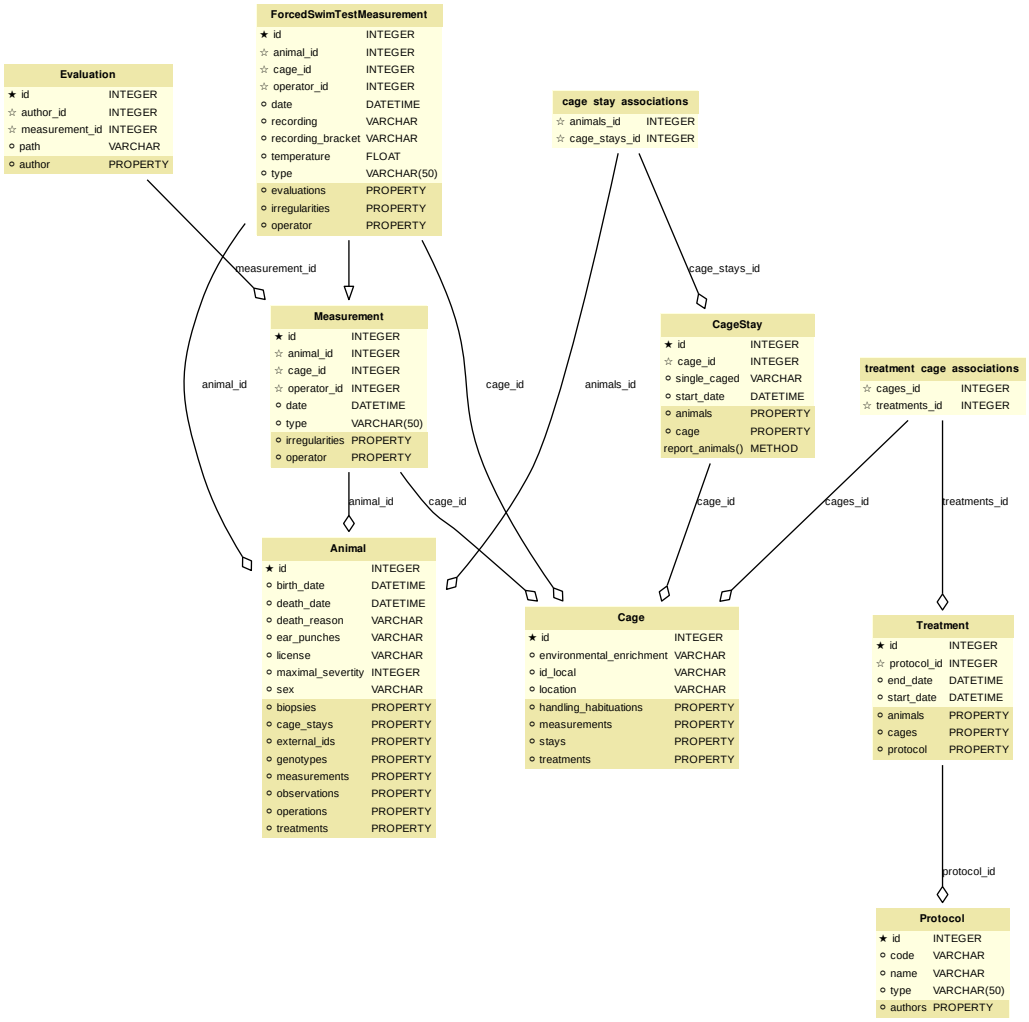


Fig. 4: *LabbookDB schema section relevant for constructing a plottable forced swim test dataframe.*

e.g. via specialized sensors, via semantic image [YJW+16] or video evaluation, or via an entity-barcode-scanner (as currently used by the iRATS system) . This poses nontrivial engineering challenges in excess of relation modelling, and requires distinctly more manpower than currently available. However, LabbookDB is from the licensing point of view suitable for use in commercial products, and additional manpower may be provided by science service providers interested in offering powerful, transparent, and extendable metadata tracking to their discerning customers.

Graphical User Interface

A notable special case of data input is the graphical user interface (GUI). While we acknowledge the potential of a GUI to attract scientists who are not confident users of the command line, we both believe that such an outreach effort is incompatible with the immediate goals of the project and that it is not typically an attractive long-term outlook for scientific Python applications.

Particularly at this stage in development, manpower is limited, and contributions are performed on a per-need basis (little code was written which was not relevant to addressing an actual data management issue). Presently our foremost outreach target are researchers who possess the technical affinity needed to test our schema at its fringes and contribute to—or comment on—our code and schema. A GUI would serve to add further layers of abstraction and make it more difficult for users to provide helpful feedback in our technology development efforts.

In the long run, we would rather look towards developing more automatic or implicit tracking of wet work, rather than simply writing a GUI. Our outlook towards automation also means that a GUI is likely to remain uninteresting for the use cases of the developers themselves, which would make the creation of such an interface more compatible with a commercial service model than with the classical Free and Open Source user-developer model.

REFERENCES

[Bra07] Jean-Claude Bradley. Open notebook science using blogs and wikis. 2007.

[Bri11] Dave Bridges. Mousedb. GitHub, 2011. URL: https://github.com/davebridges/mousedb.

[Chr16] Horea Christian. Behaviopy - behavioural data analysis and plotting in python. GitHub, 2016. URL: https://github.com/TheChymera/behaviopy, doi:10.5281/zenodo.188169.

[Chr17a] Horea Christian. Labbookdb - lab book database schema with information addition, retrieval, and reporting functions. GitHub, 2017. URL: https://github.com/TheChymera/LabbookDB, doi:10.5281/zenodo.823366.

[Chr17b] Horea Christian. Logging examples for labbookdb for scipy2017 proceedings, 05 2017. URL: https://bitbucket.org/TheChymera/demolog/src/9ce8ca3b808259a1cfe74169d7a91fb40e4cfd90?at=master.

[CNM12] Nicolas Carpi, Pascal Noirci, and Alexander Minges. elabftw. online, 2012. URL: https://www.elabftw.net/.

[Cod74] Edgar F. Codd. Recent investigations into relational data base systems. Technical Report RJ1385, IBM, 4 1974.

[HTT+09] Paul A Harris, Robert Taylor, Robert Thielke, Jonathon Payne, Nathaniel Gonzalez, and Jose G Conde. Research electronic data capture (redcap)—a metadata-driven methodology and workflow process for providing translational research informatics support. *Journal of biomedical informatics*, 42(2):377–381, 2009.

[Ini99] Open Source Initiative. The 3-clause bsd license. online, 07 1999. URL: https://opensource.org/licenses/BSD-3-Clause.

[PDCB05] Benoit Petit-Demouliere, Franck Chenu, and Michel Bourin. Forced swimming test in mice: a review of antidepressant activity. *Psychopharmacology*, 177(3):245–255, 2005.

[YJW+16] Quanzeng You, Hailin Jin, Zhaowen Wang, Chen Fang, and Jiebo Luo. Image captioning with semantic attention. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016.

pyMolDyn: Identification, structure, and properties of cavities in condensed matter and molecules

Ingo Heimbach‡*, Florian Rhiem‡, Fabian Beule‡, David Knodt‡, Josef Heinen‡, Robert O. Jones‡

Abstract—pyMolDyn is an interactive viewer of atomic systems defined in a unit cell and is particularly useful for crystalline and amorphous materials. It identifies and visualizes cavities (vacancies, voids) in simulation cells corresponding to all seven 3D Bravais lattices, makes no assumptions about cavity shapes, allows for atoms of different size, and locates the cavity centers (the centers of the largest spheres not including an atom center). We define three types of cavity and develop a method based on the *split and merge* algorithm to calculate all three. The visualization of the cavities uses the *marching cubes* algorithm. The program allows one to calculate and export pair distribution functions (between atoms and/or cavities), as well as bonding and dihedral angles, cavity volumes and surface areas, and measures of cavity shapes, including asphericity, acylindricity, and relative shape anisotropy. The open source Python program is based on `GR framework` and `GR3` routines and can be used to generate high resolution graphics and videos.

Index Terms—Cavity shape, volume, and surface area; Python; marching cubes; split and merge

Introduction

The properties of many materials are influenced significantly or even dominated by the presence of empty regions, referred to as cavities, vacancies, or voids. In phase change materials, for example, they play an essential role in the rapid and reversible transformation between amorphous and crystalline regions of chalcogenide semiconductors [AJ07], [AJ08], [AJ12]. In soft matter, such as polymers, cavities can lead to structural failure and are often crucial for diffusion of small molecules. Voids caused by radiation (neutrons, x-rays) can lead to dramatic changes in the strength of materials. It is essential to provide efficient algorithms and programs to visualize cavities in the course of computer simulations. We describe here methods developed in the context of phase change materials, where the empty regions are rarely larger than a few atomic volumes, and the term "vacancy" is also in common use [LE11b]. The approach will be useful in other contexts. The present manuscript is an extended and corrected version of [Hetal17].

Geometrical algorithms to identify cavities have a long history in the discussion of disordered materials. Bernal [Be64] discussed liquid structures in terms of space-filling polyhedra and noted that "holes" or "pseudonuclei" would occur in general. Finney [Fi70] extended this analysis by using the Dirichlet [Di50] or Voronoi [Vo08] construction, where space is divided into regions bounded by planes that bisect interatomic vectors perpendicularly. This construction for a crystalline solid leads to the well-known Wigner-Seitz cell. The polyhedron associated with an atom is the smallest surrounding the atom in question, and its structural features (volume, number of vertexes, etc.) can be used for identification and characterization. A small Voronoi polyhedron indicates an interstitial defect, and a local assembly of large polyhedra could imply the existence of a cavity. This approach has been used to analyze defect structures in simulations of radiation damage [CL85] and the motion of vacancies in colloids [LAC13], although the coordination number (the number of faces of the Voronoi cell) is not necessarily a convenient measure of their positions [LAC13]. Similar techniques have been applied to the distinction between solute and solvent in a liquid, such as hydrated peptide molecules [Vetal11].

Delaunay triangulation [De34], a division of space closely related to the Dirichlet-Voronoi analysis, has been used to identify the "unoccupied space" [AMS92] or "cavities" [VBM15] in polymer systems and to analyze their connectivity, and it has been used to analyze the normal modes in a molecular dynamics simulation of a glass [LMNS00]. Efficient programs are available for performing Voronoi analyses (see, for example, Ref. [Ry09]) and its extension to Voronoi *S*-surfaces, which are appropriate for systems with atoms of different sizes [MVLG06], [VNP]. Ref. [MVLG06] contains many references to physical applications of Dirichlet-Voronoi-Delaunay analyses. The present work and the above approaches focus on the geometrical arrangement resulting from a simulation, rather than determining other physical quantities such as local atomic pressures [LN88].

In the following section, we define essential terms and describe the analysis, based on the "split and merge" [HP76] and "marching cubes" [LC87], [NY06] algorithms, that we have used to study three definitions of cavities:

- Regions (sometimes referred to as "domains") where each point is outside spheres centered on the atoms. The radii of the spheres are generally element-dependent, but an equal cutoff for all elements (2.5 Å) was chosen in a study of Ge/Sb/Te phase change materials [LE11b].
- "Center-based" cavities resulting from a Dirichlet-Voronoi construction using the atomic positions and the cavity *centers*.
- "Surface-based" cavities [AJ07], where the boundaries are determined by Dirichlet-Voronoi constructions from each

* Corresponding author: i.heimbach@fz-juelich.de
‡ PGI-1 and PGI/JCNS-TA, Forschungszentrum Jülich, D-52425 Jülich, Germany

point on the domain surface to neighboring atoms, have been used in numerous studies of phase change materials [AJ12], [CBP10], [KAJ14].

The code, the ways to access it, and the online documentation are described below, and an application demonstrates some of its features.

Definitions and Algorithms

Essential input for a calculation of cavities and their properties is the location of the atoms, which is often provided as a list of coordinates and atom types within an appropriate unit cell. However, the *definition* of a cavity is not unique and is a prerequisite for any study. Calculation of pair distribution functions involving cavities (with atoms and with other cavities) means that we must also associate appropriate coordinates with the *center* of each cavity. We now define cavities and describe how we calculate their centers.

Cavity domains and cavity centers

The first step is the discretization of the simulation cell by creating a cuboid grid containing the cell bounding box and a surrounding layer, which enables periodic boundary condition to be implemented effectively. The *resolution* $d_{\max}$ refers to the number of points along the longest edge, and two units are added at each end of each cell edge. Each grid point outside the cell has one equivalent point inside. If there are more than one equivalent inside points, we choose the one closest to the origin or—if this is still ambiguous—search for the smallest component in the order x, y, z. Outside points are labeled with the index of the translation vector pointing to the equivalent inside point. This step depends only on the cell shape and the resolution of the discrete grid, and the results can be cached and used for other data files.

As shown in Fig. 1(a), we now construct spheres centered on each atom with radii specified for each atom type (element). In earlier work on alloys of Ge/Sb/Te [AJ07], [AJ12] and Ag/In/Sb/Te [Metal11], the radius was chosen to be the same (2.8 Å) for all elements [r_C in Fig. 1(a)]. Points outside the simulation cell are replaced by equivalent points inside. All points outside these spheres form "cavity domains" [yellow in Fig. 1(a)], and the "cavity center" [X in the 2D scheme 1(b)] is the center of the largest sphere that does not overlap a neighboring atom. It is possible, for example in unusually long, thin cavities, that more than one point satisfy this condition approximately equally well, so that the center can switch between them as a simulation proceeds.

Some structures are unusually sensitive to the choice of cutoff radius r_C and/or resolution, particularly when the cavity domains are very small, and it is essential to carry out detailed tests before performing production runs. The program provides a warning when one or more cavity domains consist of a single cell of the discretization grid. The calculation should be repeated with a higher resolution to increase the number of numerically stable cavity domains.

Domains and center-based cavities

A knowledge of the positions of the atoms and the cavity center enables us to perform a Dirichlet-Voronoi construction (see above) leading to the cavities shown as red in Fig. 1(b). Overlapping cavities from different domains are merged to form "multicavities",

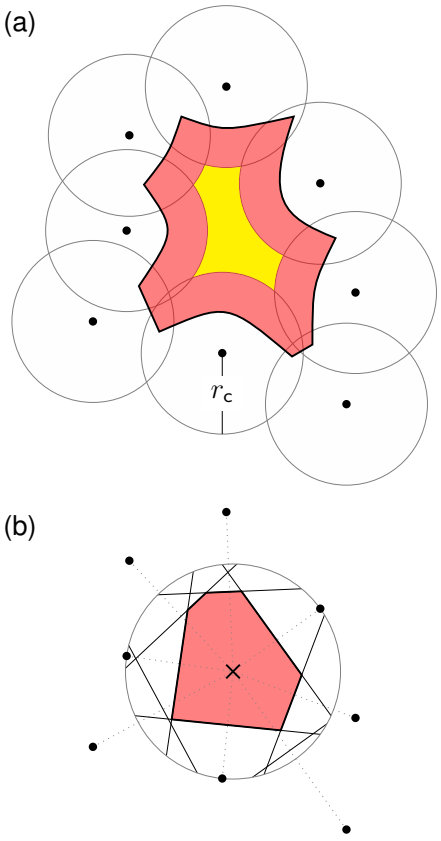


Fig. 1: *Construction for a 2D geometry of (a) "cavity domain" (yellow, cutoff radius r_C) and "surface-based cavity" (red), (b) cavity center (X) and "center-based" cavity (red).*

and the volumes and surface areas of cavities and cavity domains are determined as follows.

Points in domains are grouped together by applying the *split and merge* algorithm [HP76], which consists of three main steps for periodic cells. First, the discrete grid is split recursively into subgrids until they contain either atoms or domain points. Subgrids containing atom points are not needed to determine the domains and are ignored. During the split phase the direct neighbors of each subgrid are recorded, and neighboring subgrid points are then merged to form the cavity domains. As noted above, these domains can be identified as cavities by choosing an appropriate cutoff radius [LE11b].

Center-based cavities comprise points that are closer to domain centers than to any atom, and their construction requires points inside atomic spheres for which there is an atom no farther away than the largest atomic sphere radius. The grid is split into cubes with sides of at least this length, for which only atoms and surface or center points inside neighboring cubes are relevant. If a point is closer to the center of cavity domain i than to the nearest atom, it is marked as part of cavity i. In the case of multicavities, intersections are detected by checking the neighboring points of the surface of a cavity. If two such points belong to different cavities, the two cavities are parts of a common multicavity.

The surface of each domain, cavity, or multicavity is important for calculating the surface area and for visualization, and it is determined by applying a variation of the *marching cubes* algorithm [LC87], [NY06] to new grids based on those derived above. Each grid contains the bounding box of a cavity domain or

multicavity, and each point in the grid is assigned the number of neighboring points inside the domain or cavity. The algorithm then constructs a surface containing all points with a non-zero count. Neighboring grid points are grouped together into disjoint cubes, and points with a count of 1 are found by interpolation along the edges and connected to form triangles. The set of all such triangles is the surface of a domain or cavity.

Surface-based cavities

The surface-based cavity [red in Fig. 1(a)] can be determined as for center-based cavities, except that the Dirichlet-Voronoi construction is performed from each point of the domain surface to the neighboring atoms.

Analysis of structure and cavities

A range of quantities can be calculated for the atomic structure (including bond and dihedral angles) and for each of the above definitions of cavity. In addition to the volume V_C, surface area, and location of the center, we calculate the characteristic radius $r_{\text{char}} = (3V_C/4\pi)^{1/3}$, which is the radius of a spherical cavity with volume V_C. We also evaluate and export pair distribution functions (PDF) between all atom types and/or cavity centers. Continuous representations can be generated using Gaussian, Epanechnikov [Ep69], compact, triangular, box, right box, and left box window functions. The corresponding kernels are listed in the online documentation, and the default bandwidth σ is 0.4 in all cases. Following earlier work [AMS92], [VBM15], [TS85], we calculate the volume-weighted gyration tensor $\mathbf{R}$, which describes the second moment of the coordinates (x, y, z) of points inside a cavity

$$\mathbf{R} = \frac{1}{V_C}\begin{pmatrix} \overline{xx} & \overline{xy} & \overline{xz} \\ \overline{yx} & \overline{yy} & \overline{yz} \\ \overline{zx} & \overline{zy} & \overline{zz} \end{pmatrix}.$$

Here $\overline{xx} = \sum_j^{n_c} v_j x_j x_j$, $\overline{xy} = \sum_j^{n_c} v_j x_j y_j$, ..., v_j is the volume of cell j, and n_c is the number of cells in cavity C. (x_j, y_j, z_j) are the Cartesian coordinates of the center of cell j relative to the centroid or center of gyration of the cavity, which differs in general from the center defined above. Measures of the size and shape of individual cavities are the squared radius of gyration R_g^2, the asphericity η, the acylindricity c, and the relative shape anisotropy κ^2. These are defined as

$$\begin{aligned} R_g^2 &= \lambda_1 + \lambda_2 + \lambda_3 \\ \eta &= \big(\lambda_1 - 0.5(\lambda_2 + \lambda_3)\big)/(\lambda_1 + \lambda_2 + \lambda_3) \\ c &= (\lambda_2 - \lambda_3)/(\lambda_1 + \lambda_2 + \lambda_3) \\ \kappa^2 &= \big(\eta^2 + 0.75c^2\big)/R_g^4, \end{aligned}$$

where λ_1, λ_2, and λ_3 are the ordered eigenvalues of $\mathbf{R}$ ($\lambda_1 \geq \lambda_2 \geq \lambda_3$).

These quantities provide compact information about the symmetry and overall shape of a cavity and have been useful in the context of diffusants in polymers [AMS92]. The asphericity is always non-negative and is zero only when the cavity is symmetric with respect to the three coordinate axes, e.g. for a spherically symmetric or a cubic cavity. The acylindricity is zero when the cavity is symmetric with respect to two coordinate axes, e.g., for a cylinder. The relative shape anisotropy is bounded by zero (spherical symmetry) and unity (all points collinear). The calculation of these shape parameters requires particular care (and more computer time) when cavities cross the boundaries of the unit cell, and the default is not to calculate these parameters. The parameters are also not calculated for (infinite) cavities that span the simulation cell, and a warning is issued in this case.

Description of the Code

The program `pyMolDyn` is written in Python (2.7.13), uses the graphical user interface Qt 5.8.0, the Python module PyQt5 (5.8.2), and the `GR Framework` and `GR3` packages (0.24.0) [HRH15] for 2D- and 3D-graphics, respectively. It has been tested with NumPy (1.12.1). Numerically intensive sections are written in C, compiled using Apple Clang 8.1.0 (macOS) or gcc 4.2.1 (Linux) and embedded using `ctypes` and extension modules. A ready-to-use bundle for OS X (Mavericks, Yosemite, El Capitan) and macOS Sierra is provided at:
http://pgi-jcns.fz-juelich.de/pub/downloads/software/pyMolDyn.dmg
with installation scripts and a package repository for Linux [Debian 8 (Jessie), Ubuntu 16.04 LTS (Xenial Xerus), Centos 7.2, Fedora 25, and OpenSUSE Leap 42.2] at:
https://pgi-jcns.fz-juelich.de/portal/pages/pymoldyn-main.html
Documentation is available in the same directory under pymoldyn-doc.html, with links to the graphical user and command line interfaces. The source code is available via the public git repository:
http://github.com/sciapp/pyMolDyn.

The program supports unit cells of all seven 3D Bravais lattices: triclinic (TRI), monoclinic (MON), orthorhombic (ORT), tetragonal (TET), rhombohedral (RHO), hexagonal (HEX), and cubic (CUB). These cells and the parameters required for their definition are shown in Fig. 2. The bond length cutoffs in all visualizations are 15% longer than the sum of the covalent radii of the elements [OB], [Cetal08]. The default colors for the elements are those used in Jmol/JSmol [Jmol] and other programs ("CPK", Corey-Pauling-Koltun) [SF].

Each frame to be analyzed requires input in the `.xyz`-form, which can be read by Jmol/JSmol and other packages. The first lines of an `.xyz` file are typically:

```
<number of atoms>
<comment>
<element> <X> <Y> <Z>
...
```

where `element` is the symbol for the element in question, e.g. `SB` or `TE`, and `<X>`, `<Y>`, and `<Z>` are the Cartesian coordinates of the first atom. For each atom there is an input line with its coordinates. In `pyMolDyn`, the second (usually comment) line provides the necessary information concerning the Bravais lattice and its parameters. In the case of a hexagonal lattice with $a = 17.68942$ and $c = 22.61158$ (in Å), for example, we write:
`HEX 17.68943 22.61158`
Additional comments on the first line (after the number of atoms and a space) or the second line are ignored, and the space may be used to store additional information.
The organization of the program and the workflow in practice are clarified in the video linked after the title.

Application

The use of `pyMolDyn` is described in detail in the online documentation (see links above). To illustrate its usage, we take the attached input file AgGeS-BOX.xyz, which shows a result of a 500-atom simulation of an amorphous alloy of Ag, Ge, and S

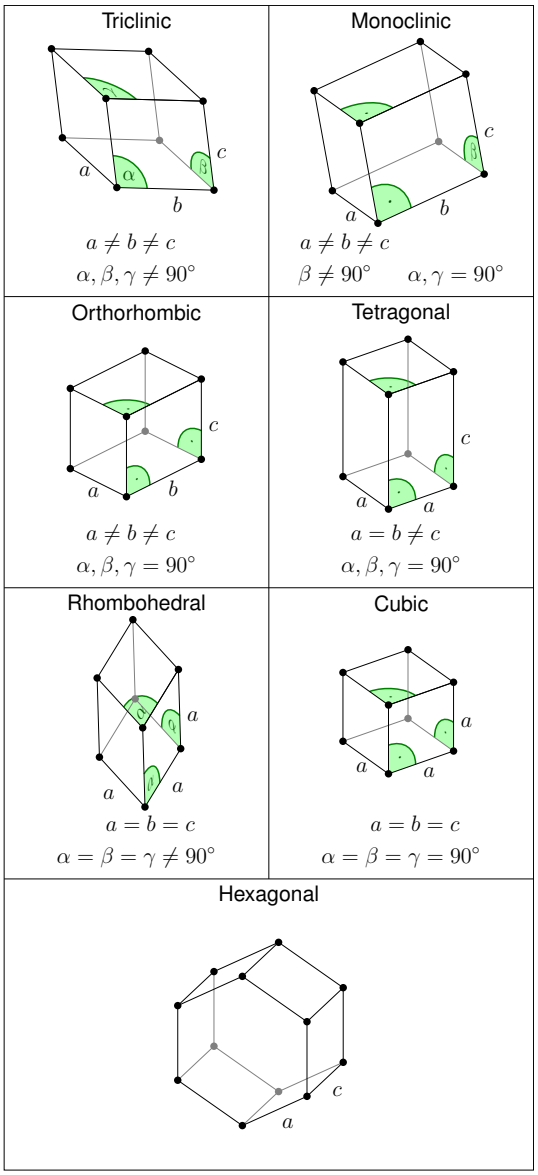


Fig. 2: *The unit cells of the seven 3D Bravais lattices, together with the parameters that define them.*

($Ag_{100}Ge_{168}S_{232}$) in a cubic box of size 21.799 Å [Aetal15]. The first three lines of the input are then:

```
500
CUB 21.799
AG -7.738 ...
```

At this point, clicking "pyMolDyn/Preferences" (OS X, macOS) or "File/Settings" (Linux) allows changes to be made to the default colors for background (black), bounding box (white), bonds (grey), domains (green), and center-based (brown) and surface-based cavities (blue), as well as the cutoff value r_C for calculating surface-based cavities. The default is 2.8 Å for all atoms (we use 2.5 Å in the present application because of the relatively small sulfur atoms), but distinct cutoff radii may be chosen for each element. To guide this choice, the covalent radii for the elements present are provided when setting up the calculation. The resolution can be set by the user and is 384 in the present application. The program is started with the command:

```
pymoldyn
```

The choice of file can be made after clicking "Open", after which "Calculate" leads to the window shown in Fig. 3.

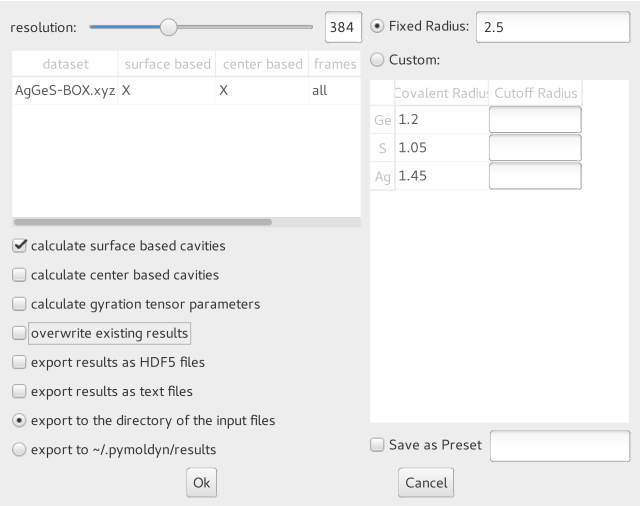


Fig. 3: *Window prior to setting parameters for calculation.*

The resolution and other quantities can then be changed as needed in the appropriate box, after which "OK" starts the calculation and leads to the screen shown in Fig. 4.

The program allows the generation of high-resolution images for publications and presentations, as well as creating videos that illustrate changes in structure (and cavities) as a function of time. Statistics generated by the program include surface areas and volumes (and the surface/volume ratio) of all cavities and domains, pair distribution functions and partial PDF, the distributions of bond types and of bond and dihedral angles, as well as the shape parameters discussed above. Pair distribution functions can be calculated and represented using seven window functions, and properties of individual atoms and cavities may be filtered. This information is available graphically, as an ASCII file, or as hdf5 output. For more details, see https://pgi-jcns.fz-juelich.de/portal/pages/pymoldyn-gui.html

A batch (command line interface) version is useful for generating multiple frames needed for videos and can be called via

```
pymoldyn --batch <filename>
```

Further information concerning the batch version is provided in https://pgi-jcns.fz-juelich.de/portal/pages/pymoldyn-cli.html

Concluding Remarks

The open source program `pyMolDyn` identifies cavities (vacancies, voids) in periodic systems of atoms in a unit cell with one of the seven 3D Bravais lattices. The program makes no assumptions about cavity shapes, allows for atoms of different sizes, and it calculates cavities defined in three ways: (a) "domains" determined by excluding spherical regions around each atom, (b) "center-based" cavities determined by Dirichlet-Voronoi constructions for atoms and cavity centers, and (c) Dirichlet-Voronoi constructions for atoms and points of domain surfaces ("surface-based" cavities). The "split and merge" and "marching cubes" algorithms are utilized. The program is based on the GR3 and GR framework software [HRH15] and the input files use the `.xyz` format used in Jmol/JSmol and other packages.

The size of systems that can be calculated depends on the number of atoms, the necessary resolution, and on the computing

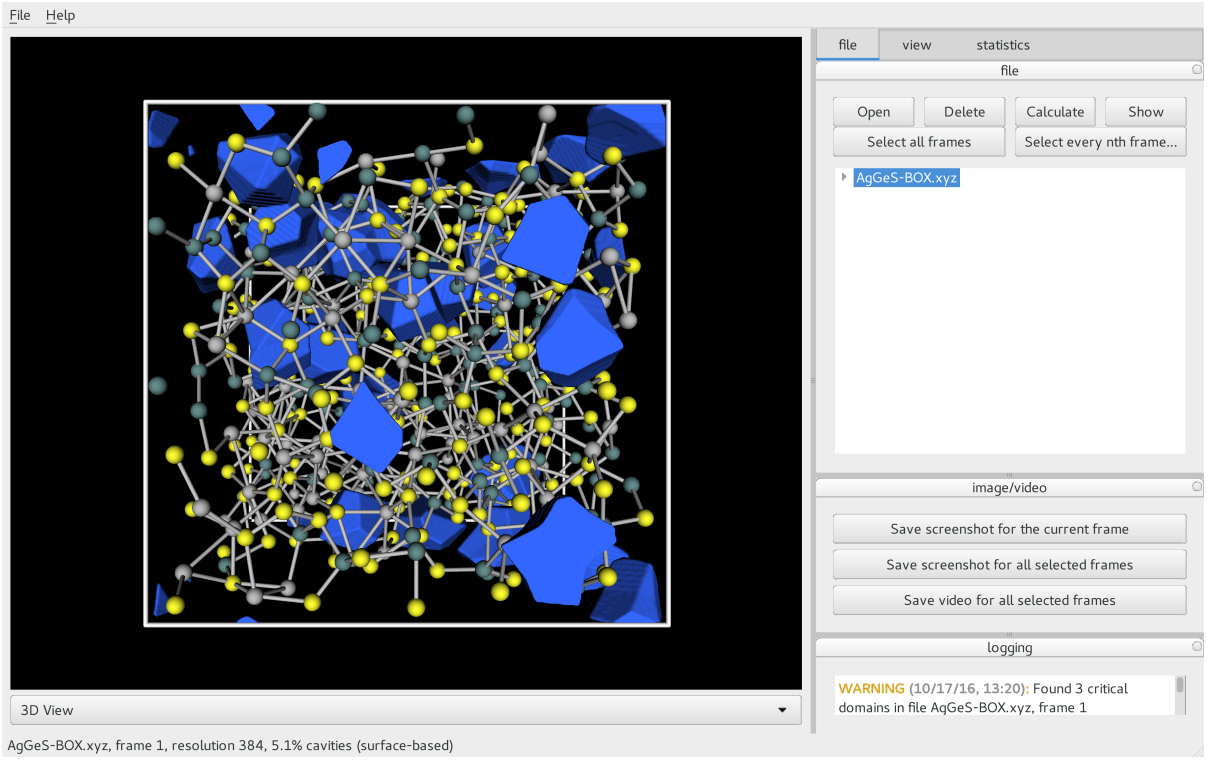


Fig. 4: *Visualization of structure of Ag/Ge/S (silver: Ag, green: Ge, yellow: S) and surface-based cavities (blue).*

resolution	time
128	4.0 s
192	6.5 s
256	15.3 s
384	43.0 s
512	158.3 s

TABLE 1: *Time consumption for calculating surface and center based cavities for* `AgGeS-BOX.xyz` *on a 2.5 GHz Core i7 with 16 GB RAM.*

hardware (processor, memory) used. Systems with 500 atoms (for example `AgGeS-BOX.xyz`) can be computed in minutes on a 2.5 GHz Intel Core i7 (see TABLE 1).

Up to a resolution of 512 points the memory consumption is modest (4 GB of RAM are adequate). For higher resolutions over 10 GB of memory should be available.

Extensions to simplify calculations for isolated molecules and to allow the easy use of many-core, large memory computers are being implemented. We welcome suggestions and contributions to this ongoing project. Full details are available on https://pgi-jcns.fz-juelich.de/portal/pages/pymoldyn-main.html.

Acknowledgments

We thank numerous colleagues, particularly J. Akola, for suggestions and contributions. The program was developed to analyze the results of simulations of phase change materials carried out on supercomputers in the Forschungszentrum Jülich. We are grateful for computer time provided for this purpose by the JARA-HPC Vergabegremium on the JARA-HPC partition of JUQUEEN and for time granted on JUROPA and JURECA at the Jülich Supercomputer Centre.

REFERENCES

[AJ07] Akola, J. and Jones, R. O., *Phys. Rev. B* **2007**, *76*, 235201.
[AJ08] Akola, J. and Jones, R. O., *J. Phys.: Condens. Matter* **2008**, *20*, 465103.
[AJ12] Akola, J. and Jones, R. O., *Phys. Status Solidi B* **2012**, *249*, 1851–1860.
[LE11b] Lee, T. H. and Elliott, S. R., *Phys. Rev. B* **2011**, *84*, 094124.
[Hetal17] I. Heimbach, F. Rhiem, F. Beule, D. Knodt, J. Heinen and R. O. Jones, *J. Comput. Chem.* **2017**, *38*, 389–394.
[Be64] Bernal, J. D., *Proc. R. Soc. A* **1964**, *280*, 299–322.
[Fi70] Finney, J. L., *Proc. R. Soc A* **1970**, *319*, 479–493.
[Di50] Lejeune Dirichlet, G., *J. Reine Angew. Mathematik* **1850**, *40*, 209–227.
[Vo08] Voronoï, G., *J. Reine Angew. Mathematik* **1908**, *134*, 198–287.
[CL85] Chaki, T. K. and Li, J. C. M., *Philos. Mag. B* **1985**, *51*, 557–565.
[LAC13] Laghaei, R., Asher, S. A. and Coalson, R. D., *J. Phys. Chem. B* **2013**, *117*, 5271–5279.
[Vetal11] Voloshin, V. P., Medvedev, N. N., Andrews, M. N., Burri, R. R., Winter, R. and Geiger, A., *J. Phys. Chem. B* **2011**, *115*, 14217–14228.
[De34] Delaunay, B., *Bulletin de l'Académie des Sciences de l'URSS. Classe des sciences mathématiques et naturelles* **1934**, *6*, 793–800.
[AMS92] Arizzi, S., Mott, P. H. and Suter, U. W., *J. Polym. Sci., Part B: Polym. Phys.* **1992**, *30*, 415–426.
[VBM15] Voyiatzis, E., Böhm, M. C. and Müller-Plathe, F., *Comput. Phys. Commun.* **2015**, *196*, 247–254.
[LMNS00] Luchnikov, V. A., Medvedev, N. N., Naberukhin, Y. I. and Schober, H. R., *Phys. Rev. B* **2000**, *62*, 3181–3189.
[Ry09] Rycroft, C., *Voro++: a three-dimensional Voronoi cell library in C++*, United States Department of Energy, 2009. http://www.osti.gov/scitech/servlets/purl/946741
[MVLG06] Medvedev, N. N., Voloshin, V. P., Luchnikov, V. A. and Gavrilova, M. L., *J. Comput. Chem.* **2006**, *27*, 1676–1692.
[VNP] VNP program: Calculation of the Voronoi *S*-network (Additively weighted Voronoi Diagram), http://www.kinetics.nsc.ru/mds/?Software:VNP

[LN88] Laakkonen, J. and Nieminen, R. M., *J. Phys. C: Solid St. Phys.* **1988**, *21*, 3663–3685.
[HP76] Horowitz, S. L. and Pavlidis, T., *J. ACM* **1976**, *23*, 368–388.
[LC87] Lorensen, W. E. and Cline, H. E., *SIGGRAPH Comput. Graph.* **1987**, *21*, 163–169.
[NY06] Newman, T. S. and Yi, H., *Computers & Graphics* **2006**, *30*, 854–879.
[CBP10] Caravati, S., Bernasconi, M. and Parrinello, M., *Phys. Rev. B* **2010**, *81*, 014201.
[KAJ14] Kalikka, J., Akola, J. and Jones, R. O., *Phys. Rev. B* **2014**, *90*, 184109.
[Metal11] Matsunaga, T., Akola, J., Kohara, S., Honma, T., Kobayashi, K., Ikenaga, E., Jones, R. O., Yamada, N., Takata, M. and Kojima, R., *Nature Mater.* **2011**, *10*, 129–134.
[Ep69] Epanechnikov, V. A., *Theory Probabl. Appl.* **1969**, *14*, 153–158.
[TS85] Theodorou, D. N. and Suter, U. W., *Macromolecules (Washington, DC, U.S.)*, **1985**, *18*, 1206–1214.
[HRH15] Heinen, J., Rhiem, F., Felder, C., Beule, F., Brandl, G., Dück, M., Goblet, M., Heimbach, I., Kaiser, D., Klinkhammer, P., Knodt, D., Nesselrath, R., Westphal, E. and Winkler, J., GR—a universal framework for visualization applications, 2015. http://gr-framework.org
[OB] Open babel file: element.txt, http://sourceforge.net/p/openbabel/code/5041/tree/openbabel/trunk/data/element.txt
[Cetal08] Cordero, B., Gomez, V., Platero-Prats, A. E., Reves, M., Echeverria, J., Cremades, E., Barragan, F. and Alvarez, S., *Dalton Trans.* **2008**, 2832–2838.
[Jmol] Jmol: an open-source Java viewer for chemical structures in 3D, http://www.jmol.org or http://wiki.jmol.org/. JSmol is an implementation of Jmol that does not require Java and runs on any web browser supporting HTML5.
[SF] Jmol colors, http://jmol.sourceforge.net/jscolors/
[Aetal15] Akola, J., Beuneu, B., Jones, R. O., Jóvári, P., Kaban, I., Kolář, J., Voleská, I. and Wágner, T., *J. Phys.: Condens. Matter* **2015**, *27*, 485304.

PyHRF: A Python Library for the Analysis of fMRI Data Based on Local Estimation of the Hemodynamic Response Function

Jaime Arias[§†*], Philippe Ciuciu[‡†], Michel Dojat[¶‖†], Florence Forbes[§†], Aina Frau-Pascual[§†], Thomas Perret[§†], Jan M. Warnking[¶‖†]

https://youtu.be/Usyvds1o4lQ

Abstract—Functional Magnetic Resonance Imaging (fMRI) is a neuroimaging technique that allows the non-invasive study of brain function. It is based on the hemodynamic variations induced by changes in cerebral synaptic activity following sensory or cognitive stimulation. The measured signal depends on the variation of blood oxygenation level (BOLD signal) which is related to brain activity: a decrease in deoxyhemoglobin concentration induces an increase in BOLD signal. The BOLD signal is delayed with respect to changes in synaptic activity, which can be modeled as a convolution with the Hemodynamic Response Function (HRF) whose exact form is unknown and fluctuates with various parameters such as age, brain region or physiological conditions.

In this paper we present PyHRF, a software to analyze fMRI data using a Joint Detection-Estimation (JDE) approach. It jointly detects cortical activation and estimates the HRF. In contrast to existing tools, PyHRF estimates the HRF instead of considering it as a given constant in the entire brain. Here, we present an overview of the package and showcase its performance with a real case in order to demonstrate that PyHRF is a suitable tool for clinical applications.

Index Terms—BOLD response, fMRI, hemodynamic response function

Introduction

Neuroimaging techniques, such as functional Magnetic Resonance Imaging (fMRI), allow the *in vivo* study of brain function by measuring the cerebral response to sensory or cognitive stimulation. For more than 20 years, the Blood-Oxygen-Level-Dependent (BOLD) fMRI modality has being the technique most used by neuroscientists to map the main functional regions and their links in the healthy and diseased brain.

The BOLD signal [OLKT90] reflects the changes in deoxyhemoglobin concentration in the brain. Briefly, when brain activity increases, local oxygen consumption in brain tissue increases, slightly increasing the concentration of deoxyhemoglobin in blood (see Fig. 1). Subsequently, cerebral blood flow is strongly upregulated locally by arteriolar vasodilation to replenish the tissue, increasing local blood oxygen saturation in veins and capillaries above the initial level. Oxygenated and deoxygenated blood has different magnetic properties. As a result, the above causes a BOLD signal increase. Thus, the BOLD signal is an indirect measure of cerebral activity based on physiological changes in oxygen consumption, cerebral blood flow and cerebral blood volume.

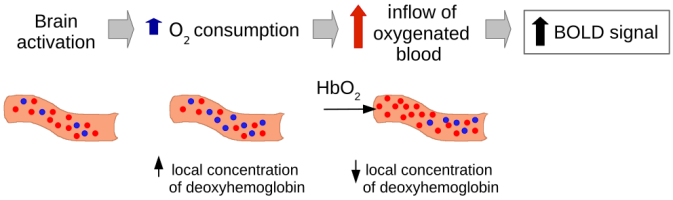


Fig. 1: *fMRI BOLD signal [OLKT90]. The BOLD signal measures the local changes in blood oxygenation ocurring during brain activity.*

fMRI data is acquired by repeated imaging of the brain while the subject or patient executes a task or receives a sensory stimulus during repeated epochs separated by periods of rest. This data is analyzed by correlating the measured time-varying BOLD signal in each image location with a predicted BOLD signal, obtained by convolving the known function representing the stimulus with a Hemodynamic Response Function (HRF) modeling the delay in the vascular response. Locations in the brain where this correlation is statistically significant are considered to exhibit a neuronal response to the task or stimulus, and thus to be involved in its cognitive processing.

BOLD fMRI is non-invasive, non-ionizing, and gives access *in vivo* to brain activity with a relatively high spatial resolution. However, it is highly dependent of the HRF of the brain. The BOLD signal does not give access to true physiological parameters such as cerebral blood flow or cerebral blood volume, but rather measures a mixture of these quantities that is difficult to untangle. In this regard, BOLD is a very interesting tool in neuroscience, but in general it is not widely used for clinical applications because the impact of physiopathological situation on the HRF and the response amplitude are unknown, hampering the BOLD signal interpretation. For instance, the vascular response giving rise to the BOLD signal is altered in normal ageing [FGM+14]

† *These authors contributed equally.*
∗ *Corresponding author: jaime.arias@inria.fr*
§ *Univ. Grenoble Alpes, Inria, MISTIS, LJK, F-38000 Grenoble, France*
‡ *CEA/NeuroSpin and Inria Saclay, Parietal, France*
¶ *Inserm, U1216, F-38000 Grenoble, France*
‖ *Univ. Grenoble Alpes, GIN, F-38000 Grenoble, France*

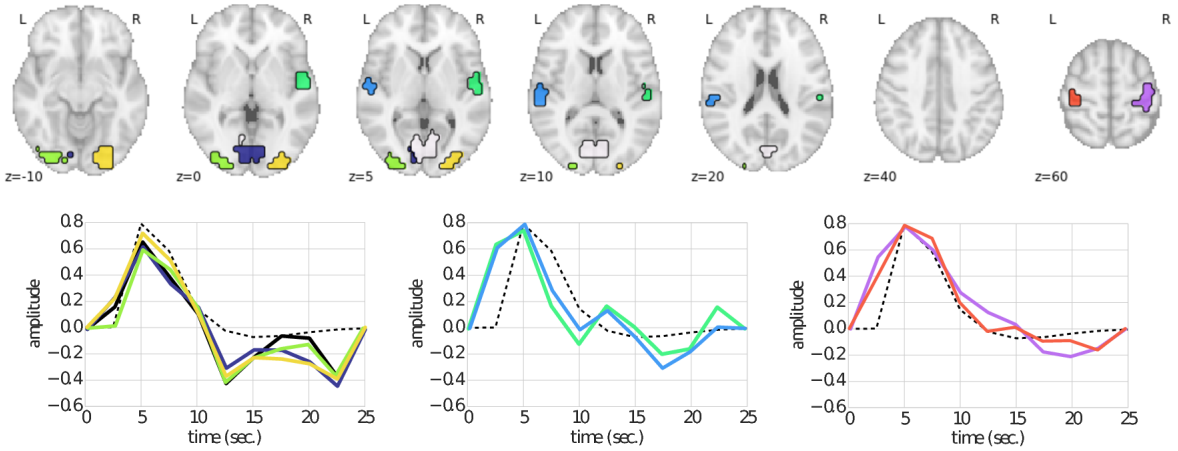


Fig. 2: *HRF computed using PyHRF from BOLD data in several parcels belonging, respectively from left to right, to visual (yellow, dark blue and green parcels), auditory (cyan and light green parcels) and motor (red and purple parcels) regions.*

and pathologies like Alzheimer's disease [CVM+11] or Stroke [AVT+14].

Most used open source libraries for the analysis of fMRI data (*e.g.,* SPM[1], FSL[2]) consider the HRF as constant in all the brain and the same for all subjects. However, several works (see [BVC13] for a survey) show that the HRF changes across different regions of the brain and across individuals, increasing thus the possibility of obtaining false negatives and decreasing the reliability of the results. The software PyHRF [VBR+14] was developed to overcome the above limitation by analyzing fMRI data using a Joint Detection-Estimation (JDE) approach.

In the JDE approach, the detection of the cortical activation is achieved together with the estimation of the unknown HRF response by analyzing non-smoothed data. This detection-estimation is calculated on different parcels of interest paving the cerebral volume. Therefore, PyHRF allows to navigate throughout the brain and to focus on regions of interest during the experiment in order to visualize the activations and their temporal behavior through the estimated HRF. Over the last years, efforts have been made in terms of image processing, user-friendliness and usability of the PyHRF package to make it more easy to use by non experts and clinicians.

Next, we present the PyHRF package. Then we illustrate its use on real fMRI data. Finally, we conclude by discussing directions of current/future work. An online Jupyter notebook containing the results presented here can be found at http://www.pyhrf.org/scipy2017_notebook.

PyHRF

PyHRF (http://www.pyhrf.org) is an open source tool implemented in Python that allows to jointly detect activation and estimate (JDE) the hemodynamic response function (HRF) [MIV+08], which gives the temporal changes in the BOLD effect induced by brain activity. This estimation is not easy in a *voxel-wise* manner [PJG+03], and a spatial structure was added to JDE [VRC10] in order to get reliable estimates. In this regard, HRF estimation in JDE is *parcel-wise* and an input parcellation is required. However, the use of the Markov Chain Monte Carlo (MCMC) method for estimation added a huge computational load to the solution, leading to the development of a faster method to deal with the parameter estimation. Thus, a Variational Expectation Maximization (VEM) solution [CVF+13] was implemented.

JDE aims at improving activation detection by capturing the correct hemodynamics, since using the wrong HRF function could hide existing activations. The use of a canonical HRF is usually sufficient for activation detection. However, HRF functions have been found to have different shapes in different regions [HOM04], and to have different delays in specific populations [BVC13]. They change depending on pathologies such as stenosis.

Fig. 2 shows some HRF functions estimated by PyHRF from BOLD data on a healthy adult. This data was acquired in a block-design setting with visual, auditory and motor experimental conditions. The parcels correspond to regions of the brain that are known induce evoked activity in response to these experimental conditions. Observe that the HRF estimated in the visual and motor regions (first and third figure from left to right) are well approximated by the canonical HRF whereas in the auditory area (second figure from left to right), the recovered hemodynamic profiles peak earlier than the canonical shape.

Standard methods (*e.g.,* GLM), with the posterior classical statistics applied, give Statistical Parametric Maps (SPM) that describe the significance of the activation in each region. JDE is a Bayesian approach and estimates, for each parameter, posterior probability functions. For this reason, we can compute Posterior Probability Maps (PPMs) from the output of PyHRF. These PPMs are not directly comparable to the classical SPM maps, but give a similar measure of significance of activation. For instance, in Fig. 4 we show the SPM and PPM maps for a visual experimental condition in the same data used for Fig. 2. We used the package Nilearn (http://nilearn.github.io) to generate the brain maps presented in this document.

In Fig. 3 we present the inputs and the outputs of PyHRF for the analysis of BOLD data. It needs as inputs the data volume (BOLD), the experimental paradigm, and a parcellation of the brain. After running the JDE algorithm, the output will consist of HRF functions per parcel, BOLD effect maps per experimental condition, and PPMs per condition. In the next section, we will describe in more details these elements and how to use PyHRF.

1. SPM official website: http://www.fil.ion.ucl.ac.uk/spm/software/
2. FSL official website: https://fsl.fmrib.ox.ac.uk/fsl/fslwiki/

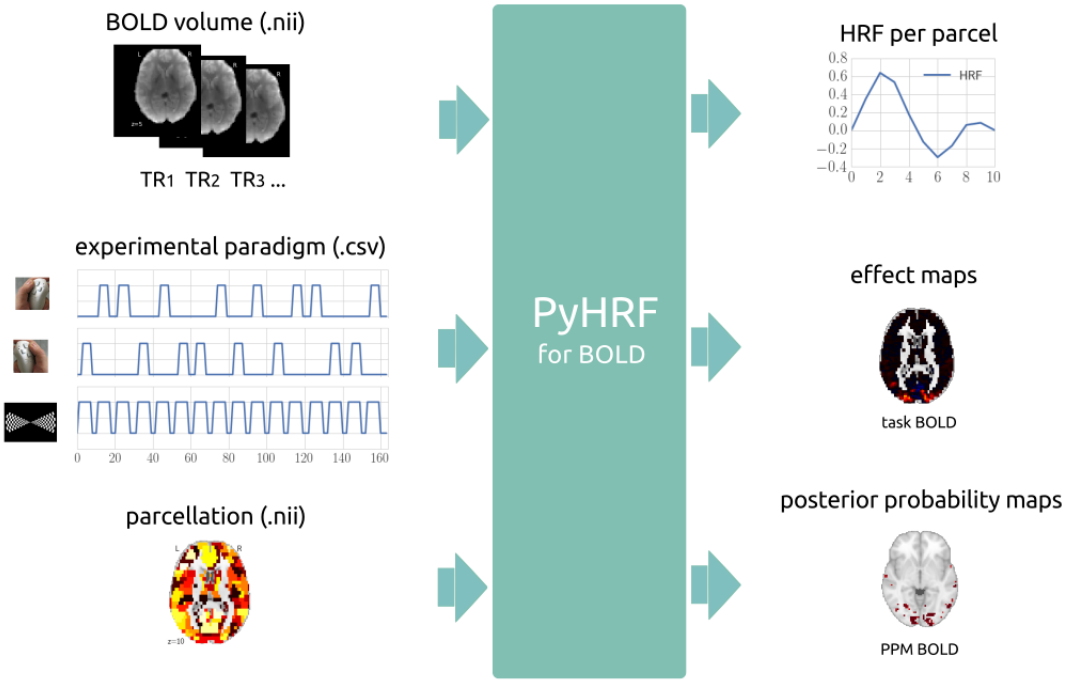


Fig. 3: *Inputs and outputs of PyHRF when analyzing BOLD data.*

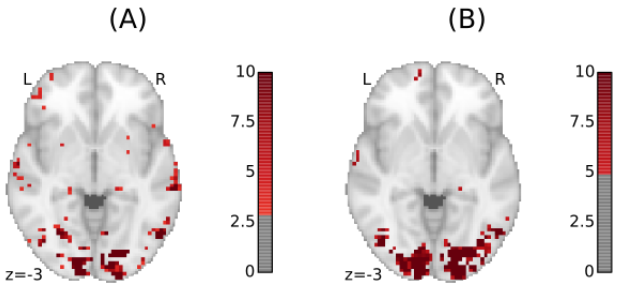


Fig. 4: *A) PPM and B) SPM maps computed with JDE and GLM, respectively. Scale is logarithmic.*

Example of Use

To illustrate the use of PyHRF, we will describe the steps for the analysis of BOLD data. A Jupyter notebook containing the complete code is available at http://www.pyhrf.org/scipy2017_notebook.

Getting fMRI BOLD Data

First of all, we need to get our fMRI BOLD data. In our running example, we will analyze the dataset used in [GSB+13]. This dataset (`ds000114`) is open shared and it can be downloaded from https://openfmri.org/dataset/ds000114/. For convenience, we implemented the method `get_from_openfmri` that uses the library `fetchopenfmri` (https://github.com/wiheto/fetchopenfmri) to download datasets from the site `openfmri`. For instance, the following code downloads the dataset `ds000114` to the folder `~/data`.

```
>>> dataset_path = get_from_openfmri('114', '~/data')
Dataset ds000114 already exists
/home/jariasal/data/openfmri/ds000114_R2.0.1
```

Briefly, in this dataset ten healthy subjects in their fifties were scanned twice using an identical experimental paradigm. This paradigm consists of five task-related fMRI time series: finger, foot and lip movement; overt verb generation; covert verb generation; overt word repetition; and landmark tasks. For the sake of simplicity, we will focus our analysis only on motor tasks (*i.e.,* finger, foot and lips movement). Fig. 5 shows the paradigm containing only the three tasks mentioned above. As we can see in the experimental paradigm, tasks do not overlap each other and stimuli are presented to the subject during a certain time (*i.e.,* block paradigm).

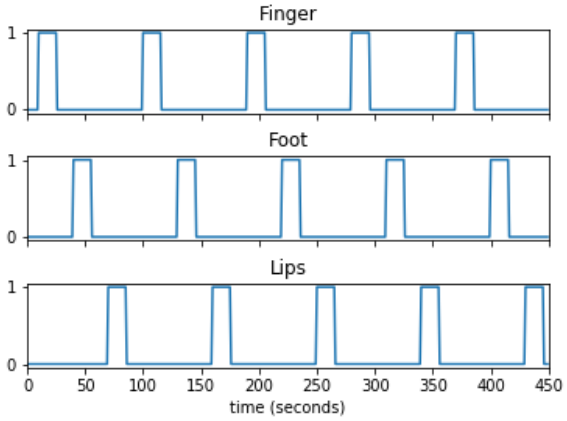


Fig. 5: *Experimental paradigm of the dataset `ds000114`. We show only the motor tasks of the dataset (finger, foot and lips movement).*

fMRI BOLD Preprocessing

Once we have the BOLD volumes, we need to apply some transformations to the images in order to correct for possible errors that may have occurred along the acquisition. For instance,

a BOLD volume (*e.g.,* a whole brain) is usually not built at once but using a series of successively measured 2D slices. Each slice takes some time to acquire, so slices are observed at different time points, leading to suboptimal statistical analysis.

We used the library `Nipype` (https://github.com/nipy/nipype) to define and apply our preprocessing pipeline. This library allows to use robust tools, such as SPM and FSL, in an easy manner. The proposed workflow (see Fig. 6) starts by uncompressing the images since they are in a `nii.gz` format (`gunzip_func` and `gunzip_anat` nodes). After, it applies a *slice timing* operation in order to make appear that all voxels of the BOLD volume have been acquired at the same time. We then apply a *realignment* in order to correct for head movements. Moreover, we apply a *coregistration* operation in order to have the anatomical image (high spatial resolution) in the same space as the BOLD images. Finally, we *normalize* our images in order to transform them into a standard space (a template).

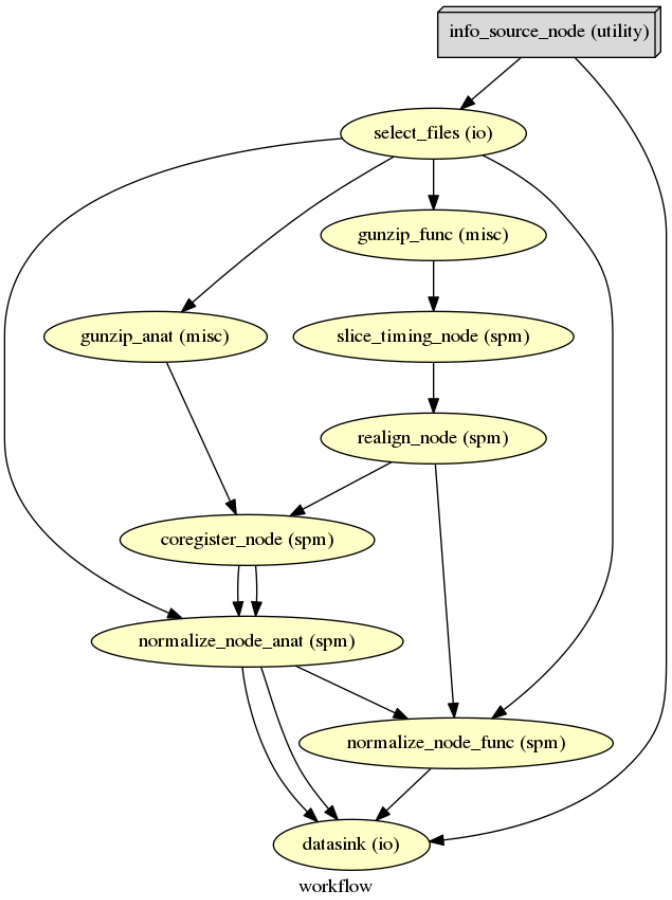


Fig. 6: *Preprocessing pipeline defined with `Nipype` and used in our running example.*

The pipeline described above was run for the images of all subjects from the dataset (*i.e.,* 10 subjects) on multiple processors, since `Nipype` uses the library `joblib` (https://github.com/joblib/joblib). We used the acquisition parameters presented in [GSB+13] to parameterize each preprocessing operation. For instance, the number of slices for the volume, the time for acquiring all slices (TR), and the order in which they were acquired (*e.g.,* interleaved).

In the following snippet, we show a portion of the code to define the slice timing task with `Nipype`.

```
>>> import nipype.interfaces.spm as spm
>>> import nipype.pipeline.engine as npe

# Acquisition parameters
>>> TR = 2.5
>>> NUM_SLICES = 30
>>> TA = TR - (TR / NUM_SLICES)
>>> REF_SLICE = 1

# interleaved slice order
>>> SLICE_ORDER = list(range(1, NUM_SLICES+1, 2) +
                       range(2, NUM_SLICES+1, 2))

# slice timing with SPM
>>> spm_st = spm.SliceTiming(num_slices=NUM_SLICES,
                             time_repetition=TR,
                             time_acquisition=TA,
                             slice_order=SLICE_ORDER,
                             ref_slice=REF_SLICE)
>>> slice_timing = npe.Node(spm_st,
                            name='slice_timing_node')
```

PyHRF Analysis (Inputs)

So far, we have prepared our functional and structural images for BOLD analysis. It is important to note that PyHRF receives *non-smoothed* images as input, thus we excluded this operation from our preprocessing pipeline.

For the sake of simplicity, in our running example we only analyze the 4th subject from our dataset. Additionally, we use the package `Nilearn` to load and visualize neuroimaging volumes. Fig. 7 shows the mean of the functional images of the 4th subject after preprocessing.

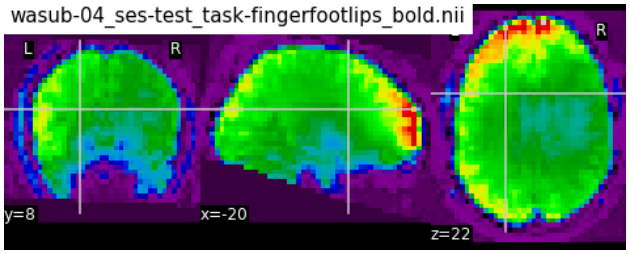


Fig. 7: *Mean of all preprocessed functional images (over time) of the 4th subject of the dataset `ds000114`.*

As we explained before, the JDE framework estimates HRF parcel-wise. This means that PyHRF needs a parcellation mask to perform the estimation-detection. The package provides a Willard atlas [RAM+15] (see Fig. 8) created from the files distributed by Stanford (http://findlab.stanford.edu/functional_ROIs.html). This atlas has a voxel resolution of 3x3x3 mm and a volume shape of 53x63x52 voxels.

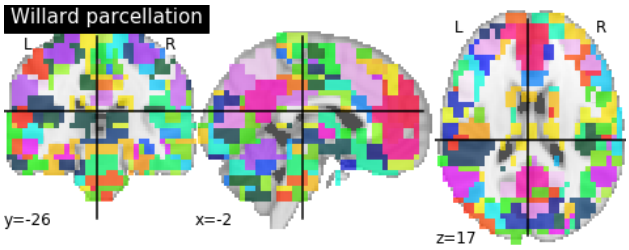


Fig. 8: *Willard atlas [RAM+15].*

We used the method `get_willard_mask` to resize the original atlas to match the shape of the BOLD images to be

session	condition	onset	duration	amplitude
0	Finger	10	15.0	1
0	Foot	40	15.0	1
0	Lips	70	15.0	1
0	Finger	100	15.0	1
0	Foot	130	15.0	1
0	Lips	160	15.0	1
0	Finger	190	15.0	1
0	Foot	220	15.0	1
0	Lips	250	15.0	1
0	Finger	280	15.0	1
0	Foot	310	15.0	1
0	Lips	340	15.0	1
0	Finger	370	15.0	1
0	Foot	400	15.0	1
0	Lips	430	15.0	1

TABLE 1: *Experimental paradigm of the dataset* `ds000114` *containing only motor stimuli. The column organization of the file follows the PyHRF format.*

analyzed. In addition, this method saves the resampled mask in a specified path. For instance, Fig. 8 shows the Willard atlas resized to the shape of the functional image in Fig. 7. The following code illustrates how to resize the Willard atlas provided by PyHRF to match the shape of the image `~/data/bold.nii`, and saves it in the folder `~/pyhrf`.

```
>>> willard = get_willard_mask('~/pyhrf',
                               '~/data/bold.nii')
/home/jariasal/pyhrf/mask_parcellation/willard_3mm.nii
```

PyHRF also needs the experimental paradigm as input. It must be given as a `csv` file following the convention described in the documentation (https://pyhrf.github.io/manual/paradigm.html). For the sake of convenience, we used the method `convert_to_pyhrf_csv` to read the paradigm file provided by the dataset (a `tsv` file) and rewrite it using the PyHRF format. Since each dataset has its own format for the paradigm, we give it as an input to our method.

```
>>> columns_tsv = ['onset', 'duration', 'weight',
                   'trial_type']
>>> paradigm = convert_to_pyhrf_csv(
                '~/data/paradigm.tsv', 0,
                columns_tsv)
/tmp/tmpM3zBD5
```

Table 1 shows the experimental paradigm of the dataset `ds000114` written using the PyHRF format. Note that it only contains motor stimuli since we are only interested in them for our BOLD analysis.

PyHRF Analysis (Run)

Now we are ready to start our BOLD analysis with PyHRF. For that, we need to define some important parameters of the underlying JDE model (*e.g.,* `beta`, `hrf-hyperprior`, `sigma-h`, `drifts-type`) and a folder to save the output (`output`).

Moreover, we need to specify if we want to estimate the HRF response or use, for example, its canonical form. In our running example, we will estimate the HRF (`estimate-hrf`) with a time resolution (`dt`) of 1.25 s, a duration (`hrf-duration`) of 25.0 s, and we force to zero the beginning and ending of the response (`zero-constraint`).

Once the parameters of the model have been defined, we run our analysis by using the command-line tool `pyhrf_jde_vem_analysis` provided by PyHRF. We can execute the analysis using several processors (`parallel`) because PyHRF uses the library `joblib`. The reader can found more details about this command and its parameters in the PyHRF documentation.

```
pyhrf_jde_vem_analysis [options] TR atlas_file \
                        paradigm_file bold_images

pyhrf_jde_vem_analysis \
  --estimate-hrf \
  --dt 1.25 \
  --hrf-duration 25.0 \
  --zero-constraint \
  --beta 1.0 \
  --hrf-hyperprior 1000 \
  --sigma-h 0.1 \
  --drifts-type cos \
  --parallel \
  --log-level WARNING \
  --output /home/jariasal/pyhrf \
  2.5 \
  {$HOME}/pyhrf/mask_parcellation/willard_3mm.nii \
  /tmp/tmpM3zBD5
  {$HOME}/data/bold.nii
```

PyHRF Analysis (Output)

We show in Fig. 9 the active parcels (A), the PPMs (B), and the estimated HRFs (C) generated by PyHRF for the motor task `Finger`. Reading the description given in [GSB+13], this task corresponds to finger tapping. Recall that PyHRF estimates a HRF for each parcel and generates a PPM for each condition.

We compared the output of PyHRF with the thresholded statistical maps shared on the site *Neurovault* (http://www.neurovault.org/images/307/) for the same dataset and same task (see Fig. 9). While the experimental paradigm is not optimized for JDE (standard block paradigm is not ideal to estimate different points of the HRF course), we obtained similar results to standard statistical analysis additionally providing the form of the HRF. As we can observe, at cut *z=60* both results (Fig. 9 B and D) are quite similar, showing an activation in the *supplementary motor area* and the *left primary sensorimotor cortex*.

Concluding Remarks

In this paper we presented PyHRF, a software to analyze fMRI data using a Joint Detection-Estimation (JDE) approach of the cerebral activity. It jointly detects cortical activation and estimates the Hemodynamic Response Function (HRF), in contrast to existing tools, that consider the HRF as constant over the brain and over subjects, thus aiming to improve the reliability of the results.

PyHRF is an open source software that has evolved rapidly over the last few years. As we showed, it allows to generate Posterior Probability Maps (PPMs) to describe the significance of the activation in each region of the brain. Moreover, PyHRF uses efficient estimation methods in order to provide a fast tool. Currently, the package does not provide finely tuned values for the parameters of the JDE model, leaving the user the cumbersome task of finding the best values for the estimation.

Since 2013, PyHRF has started to evolve to deal with functional Arterial Spin Labelling (fASL) [VWV+13] data, including a physiological prior to make the perfusion estimation more robust [FPVS+14] [FPFC15b]. A fast solution for fASL based on VEM was proposed in [FPFC15c], with similar results to the solution

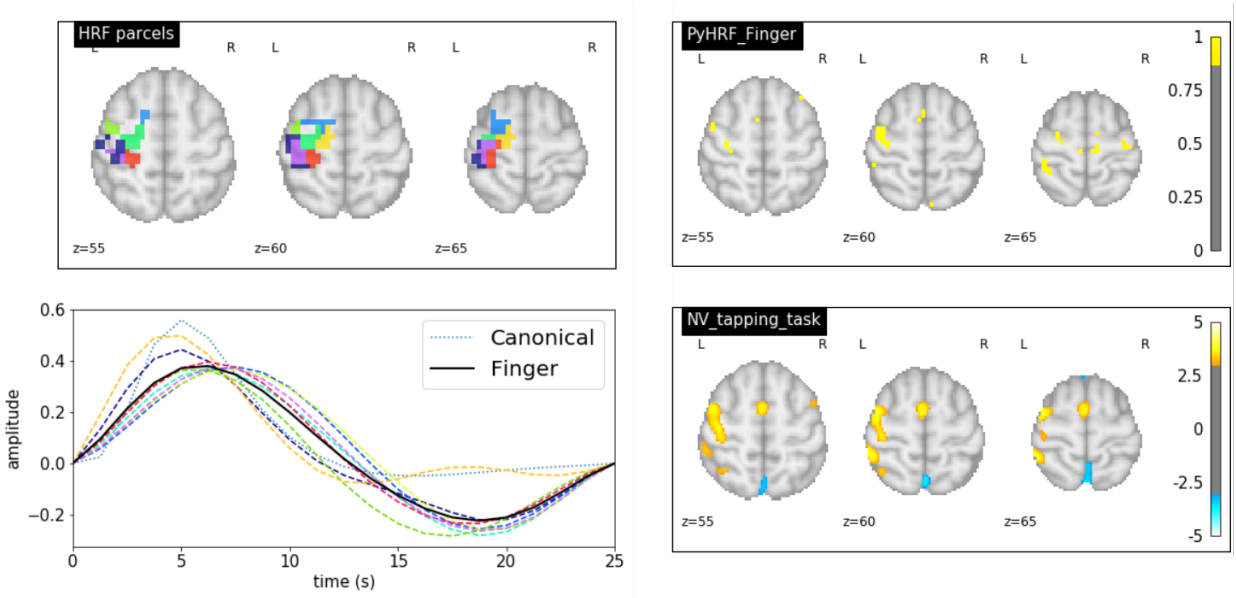


Fig. 9: *A) Active parcels, B) PPMs, and C) estimated HRFs generated by PyHRF for the dataset* `ds000114` *and the finger tapping task. D) Shows the thresholded statistical maps shared on the site* `Neurovault` *for the same dataset and task. The cut z=60 shows a high activation in the* supplementary motor area *and the* left primary sensorimotor cortex.

based on stochastic simulation techniques [FPFC15a]. Moreover, many efforts have been made in terms of image processing, user-friendliness and usability of the PyHRF tool to make it more easy to use by non experts and clinicians.

In the years to come, we plan to develop a light viewer to explore the results of PyHRF interactively. Moreover, we aspire to make the package compatible with Python 3 and extend its use to the analysis of fMRI data on small animals.

REFERENCES

[AVT+14] A. Attyé, M. Villien, F. Tahon, J. Warnking, O. Detante, and A. Krainik. Normalization of cerebral vasoreactivity using BOLD MRI after intravascular stenting. *Human Brain Mapping*, 35(4):1320–1324, apr 2014.

[BVC13] S. Badillo, T. Vincent, and P. Ciuciu. Group-level impacts of within- and between-subject hemodynamic variability in fMRI. *NeuroImage*, 82:433–448, nov 2013.

[CVF+13] L. Chaari, T. Vincent, F. Forbes, M. Dojat, and P. Ciuciu. Fast joint detection-estimation of evoked brain activity in event-related fMRI using a variational approach. *IEEE Trans. on Medical Imaging*, 32(5):821–837, May 2013.

[CVM+11] S. Cantin, M. Villien, O. Moreaud, I. Tropres, S. Keignart, E. Chipon, J.-F. Le Bas, J. Warnking, and A. Krainik. Impaired cerebral vasoreactivity to CO2 in Alzheimer's disease using BOLD fMRI. *NeuroImage*, 58(2):579–587, sep 2011.

[FGM+14] M. Fabiani, B. A. Gordon, E. L. Maclin, M. A. Pearson, C. R. Brumback-Peltz, K. A. Low, E. McAuley, B. P. Sutton, A. F. Kramer, and G. Gratton. Neurovascular coupling in normal aging: A combined optical, ERP and fMRI study. *NeuroImage*, 85:592–607, jan 2014.

[FPFC15a] A. Frau-Pascual, F. Forbes, and P. Ciuciu. Comparison of stochastic and variational solutions to ASL fMRI data analysis. In *Medical Image Computing and Computer-Assisted Intervention–MICCAI 2015*, pages 85–92. Springer, 2015.

[FPFC15b] A. Frau-Pascual, F. Forbes, and P. Ciuciu. Physiological models comparison for the analysis of ASL fMRI data. In *Biomedical Imaging (ISBI), 2015 IEEE 12th International Symposium on*, pages 1348–1351. IEEE, 2015.

[FPFC15c] A. Frau-Pascual, F. Forbes, and P. Ciuciu. Variational physiologically informed solution to hemodynamic and perfusion response estimation from ASL fMRI data. In *Pattern Recognition in NeuroImaging (PRNI), 2015 International Workshop on*, pages 57–60. IEEE, 2015.

[FPVS+14] A. Frau-Pascual, T. Vincent, J. Sloboda, P. Ciuciu, and F. Forbes. Physiologically informed Bayesian analysis of ASL fMRI data. In *Bayesian and grAphical Models for Biomedical Imaging*, pages 37–48. Springer, 2014.

[GSB+13] K. J. Gorgolewski, A. Storkey, M. E. Bastin, I. R. Whittle, J. M. Wardlaw, and C. R. Pernet. A test-retest fMRI dataset for motor, language and spatial attention functions. *GigaScience*, 2(1):6, 2013.

[HOM04] D. A. Handwerker, J. M. Ollinger, and D. Mark. Variation of BOLD hemodynamic responses across subjects and brain regions and their effects on statistical analyses. *Neuroimage*, 21:1639–1651, 2004.

[MIV+08] S. Makni, J. Idier, T. Vincent, B. Thirion, G. Dehaene-Lambertz, and P. Ciuciu. A fully Bayesian approach to the parcel-based detection-estimation of brain activity in fMRI. *Neuroimage*, 41(3):941–969, 2008.

[OLKT90] S. Ogawa, T. M. Lee, A. R. Kay, and D. W. Tank. Brain magnetic resonance imaging with contrast dependent on blood oxygenation. *Proceedings of the National Academy of Sciences*, 87(24):9868–9872, dec 1990.

[PJG+03] P. Ciuciu, J.-B. Poline, G. Marrelec, J. Idier, Ch. Pallier, and H. Benali. Unsupervised robust non-parametric estimation of the hemodynamic response function for any fMRI experiment. *IEEE Trans. Med. Imag.*, 22(10):1235–1251, Oct. 2003.

[RAM+15] J. Richiardi, A. Altmann, A.-C. Milazzo, C. Chang, M. M. Chakravarty, T. Banaschewski, G. J. Barker, A. L. W. Bokde, U. Bromberg, C. Buchel, P. Conrod, M. Fauth-Buhler, H. Flor, V. Frouin, J. Gallinat, H. Garavan, P. Gowland, A. Heinz, H. Lemaitre, K. F. Mann, J.-L. Martinot, F. Nees, T. Paus, Z. Pausova, M. Rietschel, T. W. Robbins, M. N. Smolka, R. Spanagel, A. Strohle, G. Schumann, M. Hawrylycz, J.-B. Poline, M. D. Greicius, L. Albrecht, C. Andrew, M. Arroyo, E. Artiges, S. Aydin, C. Bach, T. Banaschewski, A. Barbot, G. Barker, N. Boddaert, A. Bokde, Z. Bricaud, U. Bromberg, R. Bruehl, C. Buchel, A. Cachia, A. Cattrell, P. Conrod, P. Constant, J. Dalley, B. Decideur, S. Desrivieres, T. Fadai, H. Flor, V. Frouin, J. Gallinat, H. Garavan, F. G. Briand, P. Gowland, B. Heinrichs, A. Heinz, N. Heym, T. Hubner, J. Ireland, B. Ittermann, T. Jia, M. Lathrop, D. Lanzerath, C. Lawrence, H. Lemaitre, K. Ludemann, C. Macare, C. Mallik, J.-F. Mangin, K. Mann, J.-L. Martinot, E. Mennigen, F. Mesquita de Carvahlo, X. Mignon, R. Miranda, K. Muller, F. Nees, C. Nymberg, M.-L. Paillere, T. Paus, Z. Pausova, J.-B. Poline, L. Poustka, M. Rapp, G. Robert, J. Reuter, M. Rietschel, S. Ripke, T. Robbins, S. Rodehacke, J. Rogers, A. Romanowski, B. Ruggeri, C. Schmal, D. Schmidt, S. Schneider, M. Schumann, F. Schubert,

Y. Schwartz, M. Smolka, W. Sommer, R. Spanagel, C. Speiser, T. Spranger, A. Stedman, S. Steiner, D. Stephens, N. Strache, A. Strohle, M. Struve, N. Subramaniam, L. Topper, R. Whelan, S. Williams, J. Yacubian, M. Zilbovicius, C. P. Wong, S. Lubbe, L. Martinez-Medina, A. Fernandes, and A. Tahmasebi. Correlated gene expression supports synchronous activity in brain networks. *Science*, 348(6240):1241–1244, jun 2015.

[VBR+14] T. Vincent, S. Badillo, L. Risser, L. Chaari, C. Bakhous, F. Forbes, and P. Ciuciu. Flexible multivariate hemodynamics fMRI data analyses and simulations with PyHRF. *Frontiers in Neuroscience*, 8, apr 2014.

[VRC10] T. Vincent, L. Risser, and P. Ciuciu. Spatially adaptive mixture modeling for analysis of within-subject fMRI time series. *IEEE Trans. on Medical Imaging*, 29(4):1059–1074, April 2010.

[VWV+13] T. Vincent, J. Warnking, M. Villien, A. Krainik, P. Ciuciu, and F. Forbes. Bayesian Joint Detection-Estimation of cerebral vasoreactivity from ASL fMRI data. In *16th Proc. MICCAI*, volume 2, pages 616–623, Nagoya, Japan, September 2013.

SciSheets: Providing the Power of Programming With The Simplicity of Spreadsheets

Alicia Clark[§], Joseph L. Hellerstein[‡*]

https://www.youtube.com/watch?v=2-qCCR5r01A

Abstract—Digital spreadsheets are arguably the most pervasive environment for end user programming on the planet. Although spreadsheets simplify many calculations, they fail to address requirements for expressivity, reuse, complex data, and performance. SciSheets (from "scientific spreadsheets") is an open source project that provides novel features to address these requirements: (1) formulas can be arbitrary Python scripts as well as expressions (*formula scripts*), which addresses expressivity by allowing calculations to be written as algorithms; (2) spreadsheets can be exported as functions in a Python module (*function export*), which addresses reuse since exported codes can be reused in formulas and/or by external programs and improves performance since calculations can execute in a low overhead environment; and (3) tables can have columns that are themselves tables (*subtables*), which addresses complex data such as representing hierarchically structured data and n-to-m relationships. Our future directions include refinements to subtables, github integration, and plotting. At present, SciSheets can do robust demos, but it is not yet beta code.

Index Terms—software engineering

1. Introduction

Digital spreadsheets are the "killer app" that ushered in the PC revolution. This is largely because spreadsheets provide a conceptually simple way to do calculations that (a) closely associates data with the calculations that produce the data and (b) avoids the mental burdens of programming such as control flow, data dependencies, and data structures. Over 800M professionals author spreadsheet formulas as part of their work [MODE2017], which is over 50 times the number of software developers world wide [THIB2013].

We categorize spreadsheet users as follows:

- **Novices** want to evaluate equations, but they do not have the prior programming experience necessary to create reusable functions and longer scripts. Spreadsheet formulas work well for Novices since: (a) they can ignore data dependencies; (b) they can avoid flow control by using "copy" and "paste" for iteration; and (c) data structures are "visual" (e.g., rectangular blocks).
- **Scripters** feel comfortable with expressing calculations algorithmically using `for` and `if` statements; and they can use simple data structures such as lists and `pandas DataFrames`. However, Scripters rarely encapsulate code into functions, preferring "copy" and "paste" to get reuse.
- **Programmers** know about sophisticated data structures, modularization, reuse, and testing.

Our experience is primarily with technical users such as scientists. Most commonly, we encounter Novices and Scripters with limited prior programming experience. We do not expect these groups of users to take advantage of spreadsheet macro capabilities (e.g., Visual Basic for Microsoft Excel or AppScript in Google Sheets); we anticipate this functionality to be taken advantage of only by Programmers.

Based on this experience, we find existing spreadsheets lack several key requirements. First, they lack the **expressivity requirement** in that (a) they only permit a limited set of functions to be used in formulas (e.g., so that static dependency checking can be done); and (b) they only support formulas that are expressions, not scripts. In particular, the latter means that Scripters cannot express calculations as algorithms, and Novices cannot write linear workflows to articulate a computational recipe. A second consideration is the **reuse requirement**. Today, it is impossible to reuse spreadsheet formulas in other spreadsheet formulas or in software systems. Third, current spreadsheet systems cannot handle the **complex data requirement**, such as manipulating data that are hierarchically structured or data that have n-to-m relationships. Finally, existing spreadsheets cannot address the **performance requirement** in that spreadsheets scale poorly with the size of data and the number of formulas.

Academic computer science has recognized the growing importance of end-user programming (EUP) [BURN2009]. Even so, there is little academic literature on spreadsheets, arguably the most pervasive EUP environment on the planet. [MCLU2006] discusses object oriented spreadsheets that introduce a sophisticated object model, but the complexity of this proposal is unlikely to appeal to Novices. [JONE2003] describes a way that users can implement functions within a spreadsheet to get reuse. However, the approach imposes a significant burden on the user, and does not address reuse of formulas outside the spreadsheet environment. Industry has had significant interest in

§ Department of Mechanical Engineering, University of Washington
∗ Corresponding author: joseph.hellerstein@gmail.com
‡ eScience Institute and School of Computer Science, University of Washington. *This work was made possible by the Moore/Sloan Data Science Environments Project at the University of Washington supported by grants from the Gordon and Betty Moore Foundation (Award #3835) and the Alfred P. Sloan Foundation (Award #2013-10-29).*

innovating spreadsheets. Google Fusion Tables [GONZ2010] and the "Tables" feature of Microsoft Excel [MICROSOF] use column formulas to avoid a common source of error, copying formulas as rows are added/deleted from a table. The Pyspread [PYSPREAD] project uses Python as the formula language, but Pyspread formulas cannot be Python scripts. A more radical approach is taken by Stencila [STENCILA], a document system that provides ways to execute code that updates tables (an approach that is in the same spirit as Jupyter Notebooks [PERE2015]). Stencila supports a variety of languages including JavaScript, Python, and SQL. However, Stencila lacks features that spreadsheet users expect: (a) closely associating data with the calculations that produce the data and (b) avoiding considerations of data dependencies in calculations.

This paper introduces SciSheets [SCISHEETS], a new spreadsheet system with the objective of providing the power of programming with the simplicity of spreadsheets. The name SciSheets is a contraction of the phrase "Scientific Spreadsheet", a nod to the users who motivated the requirements that we address. That said, our target users are more broadly technical professionals who do complex calculations on structured data. We use the term **scisheet** to refer to a SciSheets spreadsheet. We note in passing that our focus for scisheets is on calculations, not document processing features such as formatting and drawing figures.

SciSheets addresses the above requirements by introducing several novel features.

- *Formula Scripts.* Scisheet formulas can be arbitrary Python scripts as well as expressions. This addresses expressivity by allowing calculations to be written as algorithms.
- *Function Export.* Scisheets can be exported as functions in a Python module. This addresses reuse since exported codes can be reused in SciSheets formulas and/or by external programs. Further, performance is improved by the export feature since calculations execute in a low overhead environment.
- *Subtables.* Tables can have columns that are themselves tables (columns within columns). This addresses the complex data requirement, such as representing hierarhically structured data and n-to-m relationships.

The remainder of the paper is organized as follows. Section 2 describes the requirements that we consider, and Section 3 details the SciSheets features that address these requirements. The design of SciSheets is discussed in Section 4, and Section 5 discusses features planned for SciSheets. Our conclusions are contained in Section 6.

2. Requirements

This section presents examples that motivate the requirements of expressivity, reuse, and complex data.

Our first example is drawn from biochemistry labs studying enzyme mediated chemical reactions. Commonly, the Michaelis-Menten [BERG2002] Model of enzyme activity is used in which there is a single chemical species, called the substrate, that interacts with the enzyme to produce a new chemical species (the product). Two properties of enzymes are of much interest: the maximum reaction rate, denoted by V_{MAX}, and the concentration K_M of substrate that achieves a reaction rate equal to half of V_{MAX}.

To perform the Michaelis-Menten analysis, laboratory data are collected for different values of the substrate concentrations S and

	A	B	C	D	E	F	G	H
1	S	V	INV_S	INV_V	INTERCEPT	SLOPE	V_MAX	K_M
2	0.01	0.11	100.00	9.09	4.357	0.047	0.229	0.011
3	0.05	0.19	20.00	5.26				
4	0.12	0.21	8.33	4.76				
5	0.20	0.22	5.00	4.55				
6	0.50	0.21	2.00	4.76				
7	1.00	0.24	1.00	4.17				

Fig. 1: *Data view for an Excel spreadsheet that calculates Michaelis-Menten Parameters.*

	A	B	C	D	E	F	G	H
1	S	V	INV_S	INV_V	INTERCEPT	SLOPE	V_MAX	K_M
2	0.01	0.11	=1/A2	=1/B2	=INTERCEPT(D2:D7,C2:C7)	=SLOPE(D2:D7,C2:C7)	=1/E2	=F2*G2
3	0.05	0.19	=1/A3	=1/B3				
4	0.12	0.21	=1/A4	=1/B4				
5	0.20	0.22	=1/A5	=1/B5				
6	0.50	0.21	=1/A6	=1/B6				
7	1.00	0.24	=1/A7	=1/B7				

Fig. 2: *Formulas used in Fig. 1.*

associated reaction rates V. Then, a calculation is done to obtain the parameters V_{MAX} and K_M using the following recipe.

1) Compute $1/S$ and $1/V$, the inverses of S and V.
2) Compute the intercept and slope of the regression of $1/V$ on $1/S$.
3) Calculate V_{MAX} and K_M from the intercept and slope.

Fig. 1 shows an Excel spreadsheet that implements this recipe with column names that correspond to the variables in the recipe. Fig. 2 displays the formulas that perform these calculations. Readability can be improved by using column formulas (e.g., as in Fusion Tables). However, two problems remain. Novices cannot *explicitly* articulate the computational recipe; rather, the recipe is implicit in the order of the columns. Even more serious, there is no way to reuse these formulas in other formulas (other than error-prone copy-and-paste), and there is no way to reuse formulas in an external program.

We consider a second example to illustrate problems with handling non-trivial data relationships in spreadsheets. Fig. 3 displays data that a university might have for students in two departments in the School of Engineering, Computer Science & Engineering (CSE) and Biology. The data are organized into two tables (`CSE` and `Biology`) grouped under `Engineering`, with separate columns for student identifiers and grades. These tables are adjacent to each other to facilitate comparisons between students. However, the tables are independent of each other in

	A	B	C	D	E	F
1	**Engineering**					
2	**CSE**			**Biology**		
3	**StudentNo**	**GPA**		**StudentNo**	**Track**	**GPA**
4	C1113	3.9		B1414	A	3.4
5	C1163	3.5		B1830	B	2.3
6	C1344	3.3		B1716	C	3.7
7	C1711	3.9				
8	C1579	2.8				

Fig. 3: *Illustrative example of student grade data from two departments in the School of Engineering.* `CSE` *and* `Biology` *are separate tables that are grouped together for convenience of analysis. In existing spreadsheet systems, users cannot perform row operations such as insert, delete, and/or hide on one subtable without affecting the other subtable.*

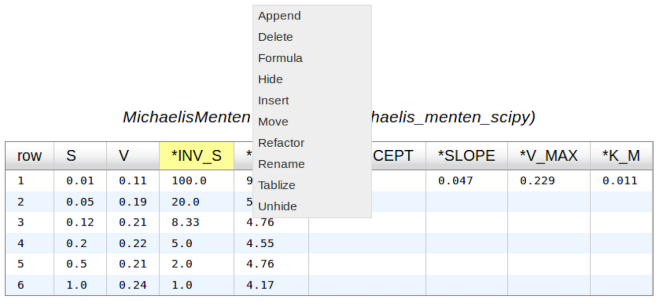


Fig. 4: *Column popup menu in a scisheet for the Michaelis-Menten calculation.*

that we should be able to insert, delete, and hide rows in one table without affecting the other table. Unfortunately, existing spreadsheet systems do not handle this well; inserting, deleting, or hiding a row in one table affects every table that overlaps that row in the spreadsheet. Note that arranging the tables vertically does not help since the problem becomes inserting, deleting, and hiding columns. We could arrange the tables in a diagonal, but this makes it difficult to make visual comparisons between tables.

3. Features

This section describes SciSheets features that address the requirements of expressivity, reuse, complex data, and performance. We begin with a discussion of the SciSheets user interface in Section 3.1. Then, Sections 3.2, 3.3, and 3.4 present formula scripts (which addresses expressivity), function export (which addresses reuse and performance), and subtables (which addresses complex data) respectively.

3.1 User Interface

Fig. 4 displays a scisheet that performs the Michaelis-Menten calculations as we did in Fig. 1. Note that columns containing a formula have a name annotated with an `*`.

A scisheet has the familiar tabular structure of a spreadsheet. However, unlike existing spreadsheets, SciSheets knows about the **elements of a scisheet**: tables, columns, rows, and cells. In SciSheets, there are two types of columns. Data columns contain data values; subtable columns contain a table. The name of a data column is a Python variable that can be referenced in formulas. These **column variables** are `numpy Arrays`. This means that formulas can be written using column names to express vector calculation using a rich set of operators that properly handle missing data (e.g., using `NaN` values).

SciSheets users interact directly with the scisheet element appropriate for the desired action. A left click on a scisheet element results in a popup menu. For example, in Fig. 4 we see the column popup for `INV_S`. Users select an item from the popup, and this may in turn present additional menus. The popup menus for row, column, and table have common items for insert, delete, hide/unhide. Columns additionally have a formula item. The scisheet popup has items for saving and renaming the scisheet as well as undoing/redoing operations on the scisheet. The cell popup is an editor for the value in the cell.

Fig. 5 displays the results of selecting the `formula` item from the popup menu in Fig. 4 for the column `INV_S`. A simple line editor is displayed. The formula is an expression that references the column `S`.

Fig. 5: *Formula for computing the inverse of the input value S.*

INV_V

```
1 import scipy.stats as ss
2 INV_S = np.round(1/S, 2)
3 INV_V = np.round(1/V, 2)
4 SLOPE, INTERCEPT, _, _, _ = ss.linregress(INV_S, INV_V)
5 V_MAX = 1/INTERCEPT
6 K_M = SLOPE*V_MAX
7
```

Fig. 6: *Formula for the complete calculation of V_{MAX} and K_M. The formula is a simple script, allowing a Novice to see exactly how the data in the scisheet are produced.*

3.2 Formula Scripts and Formula Evaluation

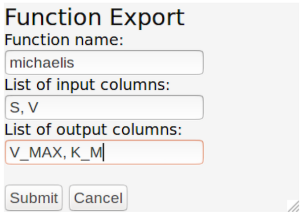
SciSheets allows formulas to be scripts with arbitrary Python statements. For example, Fig. 6 displays a script that contains the entire computational recipe for the Michaelis-Menten calculation described in Section 2. This capability greatly increases the ability of spreadsheet users to describe and document their calculations.

The formula scripts feature has a significant implication on how formulas are evaluated. Since a formula may contain arbitrary Python codes including `eval` expressions, we cannot use static dependency analysis to determine data dependencies. Thus, formula evaluation is done iteratively. But how many times must this iteration be done?

Consider an evaluation of N formula columns assuming that there are no circular references or other anomalies in the formulas. Then, at most N iterations are needed for convergence since on each iteration at least one column variable is assigned its final value. If after N iterations, there is an exception, (e.g., a column variable does not have a value assigned), this is reported to the user since there is likely an error in the formulas. Otherwise, the scisheet is updated with the new values of the column variables. Actually, we can do better than this since if the values of column variables converge after loop iteration $M < N$ (and there is no exception), then formula evaluation stops. We refer to the above workflow as the **formula evaluation loop**.

SciSheets augments the formula evaluation loop by providing users with the opportunity to specify two additional formulas. The **prologue formula** is executed once at the beginning of formula evaluation; the **epilogue formula** is executed once at the end of formula evaluation. These formulas provide a way to do high overhead operations in a one-shot manner, a feature that assists the performance requirement. For example, a user may have a prologue formula that reads a file (e.g., to initialize input values in a table) at the beginning of the calculation, and an epilogue formula that writes results at the end of the calculation. Prologue and epilogue formulas are modified through the scisheet popup menu.

At present, variable names have a global scope within the scisheet. This is often a desirable feature. For example, in Fig. 6, values computed in one column formula are assigned to another column. However, as discussed in Section 5, there are some interesting use cases for having subtable name scoping, a feature that we are implementing.



MichaelisMenten (Table File: michaelis_menten_scipy)

row	S	V	*INV_S	*INV_V	*INTERCEPT	*SLOPE	*V_MAX	*K_M
1	0.01	0.11	100.0	9.09	4.358	0.047	0.229	0.011
2	0.05	0.19	20.0	5.26				
3	0.12	0.21	8.33	4.76				
4	0.2	0.22	5.0	4.55				
5	0.5	0.21	2.0	4.76				
6	1.0	0.24	1.0	4.17				

Fig. 7: *Menu to export a scisheet as a function in a Python module.*

3.3. Function Export

A scisheet can be exported as a function in a Python module. This feature addresses the reuse requirement since exported codes can be used in scisheet formulas and/or external programs. The export feature also addresses the performance requirement since executing standalone code eliminates many overheads.

At first glance, it may seem that being able to export a scisheet as a function is in conflict with an appealing feature of spreadsheets--that data are closely associated with the calculations that produce the data. It is a central concern of SciSheets to preserve this feature of spreadsheets. Thus, users specify formulas for columns and/or for table prologues and epilogues without regard to how code might be exported. SciSheets automatically structures code for export.

Fig. 7 displays the scisheet popup menu for function export. The user sees a menu with entries for the function name, inputs (a list of column names), and outputs (a list of column names).

Function export produces two files. The first is the Python module containing the exported function. The second is a Python file containing a test for the exported function.

We begin with the first file. The code in this file is structured into several sections:

- Function definition and setup
- Formula evaluation
- Function close

The function definition and setup contain the function definition, imports, and the scisheet prologue formula. Note that the prologue formula is a convenient place to import Python packages.

```
# Function definition
def michaelis(S, V):
  from scisheets.core import api as api
  s = api.APIPlugin('michaelis.scish')
  s.initialize()
  _table = s.getTable()
  # Prologue
  s.controller.startBlock('Prologue')
  # Begin Prologue
  import math as mt
  import numpy as np
  from os import listdir
  from os.path import isfile, join
  import pandas as pd
  import scipy as sp
  from numpy import nan  # Must follow sympy import
  # End Prologue
  s.controller.endBlock()
```

In the above code, the imported package `scisheets.core.api` contains the SciSheets runtime. The object `s` is constructed using a serialization of the scisheet that is written at the time of function export. scisheets are serialized in a JSON format to a file that has the extension `.scish`.

We see that prologue formulas can be lengthy scripts. For example, one scisheet developed with a plant biologist has a prologue formula with over fifty statements. As such, it is essential that syntax and execution errors are localized to a line within the script. We refer to this as the **script debuggability requirement**. SciSheets handles this requirement by using the paired statements `s.controller.startBlock('Prologue')` and `s.controller.endBlock()`. These statements "bracket" the script so that if an exception occurs, SciSheets can compute the line number within the script for that exception.

Next, we consider the formula evaluation loop. Below is the code that is generated for the beginning of the loop and the evaluation of the formula for `INV_S`.

```
s.controller.initializeLoop()
while not s.controller.isTerminateLoop():
  s.controller.startAnIteration()
  # Formula evaluation blocks
  try:
    # Column INV_S
    s.controller.startBlock('INV_S')
    INV_S = 1/S
    s.controller.endBlock()
    INV_S = s.coerceValues('INV_S', INV_S)
  except Exception as exc:
    s.controller.exceptionForBlock(exc)
```

`s.controller.initializeLoop()` snapshots column variables. `s.controller.isTerminateLoop()` counts loop iterations, looks for convergence of column variables, and checks to see if the last loop iteration has an exception. Each formula column has a pair of `try` and `except` statements that execute the formula and record exceptions. Note that loop execution continues even if there is an exception for one or more formula columns. This is done to handle situations in which formula columns are *not* ordered according to their data dependencies.

Last, there is the function close. The occurrence of an exception in the formula evaluation loop causes an exception with the line number in the formula in which the (last) exception occurred. If there is no exception, then the epilogue formula is executed, and the output values of the function are returned (assuming there is no exception in the epilogue formula).

```
if s.controller.getException() is not None:
  raise Exception(s.controller.formatError(
      is_absolute_linenumber=True))
s.controller.startBlock('Epilogue')
# Epilogue (empty)
s.controller.endBlock()
return V_MAX,K_M
```

The second file produced by SciSheets function export contains test code. Test code makes use of `unittest` with a `setUp` method that assigns `self.s` the value of a SciSheets runtime object.

```
def testBasics(self):
  S = self.s.getColumnValue('S')
  V = self.s.getColumnValue('V')
  V_MAX,K_M = michaelis(S,V)
```

row	CSV_FILE	*K_M	V_MAX
1	Glu.csv	[5.179]	[0.568]
2	LL-DAP.csv	[0.929]	[23.81]
3	THDPA.csv	[0.011]	[0.229]

Fig. 8: *A scisheet that processes many CSV files.*

K_M

```
# Compute K_M and V_MAX for each CSV file
K_M = []
V_MAX = []
for csv_file in CSV_FILE:
  df = pd.read_csv(join(PATH, csv_file))
  s_val = df['S']
  v_val = df['V']
  v_max, k_m = michaelis(s_val, v_val)
  K_M.append(k_m)
  V_MAX.append(v_max)
```

Fig. 9: *Column formula for* `K_M` *in Fig. 8 that is a script to process a list of CSV files.*

```
self.assertTrue(
    self.s.compareToColumnValues('V_MAX', V_MAX))
self.assertTrue(
    self.s.compareToColumnValues('K_M', K_M))
```

The above test compares the results of running the exported function `michaelis` on the input columns `S` and `V` with the values of output columns `V_MAX` and `K_M`.

The combination of the features function export and formula scripts is extremely powerful. To see this, consider a common pain point with spreadsheets - doing the same computation for different data sets. For example, the Michaelis-Menten calculation in Fig. 1 needs to be done for data collected from many experiments that are stored in several comma separated variable (CSV) files. Fig. 8 displays a scisheet that does the Michaelis-Menten calculation for the list of CSV files in the column `CSV_FILE`. (This list is computed by the prologue formula based on the contents of the current directory.) Fig. 9 displays a script that reuses the `michaelis` function exported previously to compute values for `K_M` and `V_MAX`. Thus, whenever new CSV files are available, `K_M` and `V_MAX` are calculated without changing the scisheet in Fig. 8.

3.4. Subtables

Subtables provide a way for SciSheets to deal with complex data by having tables nested within tables.

	Engineering						
	[CSE]			[Biology]			
row	row	*ScholarID	GradePtAvg	row	*StudentNo	Track	GPA
	1	C1113	3.9	1	B1414	A	3.4
	2	C1163	3.5	2	B1830	B	2.3
	3	C1344	3.3	3	B1716	C	3.7
	4	C1711	3.9				
	5	C1579	2.8				

Fig. 10: *The table* `Engineering` *has two subtables* `CSE` *and* `Biology`*. The subtables are independent of one another, which is indicated by the square brackets around their names and the presence of separate* `row` *columns.*

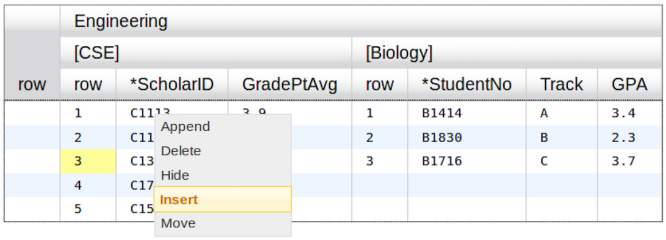


Fig. 11: *Menu to insert a row in one subtable. The menu is accessed by left-clicking on the "3" cell in the column labelled "row" in the CSE subtable.*

	Engineering						
	[CSE]			[Biology]			
row	row	*ScholarID	GradePtAvg	row	*StudentNo	Track	GPA
	1	C1113	3.9	1	B1414	A	3.4
	2	C1163	3.5	2	B1830	B	2.3
	3			3	B1716	C	3.7
	4	C1344	3.3				
	5	C1711	3.9				
	6	C1579	2.8				

Fig. 12: *Result of inserting a row in the* `CSE` *subtable. Note that the* `Biology` *substable is unchanged.*

We illustrate this by revisiting the example in Fig. 3. Fig. 10 displays a scisheet for these data in which `CSE` and `Biology` are independent subtables (indicated by the square brackets around the names of the subtables). Note that there is a column named `row` for each subtable since the rows of `CSE` are independent of the rows of `Biology`.

Recall that in Section 2 we could not insert a row into `CSE` without also inserting a row into `Biology`. SciSheets addresses this requirement by providing a separate row popup for each subtable. This is shown in Fig. 11 where there is a popup for row 3 of `CSE`. The result of selecting `insert` is displayed in Fig. 12. Note that the `Biology` subtable is not modified when there is an insert into `CSE`.

4. Design

SciSheets uses a client-server design. The client runs in the browser using HTML and JavaScript; the server runs Python using the Django framework [DJANGOPR]. This design provides a zero install deployment, and leverages the rapid pace of innovation in browser technologies.

Our strategy has been to limit the scope of the client code to presentation and handling end-user interactions. When the client requires data from the server to perform end-user interactions (e.g., populate a list of saved scisheets), the client uses AJAX calls. The client also makes use of several JavaScript packages including JQuery [JQUERYPR], YUI DataTable [YUIDATAT], and JQueryLinedText [JQUERYLI].

The SciSheets server handles the details of user requests, which also requires maintaining the data associated with scisheets. Fig 13 displays the core classes used in the SciSheets server. Core classes have several required methods. For example, the `copy` method makes a copy of the object for which it is invoked. To do this, the object calls the `copy` method of its parent class as well, and this is done recursively. Further, the object must call the `copy` method for core objects that are in its instance variables, such as

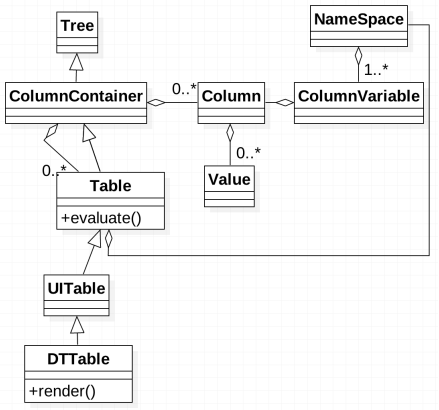


Fig. 13: *SciSheets core classes.*

`ColumnContainer` which has the instance variable `columns` that contains a list of `Column` objects. Other examples of required methods are `isEquivalent`, which tests if two objects have the same values of instance variables, and `deserialize`, which creates objects based on data serialized in a JSON structure.

Next, we describe the classes in Fig. 13. `Tree` implements a tree that is used to express hierarchical relationships such as between `Table` and `Column` objects. `Tree` also provides a mapping between the name of the scisheet element and the object associated with the name (e.g., to handle user requests). `ColumnContainer` manages a collections of `Table` and `Column` objects. `Column` is a container of data values. `Table` knows about rows, and it does formula evaluation using `evaluate()`. `UITable` handles user requests (e.g., renaming a column and inserting a row) in a way that is independent of the client implementation. `DTTable` provides client specific services, such as rendering tables into HTML using `render()`.

The classes `NameSpace` (a Python namespace) and `ColumnVariable` are at the center of formula evaluation. The `evaluate()` method in `Table` generates Python code that is executed in a Python namespace. The SciSheets runtime creates an instance of `ColumnVariable` for each `Column` in the scisheet being evaluated. `ColumnVariable` puts the name of its corresponding `Column` into the namespace, and assigns to this name a `numpy Array` that is populated with the values of the `Column`.

Last, we consider performance. There are two common causes of poor performance in the current implementation of SciSheets. The first relates to data size. At present, SciSheets embeds data with the HTML document that is rendered by the browser. We will address this by downloading data on demand and caching data locally.

The second cause of poor performance is having many iterations of the formula evaluation loop. If there is more than one formula column, then the best case is to evaluate each formula column twice. The first execution produces the desired result (e.g., if the formula columns are in order of their data dependencies); the second execution confirms that the result has converged. Some efficiencies can be gained by using the prologue and epilogue features for one-shot execution of high overhead operations (e.g., file I/O). In addition, we are exploring the extent to which SciSheets can automatically detect if static dependency checking can be used so that formula evaluation is done only once.

Clearly, performance can be improved by reducing the number of formula columns since this reduces the maximum number of iterations of the formulation evaluation loop. SciSheets supports this strategy by permitting formulas to be scripts. This is a reasonable strategy for a Scripter, but it may work poorly for a Novice who is unaware of data dependencies.

5. Future Work

This section describes several features that are under development.

5.1 Subtable Name Scoping

This feature addresses the reuse requirement. Today, spreadsheet users typically employ copy-and-paste to reuse formulas. This approach has many drawbacks. First, it is error prone since there are often mistakes as to what is copied and where it is pasted. Second, fixing bugs in formulas requires repeating the same error prone copy-and-paste.

It turns out that a modest change to the subtable feature can provide a robust approach to reuse through copy-and-paste. This change is to have a subtable define a name scope. This means that the same column name can be present in two different subtables since these names are in different scopes.

We illustrate the benefits of subtable name scoping. Consider Fig. 10 with the subtables `CSE` and `Biology`. Suppose that the column `GradePtAvg` in `CSE` is renamed to `GPA` so that both `CSE` and `Biology` have a column named `GPA`. Now, consider adding the column `TypicalGPA` to both subtables; this column will have a formula that computes the mean value of `GPA`. The approach would be as follows:

1) Add the column `TypicalGPA` to `CSE`.
2) Create the formula `np.mean(GPA)` in `TypicalGPA`. This formula will compute the mean of the values of the `GPA` column in the `CSE` subtable (because of subtable name scoping).
3) Copy the column `TypicalGPA` to subtable `Biology`. Because of subtable name scoping, the formula `np.mean(GPA)` will reference the column `GPA` in `Biology`, and so compute the mean of the values of `GPA` in the `Biology` subtable.

Now suppose that we want to change the calculation of `TypicalGPA` to be the median instead of the mean. This is handled as follows:

1) The user edits the formula for the column `TypicalGPA` in subtable `CSE`, changing the formula to `np.median(GPA)`.
2) SciSheets responds by asking if the user wants the copies of this formula to be updated as well.
3) The user answers "yes", and the formula is changed for `TypicalGPA` in subtable `Biology`.

Note that we would have the same result in the above procedure if the user had in step (1) modified the `Biology` subtable.

5.2 Github Integration

A common problem with spreadsheets is that calculations are difficult to reproduce because some steps are manual (e.g., menu interactions). Additionally, it can be difficult to reproduce a spreadsheet due to the presence of errors. We refer to this as the **reproducibility requirement**. Version control is an integral part of reproducibility. Today, a spreadsheet file as a whole can be

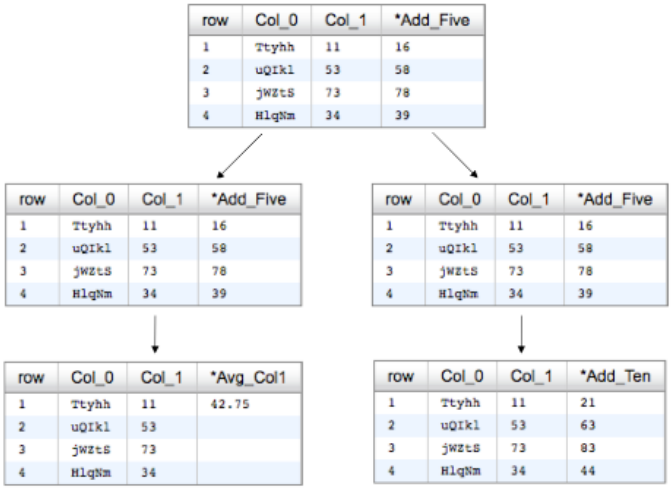


Fig. 14: *Mockup showing how a scisheet can be split into two branches (e.g., for testing and/or feature exploration).*

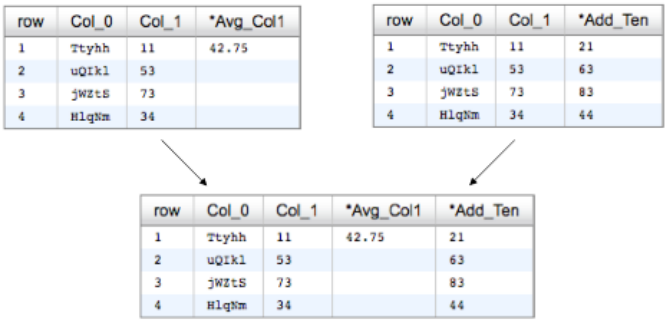


Fig. 15: *Mockup displaying two scisheets can be merged (assuming no merge conflicts).*

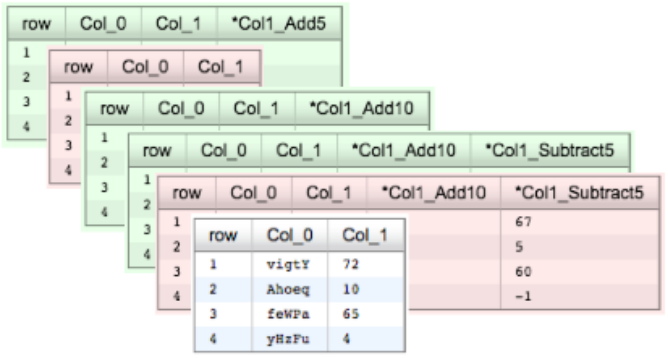


Fig. 16: *Mockup visualization of the change history of a scisheet. The versions in green show when columns have been added; the versions in red show when columns have been removed.*

version controlled, but this granularity is too coarse. More detailed version control can be done manually. However, this is error prone, especially in a collaborative environment. One automated approach is a revision history, such as Google Sheets. However, this technique fails to record the sequence in which changes were made, by whom, and for what reason.

The method of serialization used in SciSheets lends itself well to `github` integration. Scisheets are serialized as JSON files with separate lines used for data, formulas, and structural relationships between columns, tables, and the scisheet. Although the structural relationships have a complex representation, it does seem that SciSheets can be integrated with the line oriented version control of `github`.

We are in the process of designing an integration of SciSheets with `github` that is natural for Novices and Scripters. The scope includes the following use cases:

- **Branching.** Users should be able to create branches to explore new calculations and features in a scisheet. Fig. 14 shows how a scisheet can be split into two branches. As with branching for software teams, branching with a spreadsheet will allow collaborators to work on their part of the project without affecting the work of others.
- **Merging.** Users will be able to utilize the existing `github` strategies for merging documents. In addition, we intend to develop a visual way for users to detect and resolve merge conflicts. Fig. 15 illustrates how two scisheets can be merged. Our thinking is that name conflicts will be handled in a manner similar to that used in `pandas` with `join` operations. Our implementation will likely be similar to the `nbdime` package developed for merging and differencing Jupyter notebooks [NBDIME].
- **Differencing.** Users will be able to review the history of `git commit` operations. Fig. 16 displays a mockup of a visualization of the history of a scisheet. The user will be able to select any point in history (similar to `git checkout`). This functionality will allow collaborators to gain a greater understanding of changes made.

5.3 Plotting

At present, SciSheets does not support plotting. However, there is clearly a **plotting requirement** for any reasonable spreadsheet system. Our approach to plotting will most likely be to leverage the `bokeh` package [BOKEHPRO] since it provides a convenient way to generate HTML and JavaScript for plots that can be embedded into HTML documents. Our vision is to make `plot` a function that can be used in a formula. A *plot* column will have its cells rendered as HTML.

6. Conclusions

SciSheets is a new spreadsheet system with the guiding principle of providing the power of programming with the simplicity of spreadsheets. Our target users are technical professionals who do complex calculations on structured data.

SciSheets addresses several requirements that are not handled in existing spreadsheet systems, especially the requirements of expressivity, reuse, complex data, and performance. SciSheets addresses these requirements by introducing several novel features.

- *Formula Scripts.* Scisheet formulas can be Python scripts, not just expressions. This addresses expressivity by allowing calculations to be written as algorithms.
- *Function Export.* Scisheets can be exported as functions in a Python module. This addresses reuse since exported codes can be reused in SciSheets formulas and/or by external programs. Further, performance is improved by the export feature since calculations execute in a low overhead environment.

Requirement	SciSheets Feature
• Expressivity	• Python formulas • Formula scripts
• Reuse	• Function export • *Subtable name scoping*
• Complex Data	• Subtables
• Performance	• Function export • Prologue, Epilogue • *Load data on demand* • *Conditional static dependency checking*
• Plotting	• *Embed bokeh components*
• Script Debuggability	• Localized exceptions
• Reproducibility	• `github` *integration*

TABLE 1: *Summary of requirements and SciSheets features that address these requirements. Features in italics are planned but not yet implemented.*

- *Subtables.* Tables can have columns that are themselves tables (columns within columns). This addresses the complex data requirement, such as representing n-to-m relationships.

Table 1 displays a comprehensive list of the requirements we plan to address and the corresponding SciSheets features.

One goal for SciSheets is to make users more productive with their existing workflows for developing and evaluating formulas. However, we also hope that SciSheets becomes a vehicle for elevating the skills of users, making Novices into Scripters and Scripters into Programmers.

At present, SciSheets is capable of doing robust demos. Some work remains to create a beta. We are exploring possible deployment vehicles. For example, rather than having SciSheets be a standalone tool, another possibility is integration with Jupyter notebooks.

REFERENCES

[BERG2002] Berg, Jermey et al. *Biochemistry*, W H Freeman, 2002.
[BOKEHPRO] Bokeh Project. http://bokeh.pydata.org/.
[BURN2009] Burnett, M. *What is end-user software engineering and why does it matter?*, Lecture Notes in Computer Science, 2009
[DJANGOPR] Django Project. http://www.djangoproject.com.
[GONZ2010] *Google Fusion Tables: Web-Centered Data Management and Collaboration*, Hector Gonzalez et al., SIGMOD, 2010.
[JONE2003] Jones, S., Blackwell, A., and Burnett, M. i *A user-centred approach to functions in excel*, SIGPLAN Notices, 2003.
[JQUERYLI] JQueryLinedText. https://github.com/aw20/JQueryLinedText.
[JQUERYPR] JQuery Package. https://jquery.com/.
[MCCU2006] McCutchen, M., Itzhaky, S., and Jackson, D. *Object spreadsheets: a new computational model for end-user development of data-centric web applications*, Proceedings of the 2016 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, 2006.
[MICROSOF] Microsoft Corporation. *Overview of Excel tables*, https://support.office.com/en-us/article/Overview-of-Excel-tables-7ab0bb7d-3a9e-4b56-a3c9-6c94334e492c.
[MODE2017] *MODELOFF - Financial Modeling World Championships*, http://www.modeloff.com/the-legend/.
[NBDIME] *nbdime*, https://github.com/jupyter/nbdime.
[PERE2015] Perez, Fernando and Branger, Brian. *Project Jupyter: Computational Narratives as the Engine of Collaborative Data Science*, http://archive.iPython.org/JupyterGrantNarrative-2015.pdf.
[PYSPREAD] Manns, M. *PYSPREAD*, http://github.com/manns/pyspread.
[SCISHEET] *SciSheets*, https://github.com/ScienceStacks/SciSheets.
[STENCILA] *Stencila*, https://stenci.la/.
[THIB2013] Thibodeau, Patrick. *India to overtake U.S. on number of developers by 2017*, COMPUTERWORLD, Jul 10, 2013.
[YUIDATAT] Yahoo User Interface DataTable. https://yuilibrary.com/yui/docs/datatable/.

The Sacred Infrastructure for Computational Research

Klaus Greff[‡§¶*], Aaron Klein[‖], Martin Chovanec[**], Frank Hutter[‖], Jürgen Schmidhuber[‡§¶]

https://www.youtube.com/watch?v=qqg7RO0o1OE

Abstract—We present a toolchain for computational research consisting of Sacred and two supporting tools. Sacred is an open source Python framework which aims to provide basic infrastructure for running computational experiments independent of the methods and libraries used. Instead, it focuses on solving universal everyday problems, such as managing configurations, reproducing results, and bookkeeping. Moreover, it provides an extensible basis for other tools, two of which we present here: Labwatch helps with tuning hyperparameters, and Sacredboard provides a web-dashboard for organizing and analyzing runs and results.

Index Terms—reproducible research, Python, machine learning, database, hyperparameter optimization

Introduction

A major part of machine learning research typically involves a significant number of computational experiments run with many different hyperparameter settings. This process holds many practical challenges, such as bookkeeping and maintaining reproducibility. To make matters worse, experiments often run on diverse and heterogeneous environments, ranging from laptops to cloud computing nodes. Due to deadline pressure and the inherently unpredictable nature of research, there is usually little incentive for researchers to build robust infrastructures. As a result, research code often evolves quickly and compromises essential aspects like bookkeeping and reproducibility.

Many tools exist for tackling different aspects of this process, including databases, version control systems, command-line interface generators, tools for automated hyperparameter optimization, spreadsheets, and so on. Few, however, integrate these aspects into a unified system, so each tool has to be learned and used separately, each incurring its overhead. Since there is no common basis to build a workflow, the tools people create will be tied to their particular setup. This impedes sharing and collaboration on tools for major problems like optimizing hyperparameters, summarizing and analyzing results, rerunning experiments, distributing runs, etc..

* *Corresponding author: klaus@idsia.ch*
‡ *Istituto Dalle Molle di Studi sull'Intelligenza Artificiale (IDSIA)*
§ *Università della Svizzera italiana (USI)*
¶ *Scuola universitaria professionale della Svizzera italiana (SUPSI)*
‖ *University of Freiburg*
** *Czech Technical University in Prague*

Sacred aims to fill this gap by providing central infrastructure for running computational experiments. We hope that it will help researchers and foster the development of a rich collaborative ecosystem of shared tools. In the following, we briefly introduce Sacred and two supporting tools: *Labwatch* integrates a convenient unified interface to several automated hyperparameter optimizers, such as random search, RoBO, and SMAC. *Sacredboard* offers a web-based interface to view runs and organize results.

Sacred

Sacred[1] is an open source Python framework that bundles solutions for some of the most frequent challenges of computational research. It does not enforce any particular workflow and is independent of the choice of machine learning libraries. Designed to remain useful even under deadline pressure, Sacred aims to offer maximum convenience while minimizing boilerplate code. By combining these features into a unified but flexible workflow, Sacred enables its users to focus on research, and still capture all the relevant information for each run. Its standardized configuration process allows smooth integration with other tools, such as Labwatch for hyperparameter optimization. Through storage of run information in a central database, comprehensive query and sorting functionality for bookkeeping becomes available. This further enables downstream analysis and allows other tools, such as Sacredboard, to provide a powerful user interface for organizing results.

Overview

The core abstraction of Sacred is the `Experiment` class that needs to be instantiated for each computational experiment. It serves as the central hub for defining configuration, functions, and for accessing the other features. To adopt Sacred, all that is required is to instantiate an `Experiment` and to decorate the main function to serves as entry-point. A minimal example could look like this:

```
from sacred import Experiment
ex = Experiment()

@ex.automain
def main():
    ...  # <= experiment code here
    return 42
```

This experiment is ready to be run and would return a *result* of 42. It already features an automatically generated command

1. https://github.com/IDSIA/Sacred

line interface, collects relevant information about dependencies and the host system, and can do bookkeeping. The experiment is extendable in several ways to define (hyper-) parameters that can later be externally changed.

The experiment can be run through its command-line interface, or directly from Python by calling `ex.run()`. Both modes offer the same ways for passing options, setting parameters, and adding observers. Once this experiment is started, Sacred will 1) interpret the options, 2) evaluate the parameter configuration, 3) gather information about dependencies and host, and 4) construct and call a `Run` object that is responsible for executing the main function. In the previous minimal example the output would look like this:

```
WARNING - my_example - No observers have been added
INFO - my_example - Running command 'main'
INFO - my_example - Started
INFO - my_example - Result: 42
INFO - my_example - Completed after 0:00:00
```

For each run, relevant information such as parameters, package dependencies, host information, source code, and results are automatically captured. The Run also captures the stdout, custom information, and fires events at regular intervals that can be monitored for bookkeeping, by optional *observers*. Several built-in observers are available for databases, disk storage, or sending out notifications.

Configuration

An important goal of Sacred is to make it convenient to define, update and use hyperparameters, which we will call the *configuration* of the experiment. The main way to set up the configuration is through functions decorated with `@ex.config`:

```
@ex.config
def cfg():
    nr_hidden_units = 512
    optimizer = 'sgd'
    learning_rate = 0.1
    log_dir = 'log/NN{}'.format(nr_hidden_units)
```

When running an experiment, Sacred executes these functions and adds their local variables to the configuration. This syntactically convenient way of defining parameters leverages the full expressiveness of Python, including complex expressions, function calls, and interdependent variables. Alternatively, plain dictionaries or external configuration files can also be used.

To make parameters readily available throughout the code, Sacred employs the technique of *dependency injection*: any function decorated by `@ex.capture` can directly accept any configuration entry as a parameter. Whenever such a function is called, Sacred will automatically pass those parameters by name from the configuration. This allows for the flexible and convenient use of the hyperparameters throughout the experiment code:

```
@ex.capture
def set_up_optimizer(loss, optimizer, learning_rate):
    OptClass = {
        'sgd': tf.train.GradientDescentOptimizer,
        'adam': tf.train.AdamOptimizer}[optimizer]
    opt = OptClass(learning_rate=learning_rate)
    return opt.minimize(loss)
```

When calling the `setup_optimizer` function, both the `optimizer` and the `learning_rate` arguments are optional. If omitted, they will be filled in automatically from the configuration. These injected values can be mixed freely with standard parameters, and injection follows the priority: 1) explicitly passed arguments 2) configuration values 3) default values.

The main benefit of config parameters is that they can be controlled externally when running an experiment. This can happen both from the command line

```
>> python my_experiment.py with optimizer='adam'
... learning_rate=0.001
```

or from Python calls:

```
from my_experiment import ex
ex.run(config_updates={'nr_hidden_units': 64})
```

Sacred treats these values as fixed while executing the config functions. In this way, they influence dependent values as you would expect. Thus in our example `log_dir` would be set to `"log/NN64"`.

Groups of config values that should be saved or set together can be collected in so-called *named configurations*. These are defined analogously to configurations using a function decorated by `@ex.named_config` (or dictionaries/config files):

```
@ex.named_config
def adam():
    optimizer = 'adam'
    learning_rate = 0.001
```

Named configs can be added both from the command line and from Python, after which they are treated as a set of updates:

```
>> python my_experiment.py with adam
```

Reproducibility

An important goal of Sacred is to collect all necessary information to make computational experiments reproducible while remaining lightweight enough to be used for each run, even during development. In this respect it differs from environment capturing tools such as ReproZip [CRSF16], CDE [Guo12], PTU [PMF13] and CARE [JVD14]. These tools *ensure* reproducibility by tracking and storing all data files and libraries used during a run at the system level. Sacred in contrast uses heuristics to capture the source code and for determining versions of used packages, collects limited but customizable information about the host system, and offers support for manually adding relevant data files. It explicitly excludes system libraries that are not python packages, data files that are not specifically added by the user, and hardware other than the CPU and GPU. This trade-off allows Sacred to run efficiently regarding computational overhead and required storage capacity at the expense of reproducibility on systems that differ too much from the original host. The focus is on the ability of the researcher to reproduce their results. For distributing the code, we advise the use of one of the above-mentioned environment capturing tools.

The source code of an experiment is arguably the most important piece of information for reproducing any result. Unfortunately, research code often has to be rapidly adapted under deadline pressure. A typical pattern in practice is, therefore, to quickly change something and start a run, without properly committing the changes into a VCS system. To deal with such an unstructured implementation workflow, Sacred doesn't rely on any VCS system (In contrast to Sumatra [Dav12]) and instead automatically detects and stores the source files alongside the run information[2]. Source files are gathered by inspecting all imported modules and keeping those defined within the (sub-)directories of the main script. This heuristic works well for flat use-cases that consist only of a few sources but fails to detect files that are imported deeper in the dependency tree. For cases of more complex source code structure Sacred also supports a stricter

Git-based workflow and can automatically collect the current commit and state of the repository for each run. The optional `--enforce-clean` flag forces the repository to be clean (not contain any uncommitted changes) before the experiment can be started. Relevant dependencies can also always be added manually by calling `ex.add_source_file(FILENAME)`.

Similarly, Sacred collects information about the package dependencies, by inspecting the imported modules. For all modules that are do not correspond to local source files or builtins, it uses several heuristics to determine the version number. First, it checks the `__version__` property and variations thereof. If that fails it uses the (much slower) Python package resource API to This detection will catch all dependencies imported from the main file of the experiment but will miss dependencies deeper down the dependency graph and any dynamic imports that happen only during runtime. Here again, further dependencies can be added manually using `ex.add_package_dependency(NAME, VERSION)`

Sacred also collects information about the host system including the hostname, type and version of the operating system, Python version, and the CPU. Optionally, it supports information about GPU, environment variables, and can be easily extended to collect custom information.

Randomness

Randomization is an important part of many machine learning algorithms, but it inherently conflicts with the goal of reproducibility. The solution, of course, is to use pseudo-random number generators (PRNG) that take a seed and generate seemingly random numbers in a deterministic fashion. However, setting the seed to a fixed value as part of the code makes all the runs deterministic, which can be an undesired effect. Sacred solves this problem by generating a new seed that is stored as part of the configuration for each run. It can be accessed from the code in the same way as every other config entry. Furthermore, Sacred automatically seeds the global PRNGs of the `random` and `numpy` modules when starting an experiment, thus making most sources of randomization reproducible without any intervention from the user.

Bookkeeping

Sacred accomplishes bookkeeping through the observer pattern [GHJV94]: The experiment publishes all the collected information in the form of events, to which observers can subscribe. Observers can be added dynamically from the command line or directly in code:

```
from sacred.observers import MongoObserver
ex.observers.append(MongoObserver.create("DBNAME"))
```

Events are fired when a run is started, every 10 seconds during a run (heartbeat), and once it stops (either successfully or by failing). The information is thus already available during runtime, and partial data is captured even in the case of failures. The most important events are:

Started Event
: Fired when running an experiment, just before the main method is executed. Contains configuration values, start time, package dependencies, host information, and some meta information.

Heartbeat Event
: Fired continuously every 10 seconds while the experiment is running. Contains the beat time, captured stdout/stderr, custom information, and preliminary result.

Completed Event
: Fired once the experiment completes successfully. Contains the stop time and the result.

Failed Event
: Fired if the experiment aborts due to an exception. Contains the stop time and the stack trace.

Sacred ships with observers that store all the information from these events in a MongoDB, SQL database, or locally on disk. Furthermore, there are two observers that can send notifications about runs via Telegram [DD17] or Slack [Sla17], respectively. Moreover, the observer interface is generic and supports easy addition of custom observers.

The recommended observer is the `MongoObserver`, which writes to a MongoDB [Mon17]. MongoDB is a noSQL database, or more precisely a *Document Database*: it allows the storage of arbitrary JSON documents without the need for a schema as in a SQL database. These database entries can be queried based on their content and structure. This flexibility makes it a good fit for Sacred because it permits arbitrary configuration of each experiment that can still be queried and filtered later on. This feature, in particular, has been very useful in performing large-scale studies such as the one in previous work [GSK+15]. A slightly shortened example database entry corresponding to our minimal example from above could look like this:


```
{"_id": 1,
 "captured_out": "[...]",
 "status": "COMPLETED",
 "start_time": "2017-05-30T20:34:38.855Z",
 "experiment": {
     "mainfile": "minimal.py",
     "sources": [["minimal.py", "ObjectId([...])"]],
     "repositories": [],
     "name": "minimal",
     "dependencies": ["numpy==1.11.0",
                      "sacred==0.7.0"],
     "base_dir": "/home/greff/examples"},
 "result": 42,
 "info": {},
 "meta": {"command": "main",
          "options": ["..."]},
 "format": "MongoObserver-0.7.0",
 "resources": [],
 "host": {"os": "Linux-3.16.0-4-amd64-x86_64",
          "cpu": "Intel(R) Core(TM) i5-4460  CPU",
          "hostname": "zephyr",
          "ENV": {},
          "python_version": "3.4.2"},
 "heartbeat": "2017-05-30T20:34:38.902Z",
 "config": {"seed": 620395134},
 "command": "main",
 "artifacts": [],
 "stop_time": "2017-05-30T20:34:38.901Z"
 }
```


Labwatch

Finding the correct hyperparameter for machine learning algorithms can sometimes make the difference between state-of-the-art performance and performance that is as bad as random guessing. It is often done by trial and error despite a growing number of tools that can automate the optimization of hyperparameters. Their adoption is hampered by the fact that each optimizer requires

2. It does, however, avoid duplicating files that remain unchanged to reduce storage requirements.

the user to adapt their code to a certain interface. Labwatch[3] simplifies this process by integrating an interface to a variety of hyperparameter optimizers into Sacred. This allows for easy access to hyperparameter optimization in daily research.

LabAssistant

At the heart of Labwatch is the so-called LabAssistant, which connects the Sacred experiment with a hyperparameter configuration search space (in short: *searchspace*) and a hyperparameter optimizer through a MongoDB database. For bookkeeping, it leverages the database storage of evaluated hyperparameter configurations, which allows parallel distributed optimization and also enables the use of post hoc tools for assessing hyperparameter importance (e.g. fANOVA [HHLBa]). When using Labwatch, the required boilerplate code becomes:

```
from sacred import Experiment
from labwatch.assistant import LabAssistant
from labwatch.optimizers import RandomSearch

ex = Experiment()
a = LabAssistant(experiment=ex,
                 database_name="labwatch",
                 optimizer=RandomSearch)
```

Search Spaces

In general, Labwatch distinguishes between *categorical* hyperparameters that can have only discrete choices and *numerical* hyperparameters that can have either integer or float values. For each hyperparameter, the search space defines a prior distribution (e.g. uniform or Gaussian) as well as its type, scale (e.g. log scale, linear scale) and default value.

Search spaces follow the same interface as Sacred's named configurations:

```
@ex.config
def cfg():
    batch_size = 128
    learning_rate = 0.001

@a.searchspace
def search_space():
    learning_rate = UniformFloat(lower=10e-3,
                                 upper=10e-2,
                                 default=10e-2,
                                 log_scale=True)
    batch_size = UniformNumber(lower=32,
                               upper=64,
                               default=32,
                               type=int,
                               log_scale=True)
```

This `search_space` can likewise be specified when executing the Experiment through the command line:

```
>> python my_experiment.py with search_space
```

Labwatch then triggers the optimizer to suggest a new configuration based on all configurations that are stored in the database and have been drawn from the same search space.

Multiple Search Spaces

Labwatch also supports multiple search spaces, which is convenient if one wants to switch between optimizing different sets of hyperparameters. Assume that we only want to optimize the learning rate and keep the batch size fixed, we can create a second smaller search space:

```
@a.searchspace
def small_search_space():
    learning_rate = UniformFloat(lower=10e-3,
                                 upper=10e-2,
                                 default=10e-2,
                                 log_scale=True)
```

This can be run in the same way as before by just swapping out the name of the searchspace:

```
>> python my_experiment.py with small_search_space
```

The optimizer will now only suggest a value for the learning rate and leaves all other hyperparameters, such as the batch size, untouched.

Hyperparameter Optimizers

Labwatch offers a simple and flexible interface to a variety of state-of-the-art hyperparameter optimization methods, including:

- **Random search** is probably the simplest hyperparameter optimization method [BB12]. It just samples hyperparameter configurations randomly from the corresponding prior distributions. It can be used in discrete as well as continuous search spaces and can easily be run in parallel.
- **Bayesian optimization** fits a probabilistic model to capture the current belief of the objective function [SSW+16], [SLA]. To select a new configuration, it uses a utility function that only depends on the probabilistic model to trade off exploration and exploitation. There are different ways to model the objective function:
  Probably the most common way is to use a **Gaussian process** to model the objective function, which tendn to work well in low (<10) dimensional continuous search spaces but do not natively work with categorical hyperparameters. Furthermore, due to their cubic complexity, they do not scale well with the number of function evaluations. We used RoBO[4] as an implementation, which is based on the George GP library [AFG+14].
  SMAC is also a Bayesian optimization method, but uses random forest instead of Gaussian processes to model the objective function [HHLBb]. Random forest natively allow to work in high dimensional mixed continuous and discrete input spaces but seem to work less efficient compared to Gaussian processes in low-dimensional continuous searchspaces [EFH+13].
  More recently, Bayesian neural networks have been used for Bayesian optimization [SRS+15], [SKFH16]. Compared to Gaussian processes, they scale very well in the number of function evaluation as well as in the number of dimensions. Here we use the **Bohamiann** approach [SKFH16], which is also implemented in the RoBO framework.

For each of these optimizers, Labwatch provides an adapter that integrates them into a common interface:

```
class Optimizer(object):


    def suggest_configuration(self):
        # Run the optimizer and
        # return a single configuration
        return config

    def update(self, configs, costs, runs):
        # Update the internal
```

3. https://github.com/automl/labwatch

4. https://github.com/automl/RoBO

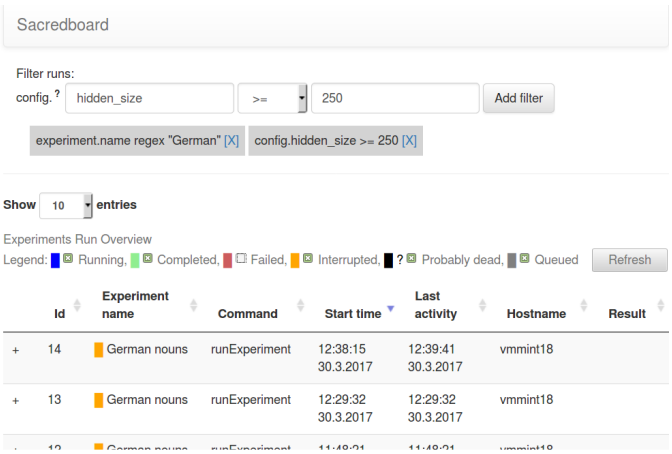


Fig. 1: *Sacredboard user interface*

```
        # state of the optimizer
        pass
```

This allows researchers to easily integrate their own hyperparameter optimization method into Labwatch. They only need to implement an adapter that provides the `suggest_configuration()` method which returns a single configuration to Sacred, and the `update()` method, which gets all evaluated configuration and costs, and updates the internal state of the optimizer.

Sacredboard

Sacredboard provides a convenient way for browsing runs of experiments stored in a Sacred MongoDB database. It consists of a lightweight `flask`-based web server that can be run on any machine with access to the database. The hosted web-interface shows a table view of both running and finished experiments, which are automatically updated. Sacredboard shows the current state and results, and offers a detail view that includes configuration, host information, and standard output of each run. At the moment, it relies exclusively on the MongoDB backend of Sacred, but in the future, we hope to support other options for bookkeeping as well.

Filtering

Experiments can be filtered by status to, for example, quickly remove failed experiments from the overview. Sacredboard also supports filtering by config values, in which case the user specifies a property name and a condition. By default, the name refers to a variable from the experiment configuration, but by prepending a dot (`.`), it can refer to arbitrary stored properties of the experiment. Possible conditions include numerical comparisons ($=, \neq, <, >, \geq, \leq$) as well as regular expressions. Querying elements of dictionaries or arrays can be done using the dot notation (e.g. `.info.my_dict.my_key`). A few useful properties to filter on include: the standard output (`.captured_out`), experiment name (`.experiment.name`), the info dictionary content (`.info.custom_key`), hostname (`.host.hostname`) and the value returned from the experiment's main function (`.result`). These filters can be freely combined.

The Details View

Clicking on any of the displayed runs expands the row to a details-view that shows the hyperparameters used, information about the machine, the environment where the experiment was run, and the standard output produced by the experiment. The view is organised as a collapsible table, allowing dictionaries and arrays to be easily browsed.

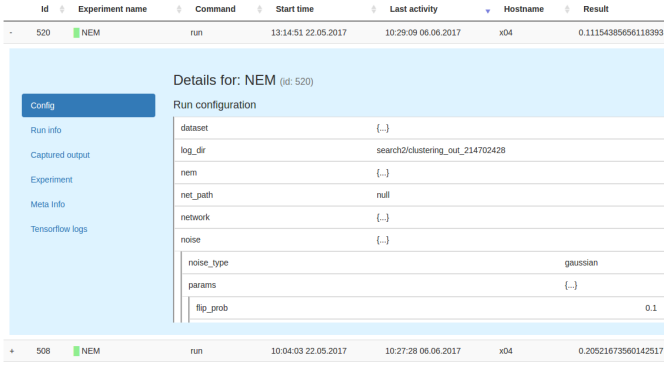


Fig. 2: *Sacredboard detail view*

Connecting to TensorBoard

Sacredboard offers an experimental integration with TensorBoard — the web-dashboard for the popular TensorFlow library [Goo]. Provided that the experiment was annotated with `@sacred.stflow.LogFileWriter(ex)` as in our example below and a TensorFlow log has been created during the run, it is possible to launch TensorBoard directly from the Run detail view.

Plotting Metrics

Sacredboard can visualize metrics such as accuracy or loss if they are tracked using Sacreds metrics interface. Metrics can be tracked through the Run object, which is accessible by adding the special `_run` variable to a captured function. This object provides a `log_scalar` method than can be called with an arbitrary metric name, its value, and (optionally) the corresponding iteration number:

```
_run.log_scalar("test.accuracy", 35.25, step=50)
```

The values for each metric are aggregated into a list of step index and values, where the last step number is autoincremented if the `step` parameter is omitted. Sacredboard will display metrics collected in this form as plots in the details view.

Example

In this section, we combine everything for the machine-learning-equivalent of a hello world program: MNIST classification. Here we use the current development version of Sacred and the Tensorflow and Keras libraries.

Header

First, we import the required packages and functions. Then an `Experiment` and a `LabAssistant` are instantiated:

```
import tensorflow as tf
from tensorflow import placeholder
from tensorflow.examples.tutorials.mnist import \
    input_data

from keras import backend as K
from keras.layers import Dense
from keras.objectives import categorical_crossentropy
from keras.metrics import categorical_accuracy

import sacred
```

```
import labwatch
from labwatch.optimizers import RandomSearch

ex = sacred.Experiment()
la = labwatch.LabAssistant(ex, optimizer=RandomSearch)
```

Configuration and Searchspace

Now we can define the configuration of the experiment. Note that we specify six parameters and that the `log_dir` depends on the `hidden_units`:

```
@ex.config
def cfg():
    hidden_units = 512
    batch_size = 32
    nr_epochs = 100
    optimizer = 'sgd'
    learning_rate = 0.1
    log_dir = 'log/NN{}'.format(hidden_units)
```

We also make use of a `named_config` to group together the adam optimizer with a reduced learning rate. In this way, we can start the experiment by specifying `with adam` and have both parameters changed.

```
@ex.named_config
def adam():
    optimizer = 'adam'
    learning_rate = 0.001
```

Finally, we define a searchspace over `learning_rate` and `hidden_units`, naturally treated in log-space. Now we can run our experiment using `with search_space` and have these two parameters set to suggestions by our hyperparameter optimizer (here `RandomSearch`).

```
@la.searchspace
def search_space():
    learning_rate = UniformFloat(0.001, 1.0,
                                 log_scale=True)
    hidden_units = UniformInt(32, 512,
                              log_scale=True)
```

Captured Functions

Sacreds config injection allows us to use the configuration parameters in any captured function. So here we use this feature to define two helper functions that set up our neural network model and our optimizer. Note that the `set_up_optimizer` function also takes the loss, which is not part of the configuration and has therefore to be passed normally:

```
@ex.capture
def build_model(hidden_units):
    img = placeholder(tf.float32, shape=(None, 784))
    label = placeholder(tf.float32, shape=(None, 10))

    h = Dense(hidden_units, activation='relu')(img)
    preds = Dense(10, activation='softmax')(h)

    loss = tf.reduce_mean(
        categorical_crossentropy(label, preds))
    accuracy = tf.reduce_mean(
        categorical_accuracy(label, preds))

    return img, label, loss, accuracy


@ex.capture
def set_up_optimizer(loss, optimizer, learning_rate):
    OptClass = {
        'sgd': tf.train.GradientDescentOptimizer,
        'adam': tf.train.AdamOptimizer}[optimizer]
    opt = OptClass(learning_rate=learning_rate)
    return opt.minimize(loss)
```

Main Method

Finally, the main method combines everything and serves as the entry point for execution. We've decorated it with `@sacred.stflow.LogFileWriter(ex)` to automatically capture the log directory used for the `FileWriter` in the appropriate format for Sacredboard. The main method is also automatically a captured function, and takes three of the configuration values as parameters. It also accepts a special parameters `_run` which grants access to the current `Run` object. Note that we call the other captured functions without passing any of the configuration values, since they will be filled in automatically.

```
@ex.automain
@sacred.stflow.LogFileWriter(ex)
def main(batch_size, nr_epochs, log_dir, _run):
    # initialize tensorflow and load data
    sess = tf.Session()
    K.set_session(sess)
    mnist = input_data.read_data_sets('MNIST_data',
                                      one_hot=True)

    # call captured functions for model and optimizer
    img, label, loss, acc = build_model()
    train_step = set_up_optimizer(loss)

    # set up FileWriter for later use of Tensorboard
    summary_writer = tf.summary.FileWriter(log_dir)
    summary_writer.add_graph(tf.get_default_graph())

    # initialize variables and main loop
    sess.run(tf.global_variables_initializer())
    for epoch in range(nr_epochs):
        batch = mnist.train.next_batch(batch_size)
        _, l, a = sess.run([train_step, loss, acc],
                           feed_dict={label: batch[1],
                                      img: batch[0]})

        # add loss and accuracy as metrics
        _run.log_scalar("train.cross_entropy", l)
        _run.log_scalar("train.accuracy", a, epoch)

    # return test accuracy as final result
    return sess.run(acc, feed_dict={
                        img: mnist.test.images,
                        label: mnist.test.labels})
```

Related Work

We are aware of only a few projects that have a focus similarly broad as Sacred, the closest one being Sumatra [Dav12]. Both projects are very similar in that they collect and store information about sources, dependencies, configurations, and host information. Their main difference is that Sumatra comes as a command line tool for running experiments "from the outside", while Sacred was designed as a Python package to be used from within the experiment script. So while Sacred is limited to Python scripts, Sumatra can track any executable as long as its command line interface matches a certain structure. This, on the other hand, allows sacred to provide many conveniences like the flexible configuration system with configuration injection, automatic seeding of random number generators, support for live updated custom information, and integration with 3rd party libraries like Tensorflow. It also means that Sacred scripts are self-sufficient, while Sumatra relies on a separate outside project-configuration stored in a hidden *.smt* directory. Another subtle but important difference is that Sumatra relies mainly on SQL for storing run information, while Sacred favors MongoDB. The use of this schema-free database enables querying Sacred runs based on dynamic structure such as configuration entries (even nested ones) and custom information.

Workflow management systems such as Taverna [WHF+13], Kepler [ABJ+04], Vistrails [BCC+05], and Galaxy [GRH+05] can also capture provenance information and ensure reproducibility. They let the user define a workflow based on standardized components, often through a graphical user interface without any direct programming. Implementing a custom component is usually difficult, which restricts their usefulness to the supported ones and thus to their intended domain. The line between workflow management and free-form programming blurs with tools like Reskit [IP17] and FBLearner Flow [Dun16]. Sacred, however, is targeted at general machine learning audience, and therefore works with arbitrary python instead of some set of standardized components.

Experiment databases [VBPH12], [SWGCM14], [emp17] represent a related bodey of work by making an effort to unify the process and storage of machine learning problems and experiments by expressing them in a common language. By standardizing that language, they improve comparability and communicability of the results. The most well-known example might be the OpenML project [VvRBT14]. This standardization has benefits, but also imposes certain restrictions on the conducted experiments. Therefore, to keep Sacred as general as possible, we chose not to build it ontop of an experiment database. That being said, we believe there is a lot of value in adding (optional) interfaces to experiment databases to Sacred.

There are several tools such as noWorkflow [PBMF15], ProvenanceCurious [HAW13], and others [BGS08] to extract fine-grained provenance information from python scripts. Whereas Sacred treats the executed code mostly as a black box, these tools use inspection and tracing techniques to extract function call graphs and data flow. This information is then often stored in the form of the Open Provenance Model in a relational database and enables in-depth analysis of the performed computations.

Some other projects, including FGLab [Aru17], the proprietary Aetros [Aet17], and Neptune [dee17], focus on providing a dashboard. Jobman [Job12] is a Python library for scheduling lots of machine learning experiments which also helps in organizing hyperparameter searches and bookkeeping. Several projects exist with a focus on capturing the entire environment of an experiment to ensure its reproducibility. They include tools such as ReproZip [CRSF16], CDE [Guo12], PTU [PMF13], CARE [JVD14]. They trace dependencies on an operating system level and help in packaging everything that is needed to rerun an experiment exactly.

Conclusion

Sacred is an open source Python framework which aims to provide infrastructure for computational experiments with minimal boilerplate code and maximum convenience. This paper presented its key features and demonstrated how they interact to tackle some of the basic problems of computational experimentation, like managing parameters, bookkeeping, and reproducibility. We hope that through convenience and modularity, Sacred will help to build a rich ecosystem of tools. Two such supporting tools are Labwatch and Sacredboard. Labwatch interfaces the powerful configuration system of sacred with several hyperparameter optimization libraries, thus significantly simplifying the tuning of configurations. Sacredboard, on the other hand, provides a web-based interface to view recorded runs, facilitating a live overview of all the experiments.

Future Work

Sacred has been useful for many researchers already, but there are still many possible improvements on our roadmap. This includes support for more complex experimental setups, like having separate training and evaluation scripts as is common with large Tensorflow models. Similarly, it would be interesting to offer support and a clear workflow for the continuation of aborted runs.

While Sacred helps to capture relevant information about experiments, it does not offer much support for organizing and analyzing results. To tackle this we plan to provide a unified interface for querying the records created by different observers. This semi-standardized format will enable the creation of general analysis tools, and extend the applicability of existing tools like Sacredboard.

Another important direction is to automate the process of actually reproducing Sacred experiments. As of now the researcher has to manually reconstruct the environment, copy the stored source files and run with the saved configuration parameters. An integration with environment capturing tools ReproZip could allow for creating packages that can be rerun on any system in a completely automated fashion.

Finally, we plan on improving the support of Sacred for scheduling and distributing runs. It already supports "queueing up" experiments, which only creates a database entry containing the sources, desired configuration, and the status `QUEUED`. In the future, we hope to include workers that can be run on different machines and which will fetch queued runs from the database and execute them. This way, Sacred could offer basic support for distributing computations.

Acknowledgements

This work has partly been supported by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme under grant no. 716721, by the Euro- pean Commission under grant no. H2020-ICT-645403-ROBDREAM, and by the German Research Foundation (DFG) under Priority Programme Autonomous Learning (SPP 1527, grant HU 1900/3-1). This research was supported by the EU project `INPUT` (H2020-ICT-2015 grant no. 687795). Access to computing and storage facilities owned by parties and projects contributing to the Czech National Grid Infrastructure MetaCentrum provided under the programme "Projects of Large Research, Development, and Innovations Infrastructures" (CESNET LM2015042) is greatly appreciated.

References

[ABJ+04] Ilkay Altintas, Chad Berkley, Efrat Jaeger, Matthew Jones, Bertram Ludascher, and Steve Mock. Kepler: an extensible system for design and execution of scientific workflows. In *Scientific and Statistical Database Management, 2004. Proceedings. 16th International Conference on*, pages 423–424. IEEE, 2004.

[Aet17] Aetros. Home // aetros, 2017. [Online; accessed 30-May-2017]. URL: http://aetros.com/.

[AFG+14] S. Ambikasaran, D. Foreman-Mackey, L. Greengard, D. W. Hogg, and M. O'Neil. Fast Direct Methods for Gaussian Processes and the Analysis of NASA Kepler Mission Data. March 2014.

[Aru17] Kai Arulkumaran. Fglab: Machine learning dashboard, 2017. [Online; accessed 30-May-2017]. URL: https://kaixhin.github.io/FGLab/.

[BB12] J. Bergstra and Y. Bengio. Random search for hyper-parameter optimization. 13:281–305, 2012.

[BCC+05] Louis Bavoil, Steven P Callahan, Patricia J Crossno, Juliana Freire, Carlos E Scheidegger, Cláudio T Silva, and Huy T Vo. Vistrails: Enabling interactive multiple-view visualizations. In *Visualization, 2005. VIS 05. IEEE*, pages 135–142. IEEE, 2005.

[BGS08] Carsten Bochner, Roland Gude, and Andreas Schreiber. Provenance and annotation of data and processes. chapter A Python Library for Provenance Recording and Querying, pages 229–240. Springer-Verlag, Berlin, Heidelberg, 2008. URL: https://doi.org/10.1007/978-3-540-89965-5_24, doi:10.1007/978-3-540-89965-5_24.

[CRSF16] Fernando Chirigati, Rémi Rampin, Dennis Shasha, and Juliana Freire. Reprozip: Computational reproducibility with ease. In *Proceedings of the 2016 International Conference on Management of Data*, pages 2085–2088. ACM, 2016.

[Dav12] Andrew Davison. Automated capture of experiment context for easier reproducibility in computational research. *Computing in Science & Engineering*, 14(4):48–56, 2012.

[DD17] Pavel Durov and Nikolai Durov. Telegram messenger, 2017. [Online; accessed 06-June-2017]. URL: https://telegram.org/.

[dee17] deepsense.io. Neptune | deepsense.io, 2017. [Online; accessed 30-May-2017]. URL: https://deepsense.io/neptune/.

[Dun16] Jeffrey Dunn. Introducing fblearner flow: Facebook's ai backbone, 2016. [Online; accessed 29-June-2017]. URL: https://code.facebook.com/posts/1072626246134461/introducing-fblearner-flow-facebook-s-ai-backbone/.

[EFH+13] K. Eggensperger, M. Feurer, F. Hutter, J. Bergstra, J. Snoek, H. Hoos, and K. Leyton-Brown. Towards an empirical foundation for assessing Bayesian optimization of hyperparameters. In *NIPS Workshop on Bayesian Optimization in Theory and Practice (BayesOpt'13)*, 2013.

[emp17] empiricalci. Empirical, 2017. [Online; accessed 30-May-2017]. URL: https://empiricalci.com/.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Pearson Education, 1994.

[Goo] Google Inc. Tensorflow. [online]. [cit. 2017-05-29]. URL: www.tensorflow.org.

[GRH+05] Belinda Giardine, Cathy Riemer, Ross C Hardison, Richard Burhans, Laura Elnitski, Prachi Shah, Yi Zhang, Daniel Blankenberg, Istvan Albert, James Taylor, et al. Galaxy: a platform for interactive large-scale genome analysis. *Genome research*, 15(10):1451–1455, 2005.

[GSK+15] Klaus Greff, Rupesh Kumar Srivastava, Jan Koutník, Bas R Steunebrink, and Jürgen Schmidhuber. Lstm: A search space odyssey. *arXiv preprint arXiv:1503.04069*, 2015.

[Guo12] Philip Guo. Cde: A tool for creating portable experimental software packages. *Computing in Science & Engineering*, 14(4):32–35, 2012.

[HAW13] Mohammad Rezwanul Huq, Peter M. G. Apers, and Andreas Wombacher. Provenancecurious: A tool to infer data provenance from scripts. In *Proceedings of the 16th International Conference on Extending Database Technology*, EDBT '13, pages 765–768, New York, NY, USA, 2013. ACM. URL: http://doi.acm.org/10.1145/2452376.2452475, doi:10.1145/2452376.2452475.

[HHLBa] F. Hutter, H. Hoos, and K. Leyton-Brown. An efficient approach for assessing hyperparameter importance. pages 754–762.

[HHLBb] F. Hutter, H. Hoos, and K. Leyton-Brown. Sequential model-based optimization for general algorithm configuration. pages 507–523.

[IP17] Alexander Ivanov and Dmitry Petrov. Reskit: Researcher kit for reproducible machine learning experiments, 2017. [Online; accessed 29-June-2017]. URL: https://github.com/neuro-ml/reskit.

[Job12] Jobman. Welcome – jobman 0.1 documentation, 2012. [Online; accessed 30-May-2017]. URL: http://deeplearning.net/software/jobman/.

[JVD14] Yves Janin, Cédric Vincent, and Rémi Duraffort. Care, the comprehensive archiver for reproducible execution. In *Proceedings of the 1st ACM SIGPLAN Workshop on Reproducible Research Methodologies and New Publication Models in Computer Engineering*, page 1. ACM, 2014.

[Mon17] MongoDB, Inc. Mongodb. [online], 2017. [cit. 2017-05-28]. URL: https://www.mongodb.com/.

[PBMF15] João Felipe Nicolaci Pimentel, Vanessa Braganholo, Leonardo Murta, and Juliana Freire. Collecting and analyzing provenance on interactive notebooks: when ipython meets noworkflow. In *Workshop on the Theory and Practice of Provenance (TaPP), Edinburgh, Scotland*, pages 155–167, 2015.

[PMF13] Quan Pham, Tanu Malik, and Ian T Foster. Using provenance for repeatability. *TaPP*, 13:2, 2013.

[SKFH16] J. T. Springenberg, A. Klein, S. Falkner, and F. Hutter. Bayesian optimization with robust bayesian neural networks. In *Advances in Neural Information Processing Systems 29*, December 2016.

[SLA] J. Snoek, H. Larochelle, and R. P. Adams. Practical Bayesian optimization of machine learning algorithms. pages 2960–2968.

[Sla17] Slack. Slack: Where work happens, 2017. [Online; accessed 06-June-2017]. URL: https://slack.com/.

[SRS+15] J. Snoek, O. Rippel, K. Swersky, R. Kiros, N. Satish, N. Sundaram, M. Patwary, Prabhat, and R. Adams. Scalable Bayesian optimization using deep neural networks. In *Proceedings of the 32nd International Conference on Machine Learning (ICML'15)*, pages 2171–2180, 2015.

[SSW+16] B. Shahriari, K. Swersky, Z. Wang, R. Adams, and N. de Freitas. Taking the human out of the loop: A review of Bayesian optimization. *Proceedings of the IEEE*, 104(1):148–175, 2016.

[SWGCM14] Michael R. Smith, Andrew White, Christophe Giraud-Carrier, and Tony Martinez. An easy to use repository for comparing and improving machine learning algorithm usage. *arXiv:1405.7292 [cs, stat]*, May 2014. URL: http://arxiv.org/abs/1405.7292.

[VBPH12] Joaquin Vanschoren, Hendrik Blockeel, Bernhard Pfahringer, and Geoffrey Holmes. Experiment databases. *Machine Learning*, 87(2):127–158, January 2012. URL: http://link.springer.com/article/10.1007/s10994-011-5277-0, doi:10.1007/s10994-011-5277-0.

[VvRBT14] Joaquin Vanschoren, Jan N. van Rijn, Bernd Bischl, and Luis Torgo. OpenML: networked science in machine learning. *ACM SIGKDD Explorations Newsletter*, 15(2):49–60, 2014. URL: http://dl.acm.org/citation.cfm?id=2641198.

[WHF+13] Katherine Wolstencroft, Robert Haines, Donal Fellows, Alan Williams, David Withers, Stuart Owen, Stian Soiland-Reyes, Ian Dunlop, Aleksandra Nenadic, Paul Fisher, et al. The taverna workflow suite: designing and executing workflows of web services on the desktop, web or in the cloud. *Nucleic acids research*, 41(W1):W557–W561, 2013.

FigureFirst: A Layout-first Approach for Scientific Figures

Theodore Lindsay[‡], Peter T. Weir[¶], Floris van Breugel[§*]

Abstract—One major reason that Python has been widely adopted as a scientific computing platform is the availability of powerful visualization libraries. Although these tools facilitate discovery and data exploration, they are difficult to use when constructing the sometimes-intricate figures required to advance the narrative of a scientific manuscript. For this reason, figure creation often follows an inefficient serial process, where simple representations of raw data are constructed in analysis software and then imported into desktop publishing software to construct the final figure. Though the graphical user interface of publishing software is uniquely tailored to the production of publication quality layouts, once the data are imported, all edits must be re-applied if the analysis code or underlying dataset changes. Here we introduce a new Python package, FigureFirst, that allows users to design figures and analyze data in a parallel fashion, making it easy to generate and continuously update aesthetically pleasing and informative figures directly from raw data. To accomplish this, FigureFirst acts as a bridge between the Scalable Vector Graphics (SVG) format and Matplotlib [Hunter08] plotting in Python. With FigureFirst, the user specifies the layout of a figure by drawing a set of rectangles on the page using a standard SVG editor such as Inkscape [Altert13]. In Python, FigureFirst uses this layout file to generate Matplotlib figures and axes in which the user can plot the data. Additionally, FigureFirst saves the populated figures back into the original SVG layout file. This functionality allows the user to adjust the layout in Inkscape, then run the script again, updating the data layers to match the new layout. Building on this architecture, we have implemented a number of features that make complex tasks remarkably easy including axis templates; changing attributes of standard SVG items such as their size, shape, color, and text; and an API for adding JessyInk [Jagannathan12] extensions to Matplotlib objects for automatically generating animated slide presentations. We have used FigureFirst to generate figures for publications [Lindsay17] and provide code and the layouts for the figures presented in this manuscript at our GitHub page: http://flyranch.github.io/figurefirst/.

Index Terms—plotting, figures, SVG, Matplotlib

Introduction

Visualization has long been a critical element in the iterative process of science. Skill with the pen allowed the early pioneers of the scientific revolution to share, explain, and convince: Galileo was trained in the Florentine Accademie delle Arti del Disegno; and the intricate drawings of Da Vinci and Vesalius served to overturn Galen's entrenched theories—with Vesalius's historic textbook paving the way for William Harvey's discovery of a unified circulatory system [Aird11].

Although new web-enabled media formats are emerging to provide alternative mechanisms for scientific communication, the static printed publication remains the centerpiece of scientific discourse. A well-designed sequence of data-rich figures makes it easy for other researchers across disciplines to follow the narrative, assess the quality of the data, criticize the work, and remember the conclusions. In fact, the importance of the narrative in organizing and structuring the logic of research has led some to propose that writing the manuscript should be a more integral part of the original design and execution of experiments [Whitesides04]. According to this view, the researcher should create a text outline, as well as a visual story-board, long before all the data have been collected and analyzed. As new results come to light, the story-board is updated with new data and new experiments.

From a practical standpoint, taking this iterative approach with data-rich figures is challenging because desktop publishing and illustration software is not integrated with scientific analysis software, and using the Matplotlib API to directly specify plotting details is time consuming (Fig. 1). A few of the commercial software packages such as MATLAB(TM) and SigmaPlot(TM) provide some graphical tools to assist in figure layout, but these are severely limited compared to those available in vector graphics software such as Inkscape or Adobe Illustrator(TM), especially when creating multi-panel figures. For this reason, figure generation usually follows a unidirectional workflow in which authors first write code to analyze and plot the raw data, and only later do they import the figures into desktop publishing software for final editing and styling for press.

We created the open-source FigureFirst library to enable interoperability between open-source plotting and analysis tools available in Python (e.g. Matplotlib) and the graphical user interface provided by Scalable Vector Graphics (SVG) editors such as the open-source application Inkscape. By drawing a series of boxes in a blank SVG document, a researcher may rapidly generate a prototype of a multi-panel figure, and then populate this figure using powerful analysis and plotting functions in Python. The FigureFirst library allows the user to insert these plots back into the prototype SVG document, completing the loop between visualization and analysis. As data are collected, individual sub-panels in the figure may be populated, moved, resized or removed as the events of the ongoing study warrant. In this manner, the library facilitates a more iterative approach to this key aspect of the scientific method. Finally, by embedding information about the scripts used to generate the final figures within the SVG document

‡ *Caltech Division of Biology and Biological Engineering*
¶ *Data Science at Yelp*
∗ *Corresponding author: florisvb@gmail.com*
§ *University of Washington*

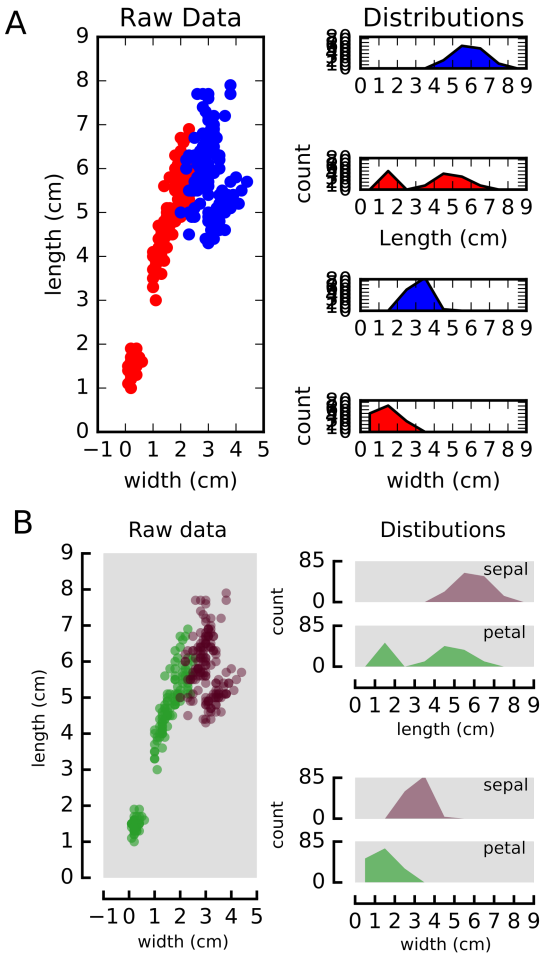


Fig. 1: *Figurefirst allows the plotting axes in a multi-panel figure to be quickly placed in a flexible and meaningful way. (A) A plot of the iris dataset created using the Matplotlib and gridspec API. (B) The same data plotted using FigureFirst. Note that the less rigid placement of axes helps highlight the inherent structure of the data. The full 27-line script used to create this panel can be found in the Summary and Future Directions section.*

itself, FigureFirst makes it possible to store an automatically updated and complete log of the transformation from raw data to a publication quality figure, encapsulating the analysis routine within the final figure. Thus, every step of the process may be kept under version control and published along with the manuscript, greatly increasing the transparency and reproducibility of the final publication.

Below we provide a short overview of the interface to the library in the Basic Usage section. We discuss more details on how to generate a layout file using Inkscape and xml tags in the Groups and Templates section. The Architecture section contains a more detailed description of the library for those interested in contributing to the project.

Basic Usage

With FigureFirst creating a new figure generally involves four steps:

1) **Design the layout file.** (Fig. 2A) Fundamentally this step entails decorating a specific subset of the objects in the SVG file with xml tags that identify what objects FigureFirst should expose to Python. For instance, the user specifies a Matplotlib axis by tagging an SVG `<rect/>` with the `<figurefirst:axis>` tag. If using Inkscape, we facilitate this step with a number of optional Inkscape extensions (Fig. 3).
2) **Import the layout into Python.** (Fig. 2B) Construct a `FigureLayout` object with the path to the layout file and then call the `make_mplfigures()` method of this object to generate `Matplotlib` figures and axes as specified in the layout.
3) **Plot data.** (Fig. 2C) All the newly created figure axes are available within the `axes` dictionary of the `FigureLayout` object.
4) **Save to SVG.** SVG graphics are merged with Matplotlib figures, allowing complex vector art to be quickly incorporated as overlays or underlays to your data presentation.

As an example, to generate Figure 2 we used Inkscape to construct a .SVG document called 'workflow_layout.SVG' containing a layer with three gray rectangles. We then used the tag axis Inkscape extension (Figure 3) to identify each `<rect/>` with a `<figurefirst:axes>` tag that has a unique name as an attribute. For instance, we tagged the gray rectangle that became panel C with `<figurefirst:axis figurefirst:name="plot_data" />`. In this example we have drawn in the axes spines and included this with the arrows and other annotations on a separate layer in the .SVG file to illustrate one way to use vector art overlays in a layout document.

In Python we may then use the FigureFirst module to plot some data to this axis using the following code:

```
1 import figurefirst as fifi
2 layout = fifi.FigureLayout('workflow_layout.SVG')
3 layout.make_mplfigures()
4 fifi.mpl_functions.kill_all_spines(layout)
5 x = np.linspace(0,2*pi); y = np.sin(x)
6 layout.axes['plot_data'].plot(x,y)
7 layout.save('workflow.SVG')
```

Lines 2 and 3 are responsible for parsing the layout document and generating the Matplotlib figures. In line 4 we pass the layout to a helper function in the mpl_functions submodule that removes the axes spines from all the axes contained within the layout. Lines 5-6 plot the data and line 7 saves the layout to a new SVG document called 'workflow.SVG' with all the Matplotlib axes associated with this figure inserted into a new layer. Because usually one will want to use Matplotlib to generate the axis spines we have included an auxiliary submodule called `mpl_functions` that contains a number of utility functions that operate on figures generated from layouts to apply consistent spine-styles and formats accross the axes of a figure. The rest of the figure panels were also generated in Python by simply calling `layout['panel_name'].imshow(screenshot_image)`. Note that there is nothing keeping us from using this new document as a layout document itself, enabling the placement of vector graphics objects in relation to plotted features.

Groups and Templates

Because the `figurefirst:name` attribute of the tagged `<rect>` will be used as the key in the layout.axes dictionary in Python, each panel in this example must be given a unique name.

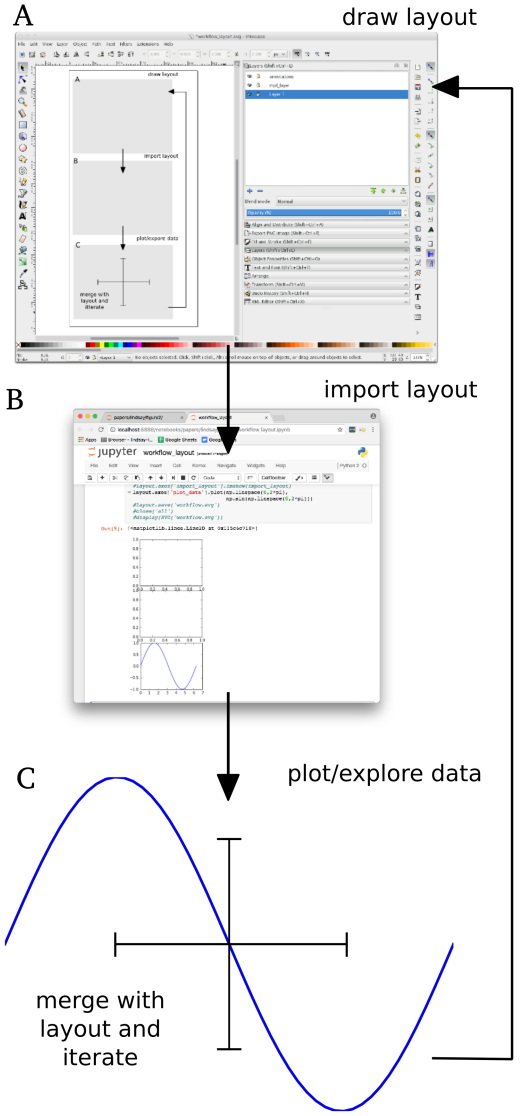


Fig. 2: *Overview of the iterative layout-based approach to figure creation using FigureFirst. (A) The user designs a figure layout in SVG, specifying the location and aspect-ratio of plotting axes. Additional vector art such as arrows or stylized axes spines can be included in the layout document. (B) FigureFirst interprets the layout document and generates Matplotlib axes and figures that the user can use to plot in Python. (C) When saving, the generated plots are merged with the original layout to incorporate the non-Matplotlib graphics. Note that this approach allows changes to the figure layout or analysis code to be applied at any point in the workflow.*

Generating these names can be a cumbersome requirement because scientific data often have a nested or hierarchical structure. Moreover, we found that when generating the code to plot a figure, it is useful if the organization of the layout document reflects the underlying data. Thus, we have provided two mechanisms to allow a hierarchical structure in the labels associated with a layout: groups and templates. Though the interfaces for working with these objects differ, they both generate a nested structure in the `layout.axes` dictionary.

When using groups, the hierarchy is specified in SVG by enclosing a set of tagged axes within the `<g>` container that itself is tagged with `<figurefirst:group>` using a

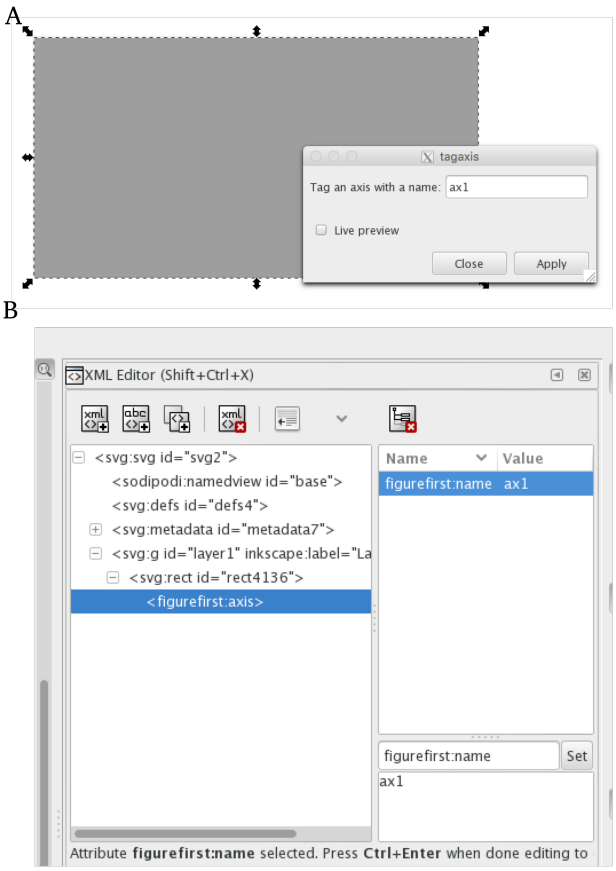


Fig. 3: *Screenshots of Inkscape illustrating the two mechanisms for applying the correct xml tags, which are used by FigureFirst to generate Matplotlib axes. (A) A dialog box allows the user to tag a rectangle as a FigureFirst axis. (B) The user can edit the document's XML directly using Inkscape's XML editor.*

`figurefirst:name` attribute. The axes are then exposed to the user in Python within the `layout.axes` dictionary keyed by tuples that contain the path in the hierarchy e.g. `myaxes = layout.axes[(groupname, axisname)]`.

Though groups allow for arbitrary nesting structure within the layout, it is common in scientific figures for a single display motif to be replicated multiple times in a multi-panel figure. For instance, one might want to plot data from a series of similar experiments performed under different conditions. In this case, the template feature allows for rapid design and modification of the layout without the need to tag each individual axis.

To illustrate the template feature, consider the task of making a more complex figure that describes three behavioral metrics for three different animal groups. With FigureFirst, the user can draw the layout for one of the groups, and then use this layout as a template for the other two (Fig. 4A-B). Later one can add, remove, or change the relative sizes of the axes in all three figures simply by editing the single template. In this example, each of the three groups was created using a new Matplotlib figure, which was then saved to a separate layer in the SVG file (Fig. 4C). Below is an excerpt of the code used to load the layout from Figure 3A, iterating through three groups and plotting the relevant data into a separate layer for each group (Fig. 4B-C). The complete code is available on our github page as a Jupyter notebook:

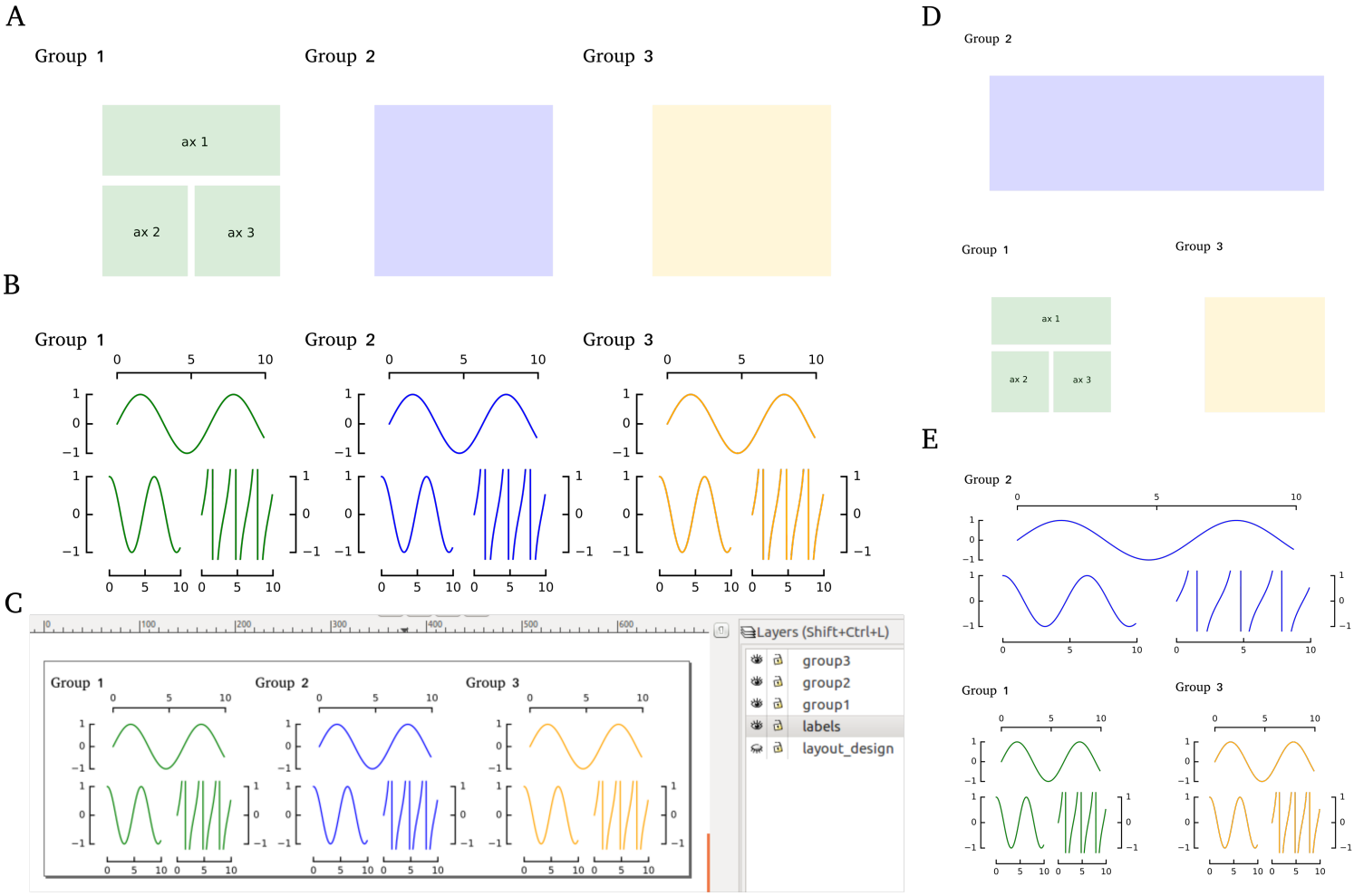


Fig. 4: *Creating and rearranging multi-panel figures using FigureFirst's template feature. (A) Layout for a figure. (B) Output. (C) Inkscape screenshot illustrating the layered structure. (D) Rearranged layout. (E) Output for the new layout (code remains identical). The code used to generate these figures is available as a Jupyter Notebook on our github page: https://github.com/FlyRanch/FigureFirst/blob/master/examples/figure_groups_and_templates/figure_templates_example.ipynb*

https://github.com/FlyRanch/FigureFirst/blob/master/examples/figure_groups_and_templates/figure_templates_example.ipynb.

```
import figurefirst as fifi
layout = fifi.FigureLayout(template_filename)
layout.make_mplfigures()

for group in ['group1', 'group2', 'group3']:
  for ax in ['ax1', 'ax2', 'ax3']:
     mpl_axis = layout.axes[(group, ax)]
     mpl_axis.plot(x_data, y_data,
                   color=colors[group])

  layout.append_figure_to_layer(
     layout.figures[group], group)

layout.write_svg(output_filename)
```

Additional SVG/Python interoperability

The decorator language we use for the FigureFirst xml tags is general, and we extended it to provide a simple mechanism for passing additional information back and forth between Python and SVG. This enables a few additional features we refer to as axis methods, path specs, xml passing, Python tracebacks and SVG items.

The axis methods feature allows the user to include Python code in the layout document to be applied to all the corresponding Matplotlib axes *en mass* when the `layout.apply_mpl_methods()` function is called in Python. Axis methods are enabled by adding an appropriate attribute to the `<figurefirst:axis>` tag. The value of this attribute will be parsed and passed as arguments to the method. For instance to specify the y limits of an axis to (0, 250) add the `figurefirst:set_ylim="0,250"` attribute to the corresponding `<figurefirst:axis>` tag.

In keeping with the notion that vector editing software is better suited for designing the visual landscape of a figure than code, we created the `<figurefirst:pathspec>` or `<figurefirst:patchspec>` tag to create a way for users to generate a palette of line and patch styles within the layout document and pass these to plotting functions in Python. Using this feature, a user can explore different stroke widths, colors and transparencies in Inkscape and then quickly pass these styles as keyword arguments to Matplotlib plotting functions.

The two tools described above allow the user to pass information from SVG to Python; we have also implemented features that allow data to be passed from Python back into SVG. For instance

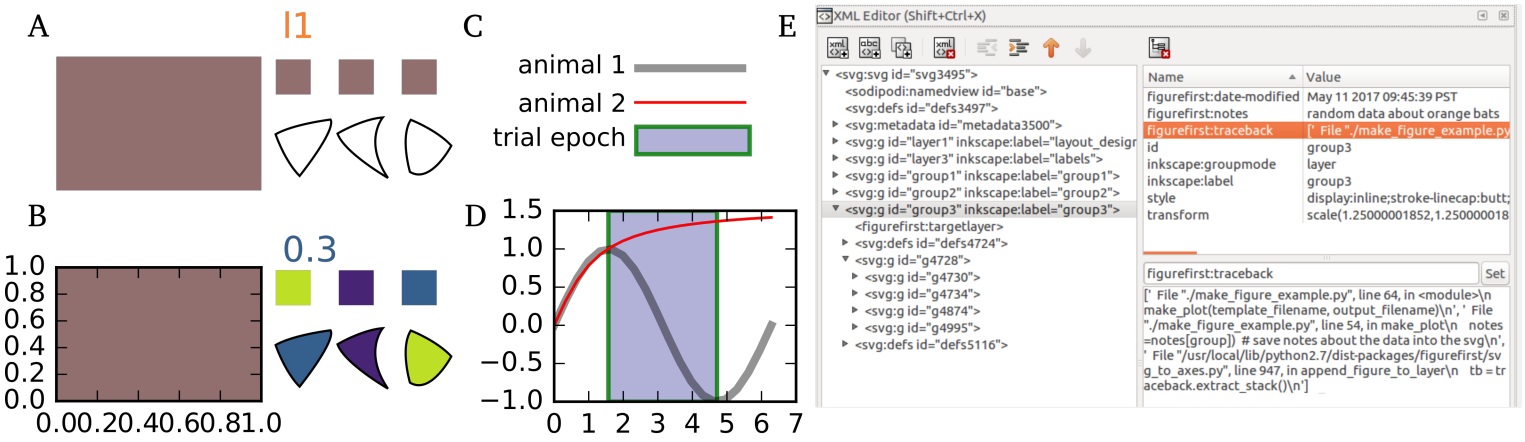


Fig. 5: *Additional features that use FigureFirst as an interface layer between SVG and Python. (A-B) SVGitems allows the attributes of SVG objects in the layout document to be to be edited and modified in Python. In the layout (A) the text item I1, the three small* `<rects/>` *as well as the three* `<path/>` *objects are tagged with* `<figurefirst:SVGitem figurefirst:name=somename>` *allowing the text and color of the objects to be changed in the final output shown in B. (C-D) Using* `<figurefirst:pathspec>` *and* `<figurefirst:patchspec>` *a palette of line or patch styles respectively, can be defined in SVG (C) and then passed as keyword arguments to Matplotlib plotting functions to generate the plot in D. (E) FigureFirst simplifies keeping track of when, how, and why your figures are created by embedding the time modified, user notes, and full traceback directly into each FigureFirst generated layer.*

the `pass_xml()` method of the layout class can be used to identify axes as slides in a JessyInk (https://launchpad.net/jessyink) presentation, or attach mouseover events or even custom javascript routines to a plotted path.

FigureFirst can also expose many types of SVG objects including text, patches, and circles to Python by tagging the object with the `<figurefirst:SVGitem>` tag (Fig. 5C-D). This makes it possible to use the Inkscape user interface to place labels, arrows, etc. while using Python to edit their attributes based on the data.

When quickly prototyping analysis and figures, it is easy to lose track of when you have updated a figure, and what code you used to generate it. FigureFirst allows the user to embed traceback information, time modified, and custom notes into the SVG file directly using the following option. See Figure 4E for a screenshot of the Inkscape output.

```
layout.append_figure_to_layer(layout.figures[group],
                              group,
                              save_traceback=True,
                              notes=notes[group])
```

In the future, we plan to expand the traceback capability by optionally linking the traceback to a github page so that when a FigureFirst generated SVG file is shared, other viewers can quickly find the code and data used to generate the figure. This option would directly and automatically link the scientific publication with the data and software, thereby facilitating open science with minimal user overhead. Alternatively, for simple and standalone Python scripts, it would be possible to embed the scripts directly into the xml.

Architecture

FigureFirst uses a minimal Document Object Model interface (xml.dom.minidom) to parse and write to an SVG file. We define a set of xml tags that the user may use to decorate a subset of SVG objects. Our library then exposes these objects to Python, where they are used, for example, to generate Matplotlib axes. We use the `<figurefirst:>` namespace in our xml to ensure that these tags will not collide with any other tags in the document.

When constructing a `figurefirst.FigureLayout`, FigureFirst parses the SVG document and transforms tagged SVG elements into a Python object that holds the key graphical data specified by SVG. For instance, as mentioned above, a box tagged with `<figurefirst:axis>` will be used to create a `FigureFirst.Axis` object that contains the x,y position of the origin, as well as the height and width of the tagged box. In the case that the tagged SVG objects are subject to geometric transforms from enclosing containers, FigureFirst will compose the transforms and apply them to the origin, height, and width coordinates of the Matplotlib axes so that the resulting Matplotlib figure matches what is seen by the user when the layout is rendered in Inkscape.

Within a `figurefirst.FigureLayout` object, axes objects are organized within a grouping hierarchy specified by the SVG groups or Inkscape layers that enclose the tagged box. Like the axes, these groups and layers are exposed to FigureFirst using xml tags: `<figurefirst:group>` and `<figurefirst:figure>` respectively.

We use Inkscape layers as the top level of the grouping hierarchy. Each layer generatea a new Matplotlib figure instance that holds the enclosed `<figurefirst:axis>` objects, and the dimensions of these figures are determined by the dimensions of the SVG document. Additional levels of grouping are specified by tagging groups with the `<figurefirst:group>` tag. In the case that a `<figurefirst:figure>` tag is not indicated, all the axes of the document are collected into the default figure with the name `'none'`.

The `<figurefirst:figure>` tag can also be used at the level of groups and individual boxes to support figure templates. Templates allow a sub-layout prototype to be replicated multiple times within the context of a larger document. To use templates a group of `<figurefirst:axis>` boxes is tagged with a `<figurefirst:figure>` tag. This template is then targeted to single boxes that are tagged with the `<figurefirst:figure>` that contains a

<figurefirst:template> attribute indicating the name of the template to use. The template is subsequently scaled and translated to fit within the bounds of the target.

Summary and Future Directions

Matplotlib provides a rich and powerful low-level API that allows exquisite control over every aspect of a plot. Although high level interfaces such as subplot and gridspec that attempt to simplify the layout of a figure exist, these do not always meet the demands of a visualization problem. For example, consider Fig. 1 where we plot the raw data and marginal distributions from Fisher's iris dataset [Fisher36]. In Fig. 1A we use the gridspec API to construct a 2X4 grid, and then define the axes within the constraints of this grid. Compare this to Fig. 1B where we use figurefirst to plot into a layout. Not only does careful placing of the plotting axes make better use of the figure space, but the spacing emphasizes certain comparisons over others. Of course, it is entirely possible to construct a nearly identical figure using the Matploltib API, however this would require writing functions that manually specify each axis location or contain a considerable amount of layout logic. In addition to being rather lengthy, it would be difficult to write these functions in a way that generalizes across figures. In contrast, as shown below, only 27 lines of code were required to load the data and plot Fig. 1B using FigureFirst. Note that nearly all the styling information is encapsulated within the layout document. In fact, in the case of the marginal distributions, we use the names from the layout to index into our Python data structure (line 21), thus the layout even specifies what data to plot and where.

```
1  from sklearn import datasets
2  import numpy as np
3  import figurefirst as fifi
4  d = datasets.load_iris()
5  data = dict()
6  for n,v in zip(d.feature_names,d.data.T):
7      data[tuple(n.split()[:2][::-1])] = v
8  layout = fifi.FigureLayout('example_layout.svg')
9  layout.make_mplfigures()
10 kwa = layout.pathspecs['petal'].mplkwargs()
11 layout.axes['raw'].scatter(data['width','petal'],
12                             data['length','petal'],
13                             **kwa)
14 kwa = layout.pathspecs['sepal'].mplkwargs()
15 layout.axes['raw'].scatter(data['width','sepal'],
16                             data['length','sepal'],
17                             **kwa)
18 for key in layout.axes.keys() :
19     if key in data.keys():
20         kwa = layout.pathspecs[key[1]].mplkwargs()
21         counts,b = np.histogram(data[key],
22                                 np.arange(0,11))
23         layout.axes[key].fill_between(
24                         b[:-1]+0.5,0,counts,**kwa)
25 layout.apply_mpl_methods()
26 fifi.mpl_functions.set_spines(layout)
27 layout.save('example.svg')
```

The use of layout documents to structure graphical elements is common in many domains of computer science, including the design of graphical user interfaces and the organization of web pages. FigureFirst takes this concept and applies it to the construction of scientific figures. This approach makes it possible to update figures with new data independently (saving computational time). Often when working on a scientific figure early in the process, the overall layout and figure size is unknown. Or perhaps the figure needs to be reformatted for a different journal's size, or for a poster or slide format. With FigureFirst these changes are as easy as rearranging the rectangles in Inkscape, and rerunning the same code (Fig. 4D-E). This workflow exemplifies the key contribution of FigureFirst: separating figure layout from data analysis, so that the software is not cluttered with code to generate the layout, and allowing for quick reorganization.

Thus far, we have focused our development efforts on using FigureFirst in conjunction with Inkscape. Inkscape is convenient in that it is (a) open source, (b) has a strong feature set, (c) uses the open SVG standard, (d) is available for all major operating systems, and (e) has a built-in xml editor. In principle, however, any SVG-compatible graphical layout software can be used (e.g. Adobe Illustrator). In the future, we plan to test other user interfaces to help increase our user base. Adobe Illustrator unfortunately does not use the same open SVG standard as Inkscape, so adding full support for Illustrator will require signficant effort, though it is possible and we will continue to explore that direction. Furthermore, developing a Javascript-based SVG editor that could easily decorate a SVG file with FigureFirst tags could then be employed as a Jupyter notebook extension to facilitate quick FigureFirst layout creation within a Jupyter session. In the meantime, layouts can be created externally and the following code can be used to display the output.SVG in the notebook:

```
from IPython.display import display,SVG
display(SVG(output.svg))
```

Presently, the most serious performance issue with FigureFirst is that large Matplotlib collections are difficult for Inkscape to render efficiently. This can be circumvented by utilizing the Matplotlib axis method <set_rasterization_zorder(N)> to rasterize large collections of patches. Other SVG rendering engines, such as the ones used by Google Chrome and Adobe Illustrator, have fewer problems, suggesting that this is a solvable issue.

As described previously in the Additional SVG/Python Interoperability section, we have implemented a simple method of embedding Python traceback information into the output SVG generated by FigureFirst. Linking this traceback with online repositories and data will make it possible for readers to easily access the data and code in an organized way, rearrange the presentation for their own needs, or apply the same analysis to a new dataset. In this way, FigureFirst simultaneously decouples the tasks of layout, analysis, and data sharing, while keeping them intimately connected, making open science easy and hassle free.

Acknowledgements

We conceived and began work on FigureFirst in lab of Michael Dickinson at Caltech, supported by a grant (T.L.) from the National Science Foundation (IOS 1452510). Travel funding and future developement is also supported by the Moore-Sloan Data Science (F.v.B).

References

[Aird11] W. C. Aird. *Discovery of the cardiovascular system: from Galen to William Harvey.*, Journal of Thrombosis and Haemostasis, 9 (Suppl. 1): 118-129, July 2011. https://doi.org/10.1111/j.1538-7836.2011.04312.x

[Altert13] M Albert, J. Andler, T. Bah, P. Barbry-Blot, J. Barraud, B. Baxter *Inkscape.*, http://www.inkscape.org , 2013.

[Fisher36] R. A. Fisher *The use of multiple measurements in taxonomic problems.*, Ann. Hum. Genet. 7 (2): 179-188, 1936. https://doi.org/10.1111/j.1469-1809.1936.tb02137.x

[Hunter08] John D. Hunter. *Matplotlib: A 2D graphics environment.*, Computing In Science & Engineering 9.3: 90-95, 2007. https://doi.org/10.1109/MCSE.2007.55

[Jagannathan12] Arvind Krishnaa Jagannathan, Srikrishnan Suresh, and Vishal Gautham Venkataraaman. *A Canvas-Based Presentation Tool Using Scalable Vector Graphics.*, 2012 IEEE Fourth International Conference on Technology for Education. 2012. https://doi.org/10.1109/T4E.2012.35

[Lindsay17] T. H. Lindsay, A. Sustar and M. Dickinson, *The Function and Organization of the Motor System Controlling Flight Maneuvers in Flies.*, Curr Biol. 27(3):345-358, 2017. https://doi.org/10.1016/j.cub.2016.12.018

[Whitesides04] George M. Whitesides, *'Whitesides' group: writing a paper.*, Advanced Materials 16.15: 1375-1377. 2004. https://doi.org/10.1002/adma.200400767

Parallel Analysis in MDAnalysis using the Dask Parallel Computing Library

Mahzad Khoshlessan[‡], Ioannis Paraskevakos[§], Shantenu Jha[§], Oliver Beckstein[‡*]

Abstract—The analysis of biomolecular computer simulations has become a challenge because the amount of output data is now routinely in the terabyte range. We evaluated if this challenge can be met by a parallel map-reduce approach with the Dask parallel computing library for task-graph based computing coupled with our MDAnalysis Python library for the analysis of molecular dynamics (MD) simulations. We performed a representative performance evaluation, taking into account the highly heterogeneous computing environment that researchers typically work in together with the diversity of existing file formats for MD trajectory data. We found that the underlying storage system (solid state drives, parallel file systems, or simple spinning platter disks) can be a deciding performance factor that leads to data ingestion becoming the primary bottleneck in the analysis work flow. However, the choice of the data file format can mitigate the effect of the storage system; in particular, the commonly used Gromacs XTC trajectory format, which is highly compressed, can exhibit strong scaling close to ideal due to trading a decrease in global storage access load against an increase in local per-core CPU-intensive decompression. Scaling was tested on a single node and multiple nodes on national and local supercomputing resources as well as typical workstations. Although very good strong scaling could be achieved for single nodes, good scaling across multiple nodes was hindered by the persistent occurrence of "stragglers", tasks that take much longer than all other tasks, and whose ultimate cause could not be completely ascertained. In summary, we show that, due to the focus on high interoperability in the scientific Python eco system, it is straightforward to implement map-reduce with Dask in MDAnalysis and provide an in-depth analysis of the considerations to obtain good parallel performance on HPC resources.

Index Terms—MDAnalysis, High Performance Computing, Dask, Map-Reduce, MPI for Python

Introduction

MDAnalysis is a Python library that provides users with access to raw simulation data and enables structural and temporal analysis of molecular dynamics (MD) trajectories generated by all major MD simulation packages [GLB+16], [MADWB11]. MD trajectories are time series of positions (and sometimes also velocities) of the simulated atoms or particles; using statistical mechanics one can calculate experimental observables from these time series [FS02], [MM14]. The size of these trajectories is growing as the simulation times are being extended beyond micro-seconds and larger systems with increasing numbers of atoms are simulated. The amount of data to be analyzed is growing rapidly into the terabyte range and analysis is increasingly becoming a bottleneck in MD workflows [CR15]. Therefore, there is a need for high performance computing (HPC) approaches for the analysis of MD trajectory data [TRB+08], [RCI13].

MDAnalysis does not yet provide a standard interface for parallel analysis; instead, various existing parallel libraries such as Python multiprocessing, joblib, and mpi4py [DPS05], [DPKC11] are currently used to parallelize MDAnalysis-based code on a case-by-case basis. Here we evaluated performance for parallel map-reduce [DG08] type analysis with the Dask parallel computing library [Roc15] for task-graph based distributed computing on HPC and local computing resources. Although Dask is able to implement much more complex computations than map-reduce, we chose Dask for this task because of its ease of use and because we envisage using this approach for more complicated analysis applications whose parallelization cannot be easily expressed as a simple map-reduce algorithm.

As the computational task we performed a common task in the analysis of the structural dynamics of proteins: we computed the time series of the root mean squared distance (RMSD) of the positions of all C_α atoms to their initial coordinates at time 0; for each time step ("frame") in the trajectory, rigid body degrees of freedom (translations and rotations) have to be removed through an optimal structural superposition that minimizes the RMSD [MM14] (Figure 1). A range of commonly used MD file formats (CHARMM/NAMD DCD [BBIM+09], Gromacs XTC [AMS+15], Amber NCDF [CCD+05]) and different trajectory sizes were benchmarked.

We looked at different HPC resources including national

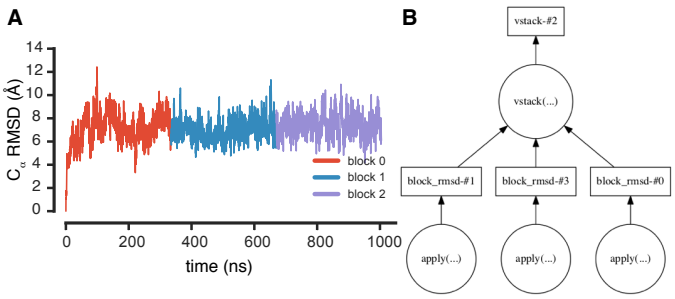


Fig. 1: *Calculation of the root mean square distance (RMSD) of a protein structure from the starting conformation via map-reduce with Dask.* ***A*** *RMSD as a function of time, with partial time series colored by trajectory block.* ***B*** *Dask task graph for splitting the RMSD calculation into three trajectory blocks.*

‡ *Arizona State University, Department of Physics, Tempe, AZ 85287, USA*
§ *RADICAL, ECE, Rutgers University, Piscataway, NJ 08854, USA*
∗ *Corresponding author: obeckste@asu.edu*

supercomputers (XSEDE TACC *Stampede* and SDSC *Comet*), university supercomputers (Arizona State University Research Computing *Saguaro*), and local resources (Gigabit networked multi-core workstations). The tested resources are parallel and heterogeneous with different CPUs, file systems, high speed networks and are suitable for high-performance distributed computing at various levels of parallelization. Different storage systems such as solid state drives (SSDs), hard disk drives (HDDs), network file system (NFS), and the parallel Lustre file system (using HDDs) were tested to examine the effect of I/O on the performance. The benchmarks were performed both on a single node and across multiple nodes using the multiprocessing and distributed schedulers in the Dask library.

We previously showed that the overall computational cost scales directly with the length of the trajectory, i.e., the weak scaling is close to ideal and is fairly independent from other factors [KB17]. Here we focus on the strong scaling behavior, i.e., the dependence of overall run time on the number of CPU cores used. Competition for access to the same file from multiple processes appears to be a bottleneck and therefore the storage system is an important determinant of performance. But because the trajectory file format dictates the data access pattern, overall performance also depends on the actual data format, with some formats being more robust against storage system specifics than others. Overall, good strong scaling performance could be obtained for a single node but robust across-node performance remained challenging. In order to identify performance bottlenecks we examined several other factors including the effect of striping in the parallel Lustre file system, over-subscribing (using more tasks than Dask workers), the performance of the Dask scheduler itself, and we also benchmarked an MPI-based implementation in contrast to the Dask approach. From these tests we tentatively conclude that poor across-nodes performance is rooted in contention on the shared network that may slow down individual tasks and lead to poor load balancing. Nevertheless, Dask with MDAnalysis appears to be a promising approach for high-level parallelization for analysis of MD trajectories, especially at moderate CPU core numbers.

Methods

We implemented a simple map-reduce scheme to parallelize processing of trajectories over contiguous blocks. We tested libraries in the following versions: MDAnalysis 0.15.0, Dask 0.12.0 (also 0.13.0), distributed 1.14.3 (also 1.15.1), and NumPy 1.11.2 (also 1.12.0) [VCV11].

```
import numpy as np
import MDAnalysis as mda
from MDAnalysis.analysis.rms import rmsd
```

The trajectory is split into `n_blocks` blocks with inital frame `start` and final frame `stop` set for each block. The calculation on each block (function `block_rmsd()`, corresponding to the *map* step) is *delayed* with the `delayed()` function in Dask:

```
from dask.delayed import delayed

def analyze_rmsd(ag, n_blocks):
    """RMSD of AtomGroup ag, parallelized n_blocks"""
    ref0 = ag.positions.copy()
    bsize = int(np.ceil(
                ag.universe.trajectory.n_frames \
                / float(n_blocks)))
    blocks = []
    for iblock in range(n_blocks):
        start, stop = iblock*bsize, (iblock+1)*bsize
        out = delayed(block_rmsd, pure=True)(
                ag.indices, ag.universe.filename,
                ag.universe.trajectory.filename,
                ref0, start, stop)
        blocks.append(out)
    return delayed(np.vstack)(blocks)
```

In the *reduce* step, the partial time series from each block are concatenated in the correct order (`np.vstack`, see Figure 1 A); because results from delayed objects are used, this step also has to be delayed.

As computational load we implement the calculation of the root mean square distance (RMSD) of the C_α atoms of the protein adenylate kinase [SB14] when fitted to a reference structure using an optimal rigid body superposition [MM14], using the qcprot implementation [LAT10] in MDAnalysis [GLB+16]. The RMSD is calculated for each trajectory frame in each block by iterating over `u.trajectory[start:stop]`:

```
def block_rmsd(index, topology, trajectory, ref0,
               start, stop):
    u = mda.Universe(topology, trajectory)
    ag = u.atoms[index]
    out = np.zeros([stop-start, 2])
    for i, ts in enumerate(
            u.trajectory[start:stop]):
        out[i, :] = ts.time, rmsd(ag.positions, ref0,
                    center=True, superposition=True)
    return out
```

Dask produces a task graph (Figure 1 B) and the computation of the graph is executed in parallel through a Dask scheduler such as `dask.multiprocessing` (or `dask.distributed`):

```
from dask.multiprocessing import get

u = mda.Universe(PSF, DCD)
ag = u.select_atoms("protein and name CA")
result = analyze_rmsd(ag, n_blocks)
timeseries = result.compute(get=get)
```

The complete code for benchmarking as well as an alternative implementation based on mpi4py is available from https://github.com/Becksteinlab/Parallel-analysis-in-the-MDAnalysis-Library under the MIT License.

The data files consist of a topology file `adk4AKE.psf` (in CHARMM PSF format; $N = 3341$ atoms) and a trajectory `1ake_007-nowater-core-dt240ps.dcd` (DCD format) of length 1.004 µs with 4187 frames; both are freely available from figshare at DOI 10.6084/m9.figshare.5108170 [SB17]. Files in XTC and NCDF formats are generated from the DCD on the fly using MDAnalysis. To avoid operating system caching, files were copied and only used once for each benchmark. All results for Dask distributed were obtained across three nodes on different clusters.

Trajectories with different number of frames per trajectory were analyzed to assess the effect of trajectory file size. These trajectories were generated by concatenating the base trajectory 50, 100, 300, and 600 times and are referred to as, e.g., "DCD300x" or "XTC600x". Run time was analyzed on single nodes (1–24 CPU cores) and up to three nodes (1–72 cores) as function of the number of cores (strong scaling behavior) and trajectory sizes (weak scaling). However, here we only present strong scaling data for the 300x and 600x trajectory sizes, which represent typical medium size results. For an analysis of the full data including weak scaling results set see the Technical Report [KB17].

The DCD file format is a binary representation for 32-bit floating point numbers (accuracy of positions about 10^{-6} Å) and

the DCD300x trajectory has a file size of 47 GB (DCD600x is twice as much); XTC is a lossy compressed format that effectively rounds floats to the second decimal (accuracy about 10^{-2} Å, which is sufficient for typical analysis) and XTC300x is only 15 GB. Amber NCDF is implemented with netCDF classic format version 3.6.0 (same accuracy as DCD) and trajectories are about the same size as DCD. DCD and NCDF natively allow fast random access to frames or blocks of frames, which is critical to implement the map-reduce algorithm. XTC does not natively support frame seeking but MDAnalysis implements a fast frame scanning algorithm for XTC files that caches all frame offsets and so enables random access for the XTC format, too [GLB+16]. In MDAnalysis 0.15.0, Amber NCDF files are read with the Python netCDF4 module that wraps the netcdf C library; in the upcoming MDAnalysis 0.17.0, netCDF v3 files are read with the pure Python `scipy.io.netcdf` module, which tends to read netCDF v3 files about five times faster than netCDF4, and hence results for NCDF presented here might change with more recent versions of MDAnalysis.

Performance was quantified by measuring the average time per trajectory frame to load data from storage into memory (I/O time per frame, $t_{I/O}$), the average time to complete the RMSD calculation (compute time per frame, t_{comp}), and the total wall time for job execution t_N when using N CPU cores. Strong scaling was assessed by calculating the speed up $S(N) = t_1/t_N$ and the efficiency $E(N) = S(N)/N$.

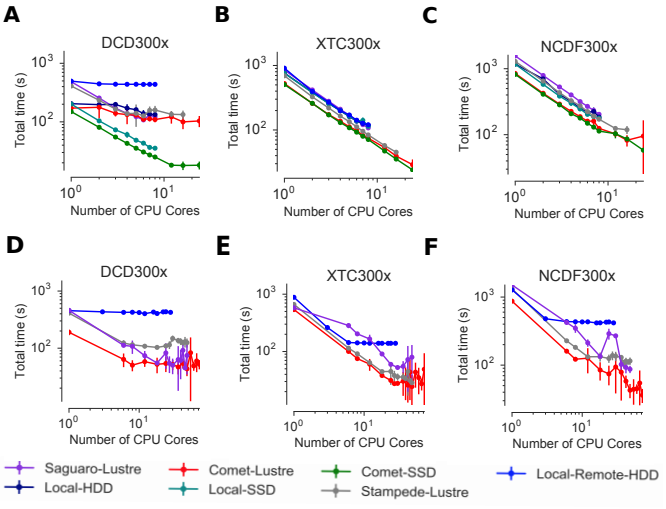


Fig. 2: *Comparison of total job execution time t_N for different file formats (300x trajectory size) using Dask multiprocessing on a* single node *(1–24 CPU cores, **A** – **C**) and Dask distributed on* up to three nodes *(1–72 CPU cores, **D** – **F**). The trajectory was split into M blocks and computations were performed using $N = M$ CPU cores. The runs were performed on different resources (ASU RC* Saguaro, *SDSC* Comet, *TACC* Stampede, local *workstations with different storage systems (locally attached* HDD, remote HDD *(via network file system, NFS), locally attached* SSD, Lustre *parallel file system with a single stripe). **A**, **D** CHARMM/NAMD DCD. **B**, **E** Gromacs XTC. **C**, **F** Amber NetCDF.*

Results and Discussion

Trajectories from MD simulations record snapshots of the positions of all particles at regular time intervals. A snapshot at a specified time point is called a frame. MDAnalysis only loads a single frame into memory at any time [GLB+16], [MADWB11] to allow the analysis of large trajectories that may contain, for example, $n_{frames} = 10^7$ frames in total. In a map-reduce approach, N processes will iterate in parallel over N chunks of the trajectory, each containing n_{frames}/N frames. Because frames are loaded serially, the run time scales directly with n_{frames} and the weak scaling behavior (as a function of trajectory length) is trivially close to ideal as seen from the data in [KB17]. Weak scaling with the system size also appears to be fairly linear, according to preliminary data (not shown). Therefore, in the following we focus exclusively on the harder problem of strong scaling, i.e., reducing the run time by employing parallelism.

Effect of File Format on I/O Performance

We first sought to quantify the effect of the trajectory format on the analysis performance. The overall run time depends strongly on the trajectory file format as well as the underlying storage system as shown for the 300x trajectories in Figure 2; results for other trajectory sizes are similar (see [KB17]) except for the smallest 50x trajectories where possibly caching effects tend to improve overall performance. Using DCD files with SSDs on a single node (Figure 2 A) is about one order of magnitude faster than the other formats (Figure 2 B, C) and scales near linearly for small CPU core counts ($N \leq 12$). However, DCD does not scale at all with other storage systems such as HDD or NFS and run time only improves up to $N = 4$ on the Lustre file system. On the other hand, the run time with NCDF and especially with XTC trajectories improves linearly with increasing N, with XTC on Lustre and $N = 24$ cores almost obtaining the best DCD run time of about 30

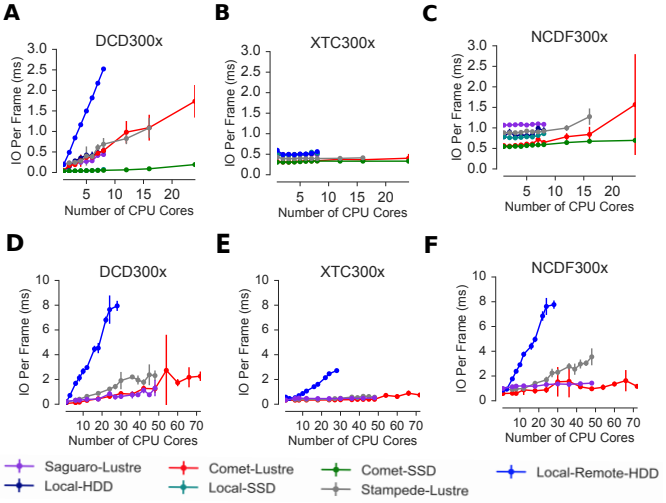


Fig. 3: *Comparison of I/O time $t_{I/O}$ per frame between different file formats (300x trajectory size) using Dask multiprocessing on a* single node *(**A** – **C**) and Dask distributed on* multiple nodes *(**D** – **F**). **A**, **D** CHARMM/NAMD DCD. **B**, **E** Gromacs XTC. **C**, **F** Amber NetCDF. All parameters as in Fig. 2.*

s (SSD, $N = 12$); at the highest single node core count $N = 24$, XTC on SSD performs even better (run time about 25 s). For larger N on multiple nodes, only a shared file system (Lustre or NFS) based on HDD was available. All three file formats only show small improvements in run time at higher core counts ($N > 24$) on the Lustre file system on supercomputers with fast interconnects and no improvements on NFS over Gigabit (Figure 2 D–F).

In order to explain the differences in performance and scaling of the file formats, we analyzed the time to load the coordinates

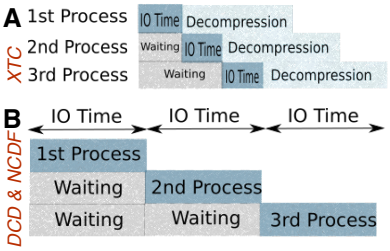


Fig. 4: *I/O pattern for reading frames in parallel from commonly used MD trajectory formats.* ***A*** *Gromacs XTC file format.* ***B*** *CHARMM/NAMD DCD file format and Amber NCDF format.*

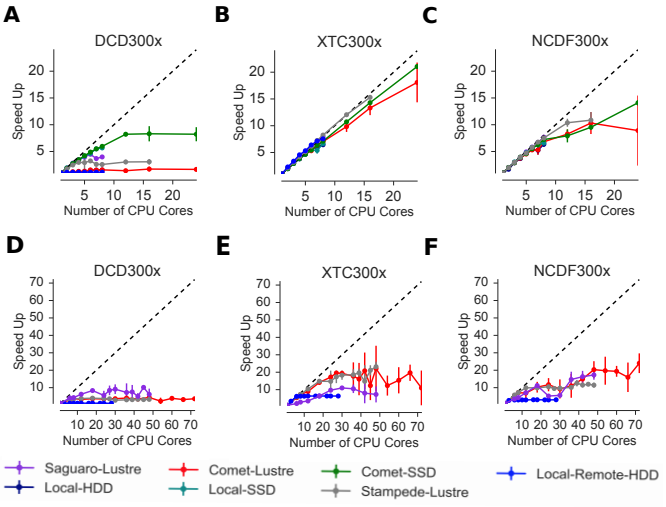


Fig. 5: *Speed-up S for the analysis of the 300x trajectory on HPC resources using Dask multiprocessing (single node,* ***A*** *–* ***C****) and distributed (up to three nodes,* ***D*** *–* ***F****). The dashed line shows the ideal limit of strong scaling. All other parameters as in Fig. 2.*

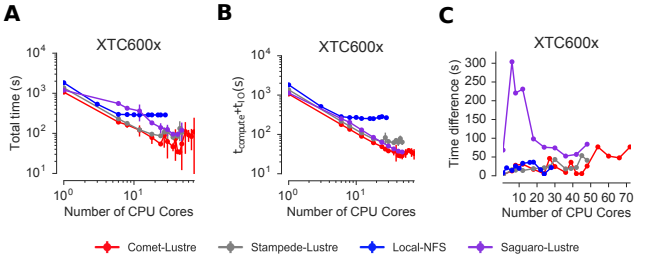


Fig. 6: *Detailed analysis of timings for the 600x XTC trajectory on HPC resources using Dask distributed. All other parameters as in Fig. 2.* ***A*** *Total time to solution (wall clock),* t_N *for N trajectory blocks using* $N_{cores} = N$ *CPU cores.* ***B*** *Sum of the I/O time per frame* $t_{I/O}$ *and the (constant) time for the RMSD computation* t_{comp} *(data not shown).* ***C*** *Difference* $t_N - n_{frames}(t_{I/O} + t_{comp})$*, accounting for the cost of communications and other overheads.*

of a single frame from storage into memory ($t_{\mathrm{I/O}}$) and the time to perform the computation on a single frame using the in-memory data (t_{comp}). As expected, t_{comp} is independent from the file format, n_{frames}, and N and only depends on the CPU type itself (mean and standard deviation on SDSC *Comet* 0.098 ± 0.004 ms, TACC *Stampede* 0.133 ± 0.000 ms, ASU RC *Saguaro* 0.174 ± 0.000 ms, local workstations 0.225 ± 0.022 ms, see [KB17]). Figure 3, however shows how $t_{\mathrm{I/O}}$ (for the 300x trajectories) varies widely and in most cases, is at least an order of magnitude larger than t_{comp}. The exception is $t_{\mathrm{I/O}}$ for the DCD file format using SSDs, which remains small (0.06 ± 0.04 ms on SDSC *Comet*) and almost constant with $N \leq 12$ (Figure 3 A) and as a result, the DCD file format shows good scaling and the best performance on a single node. For HDD-based storage, the time to read data from a DCD frame increases with the number of processes that are simultaneously trying to access the DCD file. XTC and NCDF show flat $t_{\mathrm{I/O}}$ with N on a single node (Figure 3 B, C) and even for multiple nodes, the time to ingest a frame of a XTC trajectory is almost constant, except for NFS, which broadly shows poor performance (Figure 3 E, F).

Depending on the file format the loading time of frames into memory will be different, as illustrated in Figure 4. The XTC file format is compressed and has a smaller file size when compared to the other formats. When a compressed XTC frame is loaded into memory, it is immediately decompressed (see Figure 4 A). During decompression by one process, the file system allows the next process to load its requested frame into memory. As a result, competition for file access between processes and overall wait time is reduced and $t_{\mathrm{I/O}}$ remains almost constant, even for large number of parallel processes (Figure 3 B, E). Neither DCD nor NCDF files are compressed and multiple processes compete for access to the file (Figure 4 B) although NCDF files is a more complicated file format than DCD and has additional computational overhead. Therefore, for DCD the I/O time per frame is very small as compared to other formats when the number of processes is small (and the storage is fast), but even at low levels of parallelization, $t_{\mathrm{I/O}}$ increases due to the overlapping of per frame trajectory data access (Figure 3 A, D). Data access with NCDF is slower but due to the additional computational overhead, is amenable to some level of parallelization (Figure 3 C, F).

Strong Scaling Analysis for Different File Formats

We quantified the strong scaling behavior by analyzing the speed-up $S(N)$; as an example, the 300x trajectories for multiprocessing and distributed schedulers are show in Figure 5. The DCD format exhibits poor scaling, except for $N \leq 12$ on a single node and SSDs (Figure 5 A, D) and is due to the increase in $t_{\mathrm{I/O}}$ with N, as discussed in the previous section. The XTC file format scales close to ideal on $N \leq 24$ cores (single node) for both the multiprocessing and distributed scheduler, almost independent from the underlying storage system. The NCDF file format only scales well up to 8 cores (Figure 5 C, F) as expected from $t_{\mathrm{I/O}}$ in Figure 3 C, F.

For the XTC file format, $t_{\mathrm{I/O}}$ is is nearly constant up to $N = 50$ cores (Figure 3 E) and t_{comp} also remains constant up to 72 cores. Therefore, close to ideal scaling would be expected for up to 50 cores, assuming that average processing time per frame $t_{\mathrm{comp}} + t_{\mathrm{I/O}}$ dominates the computation. However, based on Figure 5 E, the XTC format only scales well up to about 24 cores, which suggests that this assumption is wrong and there are other computational overheads.

To identify and quantify these additional overheads, we analyzed the performance of the XTC600x trajectory in more detail (Figure 6); results for other trajectory sizes are qualitatively similar. The total job execution time t_N differs from the total compute and I/O time, $N(t_{\mathrm{comp}} + t_{\mathrm{I/O}})$. This difference measures additional overheads that we did not consider so far. It increases with trajectory size for all file formats and for all machines (for details refer to [KB17]) but is smaller for SDSC *Comet* and TACC

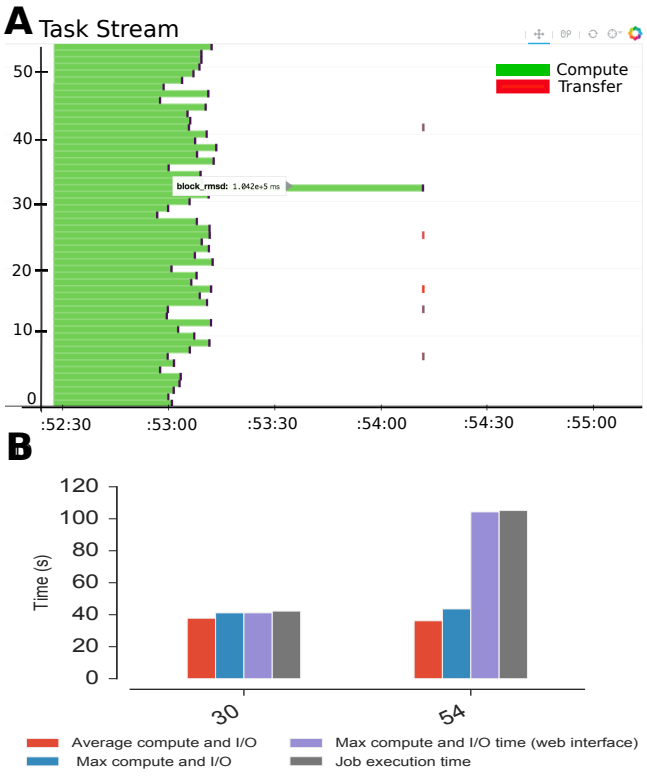


Fig. 7: *Evidence for uneven distribution of task execution times, shown for the XTC600x trajectory on SDSC* Comet *on the Lustre file system.* ***A*** *Task stream plot showing the fraction of time spent on different parts of the task by each worker, obtained using the Dask web-interface. (54 tasks for 54 workers that used $N = 54$ cores). Green bars ("Compute") represent time spent on RMSD calculations, including trajectory I/O, red bars show data transfer. A "straggler" task (#32) takes much longer than any other task and thus determines the total execution time.* ***B*** *Comparison between timing measurements from instrumentation inside the Python code (average compute and I/O time per task $n_{frames}/N\,(t_{comp}+t_{I/O})$, $\max[n_{frames}/N\,(t_{comp}+t_{I/O})]$, and t_N) and Dask web-interface for $N = 30$ and $N = 54$ cores.*

Stampede than compared to other machines. The difference is small for the results obtained using the multiprocessing scheduler on a single node but it is substantial for the results obtained using distributed scheduler on multiple nodes.

In order to obtain more insight into the underlying network behavior both at the Dask worker level and communication level and in order to pinpoint the origin of the overheads, we used the web-interface of the Dask library, which is launched together with the Dask scheduler. Dask task stream plots such as the example shown in Figure 7 A typically show one or more *straggler* tasks that take much more time than the other tasks and as a result slow down the whole run. Stragglers do not actually spend more time on the RMSD computation and trajectory I/O than other tasks, as shown by comparing the average compute and I/O time for a single task i, $n_{\text{frames}}/N(t_{\text{comp},i}+t_{\text{I/O},i})$, with the maximum over all tasks $\max_i[n_{\text{frames}}/N(t_{\text{comp},i}+t_{\text{I/O},i})]$ (Figure 7 B). These stragglers are observed at some repeats when the number of cores is more than 24. However, we do not always see these stragglers which shows the importance of collecting statistics and looking at the average value of several repeats (5 in the present study). For example, for $N = 30$ at one repeat no straggler was observed but, the statistics show poor perforrmance as also seen in Figure 6 A and B. However, as seen in the example for $N = 54$ for one repeat, the maximum compute and I/O time as measured inside the Python code is smaller than the maximum value extracted from the web-interface (and the Dask scheduler) (Figure 7 B). The maximum compute and I/O value from the scheduler matches the total measured run time, indicating that stragglers limit the overall performance of the run. The timing of the scheduler includes waiting due to network effects, which would explain why the difference is only visible when using multiple nodes where the node interconnect must be used.

Challenges for Good HPC Performance

All results were obtained during normal, multi-user, production periods on all machines, which means that jobs run times are affected by other jobs on the system. This is true even when the job is the only one using a particular node, which was the case in the present study. There are shared resources such as network file systems that all the nodes use. The high speed interconnect that enables parallel jobs to run is also a shared resource. The more jobs are running on the cluster, the more contention there is for these resources. As a result, the same job run at different times may take a different amount of time to complete, as seen in the fluctuations in task completion time across different processes. These fluctuations differ in each repeat and are dependent on the hardware and network. There is also variability in network latency, in addition to the variability in underlying hardware in each machine, which may also cause the results to vary across different machines. Since our map-reduce problem is pleasantly parallel, each or a subset of computations can be executed by independent processes. Furthermore, all of our processes have the same amount of work to do, namely one trajectory block per process, and therefore our problem should exhibit good load balancing. Therefore, observing the stragglers shown in Figure 7 A is unexpected and the following sections aim to identify possible causes for their occurrence.

Performance Optimization

We tested different features of the computing environment to identify causes of stragglers and to improve performance and robustness, focusing on the XTC file format as the most promising candidate so far. We tested the hypothesis that waiting for file access might lead to stalled tasks by increasing the effective number of accessible files through "striping" in the Lustre parallel file system. We investigated the hypothesis that the Dask distributed scheduler might be too slow to schedule the tasks and we looked at improved load balancing by over-subscribing Dask workers.

Effect of Lustre Striping: As discussed before, the overlapping of data requests from different processes can lead to higher I/O time and as a result poor performance. $t_{\text{I/O}}$ strongly affects performance since it is much larger than t_{comp} in all multi-node scenarios. Although the XTC format showed the best performance, for multiple nodes $t_{\text{I/O}}$ increased for it, too (Figure 3 E). In Lustre, a copy of the shared file can be in different physical storage devices (object storage targets, OSTs). Single shared files can have a stripe count equal to the number of nodes or processes which access the file. We set the stripe count equal to three, which is equal to the number of nodes used for our benchmark using the distributed scheduler. This might improve performance, since all the processes from each node will have a copy of the file and as a result the contention due to many data requests should decrease. Figure 8 show the speed up and I/O time per frame plots obtained

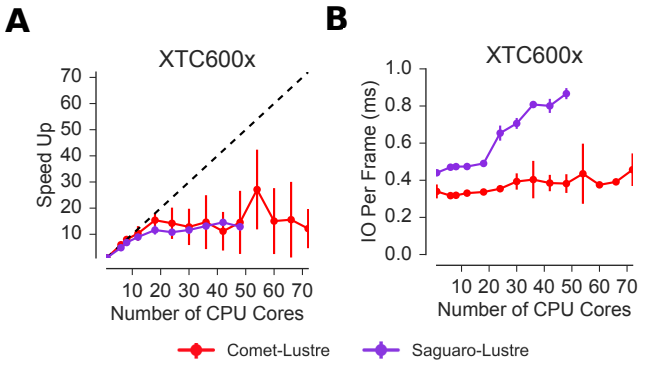


Fig. 8: *Effect of striping with the Lustre distributed file system. The XTC600x trajectory was analyzed on HPC resources (ASU RC* Saguaro*, SDSC* Comet*) with Dask distributed and a Lustre stripe count of three, i.e., data were replicated across three servers. One trajectory block was assigned to each worker, i.e., the number of tasks equaled the number of CPU cores.* ***A*** *Speed-up.* ***B*** *Average I/O time per frame, $t_{I/O}$.*

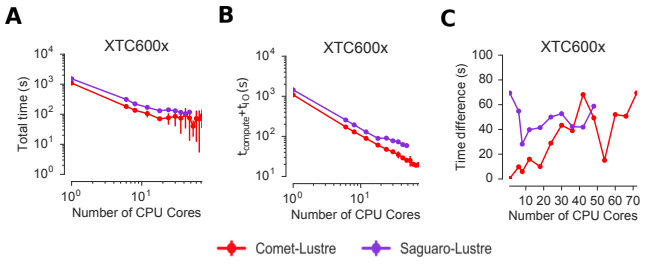


Fig. 9: *Detailed timings for three-fold Lustre striping (see Fig. 8 for other parameters).* ***A*** *Total time to solution (wall clock), t_N for M trajectory blocks using $N = M$ CPU cores.* ***B*** *$t_{comp} + t_{I/O}$, average sum of the I/O time ($t_{I/O}$, Fig. 8 B) and the (constant) time for the RMSD computation t_{comp} (data not shown).* ***C*** *Difference $t_N - n_{frames}(t_{I/O} + t_{comp})$, accounting for communications and overheads that are not directly measured.*

for XTC file format (XTC600x) when striping is activated. I/O time remains constant for up to 72 cores. Thus, striping improves $t_{I/O}$ and makes file access more robust. However, the timing plots in Figure 9 still show a time difference between average total compute and I/O time and job execution time that remains due to stragglers and as a result the overall speed-up is not improved.

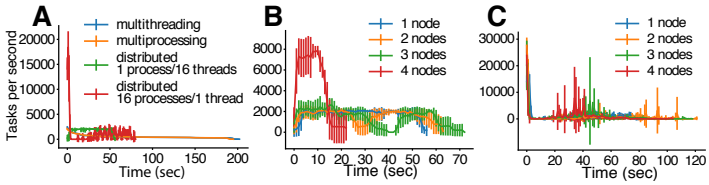


Fig. 10: *Benchmark of Dask scheduler throughput on TACC* Stampede*. Performance is measured by the number of empty* `pass` *tasks that were executed in a second. The scheduler had to launch 100,000 tasks and the run ended when all tasks had been run.* ***A*** *single node with different schedulers; multithreading and multiprocessing are almost indistinguishable from each other.* ***B*** *multiple nodes with the distributed scheduler and 1 worker process per node.* ***C*** *multiple nodes with the distributed scheduler and 16 worker processes per node.*

Scheduler Throughput: In order to test the hypothesis that straggler tasks were due to limitations in the speed of the Dask scheduler, we performed scheduling experiments with all Dask schedulers (multithreaded, multiprocessing and distributed) on TACC *Stampede* (16 CPU cores per node). In each run, a total of 100,000 zero workload (`pass`) tasks were executed in order to measure the maximum scheduling throughput; each run itself was repeated ten times and mean values together with standard deviations were reported. Figure 10 A shows the throughput of each scheduler over time on a single *Stampede* node, with Dask scheduler and worker being located on the same node. The most efficient scheduler is the distributed scheduler, which manages to schedule 20,000 tasks per second when there is one worker process for each available core. The distributed scheduler with just one worker process and a number of threads equal to the number of available cores has lower peak performance of about 2000 tasks/s and is able to schedule and execute these 100,000 tasks in 50 s. The multiprocessing and multithreading schedulers behave similarly, but need much more time (about 200 s) to finish compared to distributed.

Figure 10 B shows the distributed scheduler's throughput over time for increasing number of nodes when each node has a single worker process and each worker launches a thread to execute a task (maximum 16 threads per worker). No clear pattern for the throughput emerges, with values between 2000 and 8000 tasks/s. Figure 10 C shows the same execution with Dask distributed set up to have one worker process per core, i.e., 16 workers per node. The scheduler never reaches its steady throughput state, compared to Figure 10 B so that it is difficult to quantify the effect of the additional nodes. Although a peak throughput between 10,000 to 30,000 tasks/s is reported, overall scheduling is erratic and the total 100,000 tasks are not completed sooner than for the case with 1 worker per node with 16 threads. It appears that assigning one worker process to each core will speed up Dask's throughput but more work would need to be done to assess if the burst-like behavior seen in this case is an artifact of the zero workload test.

Either way, the distributed and even the multiprocessing scheduler are sufficiently fast as to not cause a bottleneck in our map-reduce problem and are probably not responsible for the stragglers.

Effect of Over-Subscribing: In order to make our code more robust against uncertainty in computation times we explored over-subscribing the workers, i.e., to submit many more tasks than the number of available workers (and CPU cores, using one worker per core). Over-Subscription might allow Dask to balance the load appropriately and as a result cover the extra time when there are some stragglers. We set the number M of tasks to be three times the number of workers, $M = 3N$, where the number of workers N equaled the number of CPU cores; now each task only works on n_{frames}/M frames. To reduce the influence of $t_{I/O}$ on the benchmark, Lustre-striping was activated and set to three, equal to the number of nodes used.

For XTC600x, no substantial speed-up is observed due to over-subscribing (compare Figure 11 A to 8 A), although fluctuations are reduced. As before, the I/O time is constant up to 72 cores due to striping (Figure 11 B). However, a time difference between average total compute and I/O time and job execution time (Figure 12) reveals that over-subscribing does not help to remove the stragglers and as a result the overall speed-up is not improved. Figure 13 shows a time comparison for different parts of the calculations. The overhead in the calculations is small up to 24 cores (single node). For lower N, the largest fraction of time is

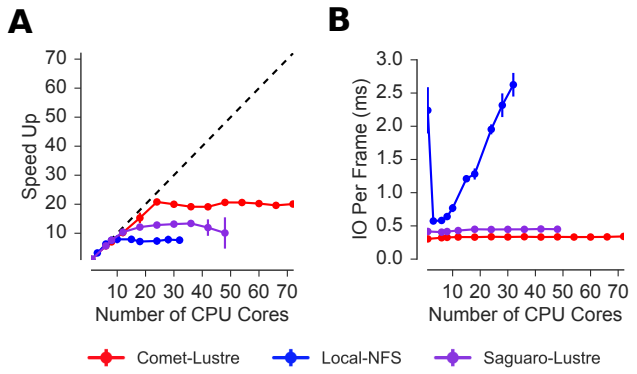


Fig. 11: *Effect of three-fold over-subscribing distributed workers. The XTC600x trajectory was analyzed on HPC resources (Lustre stripe count of three) and local NFS using Dask distributed where M number of trajectory blocks (tasks) is three times the number of worker processes, $M = 3N$, and there is one worker per CPU core.* ***A*** *Speed-up S.* ***B*** *I/O time $t_{I/O}$ per frame.*

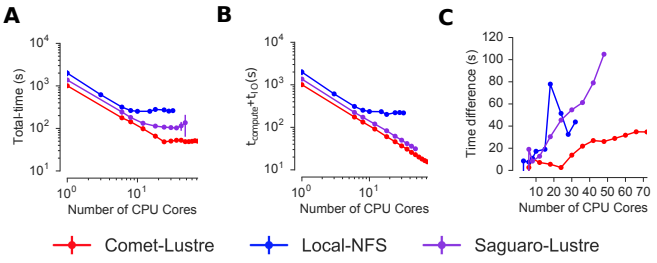


Fig. 12: *Detailed timings for three-fold over-subscribing distributed workers.* ***A*** *Total time to solution (wall clock), t_N.* ***B*** *$t_{comp} + t_{I/O}$, average sum of $t_{I/O}$ (Fig. 11 B) and the (constant) computation time t_{comp} (data not shown) per frame.* ***C*** *Difference $t_N - n_{frames}(t_{I/O} + t_{comp})$, accounting for communications and overheads that are not directly measured. Other parameters as in Fig. 11.*

spent on the calculation of RMSD arrays and I/O) (computation time) which decreases as the number of cores increases from 1 to 72. However, when extending to multiple nodes the time for overheads and communication increases, which reduces the overall performance.

In order to better quantify the scheduling decisions and to have verification of stragglers independent from the Dask web interface, we implemented a Dask scheduler reporter plugin (freely available from https://github.com/radical-cybertools/midas), which captures task execution events from the scheduler and their respective timestamps. We analyzed the execution of XTC300x on TACC *Stampede* with three-fold over-subscription ($M = 3N_{\text{cores}}$) and measured how many tasks were submitted per worker process. Table 1 shows that although most workers executed three tasks as would be expected for three-fold over-subscription, between 0 and 17% executed four tasks and others only one or two. This variability is also borne out in detail by Figure 14, which shows how RMSD blocks were submitted per worker process in each run. Therefore, over-subscription does not necessarily lead to a balanced execution and might add additional execution time; unfortunately, over-subscription does not get rid of the straggler tasks.

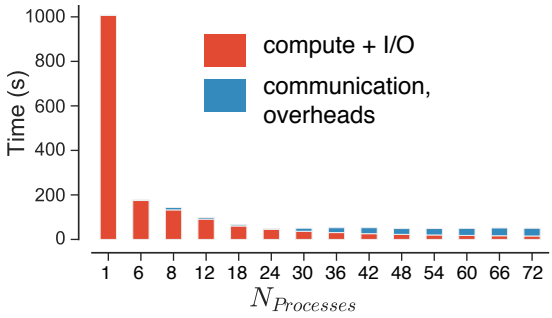


Fig. 13: *Time comparison for three-fold over-subscribing distributed workers (XTC600x on SDSC* Comet *on Lustre with stripe count three). Bars indicate the mean total execution time t_N (averaged over five repeats) as a function of available worker processes, with one worker per CPU core. Time for compute + I/O (red, see Fig. 12 B) dominates for smaller core counts (up to one node, 24) but is swamped by communication (time to gather the RMSD arrays computed by each worker for the reduction) and overheads (blue, see see Fig. 12 C) beyond a single node.*

RMSD Blocks	Run 1	Run 2	Run 3	Run 4	Run 5
1	0	0	1	0	0
2	8	5	7	7	2
3	48	54	47	50	60
4	8	5	9	7	2

TABLE 1: *Number of worker processes that executed 1, 2, 3, or 4 of tasks (RMSD calculation over one trajectory block) per run. Executed on TACC* Stampede *utilizing 64 cores*

Comparison of Performance of Map-Reduce Job Between MPI for Python and Dask Frameworks

The investigations so far indicated that stragglers are responsible for poor scaling beyond a single node. These delayed processes were observed on three different HPC systems and on different days, so they are unlikely to be infrastructure specific. In order to rule out the hypothesis that Dask is inherently limited in its applicability to our problem we re-implemented our map-reduce problem with MPI based on the Python mpi4py [DPS05], [DPKC11] module. The comparison was performed with the XTC600x trajectory on SDSC *Comet*.

The overall performance is very similar to the Dask implementation: it scales almost ideally up to 24 CPU cores (a single node)

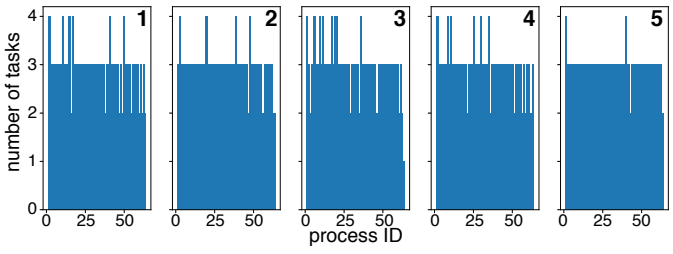


Fig. 14: *Task Histogram of RMSD with MDAnalysis and Dask with XTC 300x over 64 cores on Stampede with 192 trajectory blocks. Each histogram corresponds to an independent repeat of the same computational experiment. For each worker process ID, the number of tasks submitted to that process is shown.*

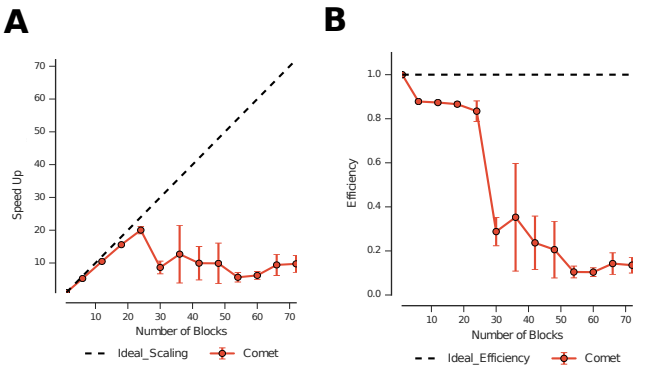


Fig. 15: ***A*** *Speed-up and* ***B*** *efficiency plots for benchmark performed on XTC600x on SDSC* Comet *using MPI for Python. Five repeats are run for each block size and the reported values are the mean values and standard deviations.*

but then drops to a very low efficiency (Figure 15). A detailed analysis of the time spent on computation versus communication (Figure 16 A) shows that the communication and overheads are negligible up to 24 cores (single node) and only moderately increases for larger N. The largest fraction of the calculations is always spent on the calculation of RMSD arrays with I/O (computation time). Although the computation time decreases with increasing number of cores for a single node, it increases again when increasing N further, in a pattern similar to what we saw earlier for Dask.

Figure 16 B compares the execution times across all MPI ranks for 72 cores. There are several processes that are about ten times slower than the majority of processes. These stragglers reduce the overall performance and are always observed when the number of cores is more than 24 and the ranks span multiple nodes. Based on the results from MPI for Python, Dask is probably not responsible for the occurrence of the stragglers.

We finally also wanted to ascertain that variable execution time is not a property of the computational task itself and replaced the RMSD calculation with optimal superposition (based on the iterative qcprot algorithm [LAT10]) with a completely different, fully deterministic metric, namely a simple all-versus-all distance calculation based on MDAnalysis.lib.distances.distance_array. The distance array calculates all distances between the reference coordinates at time 0 and the coordinates of the current frame and provides a comparable computational load. Even with the new metric the same behavior was observed in the MPI implementation (data not shown) and hence we can conclude that the qcprot RMSD calculation is not the reason why we are seeing the stragglers.

Conclusions

Dask together with MDAnalysis makes it straightforward to implement parallel analysis of MD trajectories within a map-reduce scheme. We show that obtaining good parallel performance depends on multiple factors such as storage system and trajectory file format and provide guidelines for how to optimize trajectory analysis throughput within the constraints of a heterogeneous research computing environment. Performance on a single node can be close to ideal, especially when using the XTC trajectory format that trades I/O for CPU cycles through aggressive compression, or when using SSDs with any format. However, obtaining

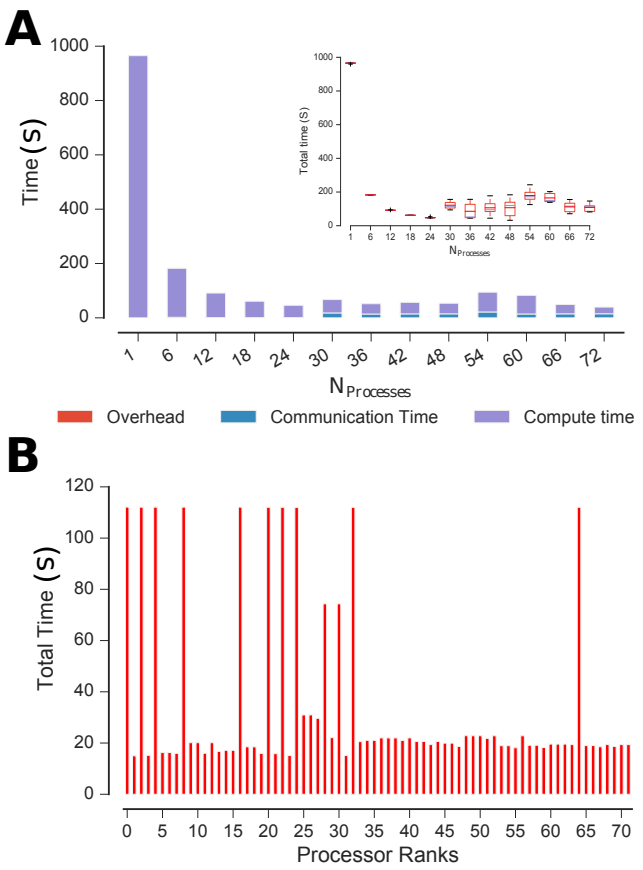


Fig. 16: ***A*** *Time comparison on different parts of the calculations obtained using MPI for Python. In this aggregate view, the time spent on different parts of the calculation are combined for different number of processes tested. The bars are subdivided into different contributions (compute (RMSD computation and I/O), communication, remaining overheads), with the total reflecting the overall run time. Reported values are the mean values across 5 repeats.* ***A inset*** *Total job execution time along with the mean and standard deviations across 5 repeats. The calculations are performed on XTC 600x using SDSC Comet.* ***B*** *Comparison of job execution time across processor ranks for 72 CPU cores obtained using MPI for python. There are several stragglers that slow down the whole process.*

good strong scaling beyond a single node was hindered by the occurrence of stragglers, one or few tasks that would take much longer than all the other tasks. Further studies are necessary to identify the underlying reason for the stragglers observed here; they are not due to Dask or the specific computational test case, and they cannot be circumvented by over-subscribing. Thus, implementing robust parallel trajectory analysis that scales over many nodes remains a challenge.

Acknowledgments

MK and IP were supported by grant ACI-1443054 from the National Science Foundation. SJ and OB were supported in part by grant ACI-1443054 from the National Science Foundation. Computational resources 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 and allocation TG-MCB090174 to SJ) and by Arizona State University Research Computing.

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.

[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.

[CR15] T. Cheatham and D. Roe. The impact of heterogeneous computing on workflows for biomolecular simulation and analysis. *Computing in Science Engineering*, 17(2):30–39, 2015. doi:10.1109/MCSE.2015.7.

[DG08] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008. doi:10.1145/1327452.1327492.

[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. doi:10.1016/j.advwatres.2011.04.013.

[DPS05] Lisandro Dalcín, Rodrigo Paz, and Mario Storti. Mpi for python. *Journal of Parallel and Distributed Computing*, 65(9):1108 – 1115, 2005. doi:10.1016/j.jpdc.2005.03.010.

[FS02] Daan Frenkel and Berend Smit. *Understanding Molecular Simulations*. Academic Press, San Diego, 2 edition, 2002.

[GLB+16] Richard J. Gowers, Max Linke, Jonathan Barnoud, Tyler J. E. Reddy, Manuel N. Melo, Sean L. Seyler, David L Dotson, Jan Domański, Sébastien Buchoux, Ian M. Kenney, and Oliver Beckstein. MDAnalysis: A Python package for the rapid analysis of molecular dynamics simulations. In Sebastian Benthall and Scott Rostrup, editors, *Proceedings of the 15th Python in Science Conference*, pages 102 – 109, Austin, TX, 2016. SciPy. URL: http://mdanalysis.org.

[KB17] Mahzad Khoshlessan and Oliver Beckstein. Parallel analysis in the MDAnalysis library: Benchmark of trajectory file formats. Technical report, Arizona State University, Tempe, AZ, 2017. doi:10.6084/m9.figshare.4695742.

[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.

[MM14] Cameron Mura and Charles E. McAnany. An introduction to biomolecular simulations and docking. *Molecular Simulation*, 40(10-11):732–764, 2014. doi:10.1080/08927022.2014.935372.

[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.

[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.

[SB14] Sean L Seyler and Oliver Beckstein. Sampling of large conformational transitions: Adenylate kinase as a testing ground. *Molec. Simul.*, 40(10–11):855–877, 2014. doi:10.1080/08927022.2014.919497.

[SB17] Sean Seyler and Oliver Beckstein. Molecular dynamics trajectory for benchmarking MDAnalysis, 6 2017. URL: https://figshare.com/articles/Molecular_dynamics_trajectory_for_benchmarking_MDAnalysis/5108170, doi:10.6084/m9.figshare.5108170.

[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.

[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.

MatchPy: A Pattern Matching Library

Manuel Krebber[‡*], Henrik Bartels[‡], Paolo Bientinesi[‡]

Abstract—Pattern matching is a powerful tool for symbolic computations, based on the well-defined theory of term rewriting systems. Application domains include algebraic expressions, abstract syntax trees, and XML and JSON data. Unfortunately, no lightweight implementation of pattern matching as general and flexible as Mathematica exists for Python [Pö16], [Hao14], [Sch14], [Jen15]. Therefore, we created the open source module MatchPy which offers similar pattern matching functionality in Python using a novel algorithm which finds matches for large pattern sets more efficiently by exploiting similarities between patterns.

Index Terms—pattern matching, symbolic computation, discrimination nets, term rewriting systems

Introduction

Pattern matching is a powerful tool which is part of many functional programming languages as well as computer algebra systems such as Mathematica. It is useful for many applications including symbolic computation, term simplification, term rewriting systems, automated theorem proving, and model checking. In this paper, we present a Python-based pattern matching library and its underlying algorithms.

The goal of pattern matching is to find a match substitution given a subject term and a pattern which is a term with placeholders [BN98]. The substitution maps placeholders in the pattern to replacement terms. A match is a substitution that can be applied to the pattern yielding the original subject. As an example consider the subject $f(a)$ and the pattern $f(x)$ where x is a placeholder. Then the substitution $\sigma = \{x \mapsto a\}$ is a match because $\sigma(f(x)) = f(a)$. This form of pattern matching without any special properties of function symbols is called syntactic matching. For syntactic patterns, the match is unique if it exists.

Among the existing systems, Mathematica [WR16] arguably offers the most expressive pattern matching. Its pattern matching offers similar expressiveness as Python's regular expressions, but for symbolic tree structures instead of strings. While pattern matching can handle nested expressions up to arbitrary depth, regular expressions cannot properly handle such nesting. Patterns are used widely in Mathematica, e.g. in function definitions or for manipulating terms. It is possible to define custom function symbols which can be associative and/or commutative. Mathematica also offers sequence variables which can match a sequence of terms instead of a single one. These are especially useful when working with variadic function symbols. Mathics [Pö16] is an open source computer algebra system written in Python that aims to replicate the syntax and functionality of Mathematica.

To our knowledge, no existing work covers pattern matching with function symbols which are either commutative or associative but not both at the same time. However, there are functions with those properties, e.g. matrix multiplication or arithmetic mean. Most of the existing pattern matching libraries for Python only support syntactic patterns. Associativity, commutativity and sequence variables make multiple distinct matches possible for a single pattern. In addition, pattern matching with either associativity or commutativity is NP-complete in both cases [BKN87]. While the pattern matching in SymPy [MSP+17] can work with associative/commutative functions, it is limited to finding a single match. Nonetheless, for some applications it is interesting to find all possible matches for a pattern, e.g. because matches need to be processed further recursively to solve an optimization problem. Furthermore, SymPy does not support sequence variables and is limited to a predefined set of mathematical operations.

In many applications, a fixed set of patterns is matched repeatedly against different subjects. The simultaneous matching of multiple patterns is called many-to-one matching, as opposed to one-to-one matching which denotes matching with a single pattern. Many-to-one matching can gain a significant speed increase compared to one-to-one matching by exploiting similarities between patterns. This has already been the subject of research for both syntactic [Chr93], [Grä91], [NWE97] and associative-commutative pattern matching [KL91], [BCR93], [LM94], [BCR+95], [Eke95], [KM01], but not with the full feature set described above. Discrimination nets [BW84] are the state-of-the-art solution for many-to-one matching. Our goal is to generalize this approach to support all the aforementioned features.

In this paper, we present the open-source library for Python MatchPy which provides pattern matching with sequence variables and associative/commutative function symbols. In addition to standard one-to-one matching, MatchPy also includes an efficient many-to-one matching algorithm that uses generalized discrimination nets. First, we give an overview of what MatchPy can be used for. Secondly, we explain some of the challenges arising from the non-syntactic pattern matching features and how we solve them. Then we give an overview of how many-to-one matching is realized and optimized in MatchPy. Next, we present our experiments where we observed significant speedups of the many-to-one matching over one-to-one matching. Finally, we draw some conclusions from the experiments and propose future work on MatchPy.

* *Corresponding author: manuel.krebber@rwth-aachen.de*
‡ *RWTH Aachen University, AICES, HPAC Group*

Usage Overview

MatchPy can be installed using `pip` and all necessary classes can be imported from the toplevel module `matchpy`. Expressions in MatchPy consist of constant symbols and operations. For patterns, wildcards can also be used as placeholders. We use Mathematica's notation for wildcards, i.e. we append underscores to wildcard names to distinguish them from symbols.

MatchPy can be used with native Python types such as `list` and `int`. The following is an example of how the subject `[0, 1]` can be matched against the pattern `[x_, 1]`. The expected match here is the replacement `0` for `x_`. We use *next* because we only want to use the first (and in this case only) match of the pattern:

```
>>> x_ = Wildcard.dot('x')
>>> next(match([0, 1], Pattern([x_, 1])))
{'x': 0}
```

In addition to regular (dot) variables, MatchPy also supports sequence wildcards. They can match a sequence of arguments and we denote them with two or three trailing underscores for plus and star wildcards, respectively. Star wildcards can match an empty sequence, while plus wildcards require at least one argument to match. The terminology is borrowed from regular expressions where *, + and . are used for similar concepts.

```
>>> y___ = Wildcard.star('y')
>>> next(match([1, 2, 3], Pattern([x_, y___])))
{'x': 1, 'y': (2, 3)}
```

In the following, we omit the definition of new variables as they can be done in the same way. In addition to native types, one can also define custom operations by creating a subclass of the `Operation` class:

```
class MyOp(Operation):
  name = 'MyOp'
  arity = Arity.variadic
  associative = True
  commutative = True
```

The name is a required attribute, while the others are optional and influence the behavior of the operations. By default, operations are variadic and neither commutative nor associative. Nested associative operations have to be variadic and are automatically flattened. Furthermore, regular variables behave similar to sequence variables as arguments of associative functions, because the associativity allows arbitrary parenthesization of arguments:

```
>>> next(match(MyOp(0, 1, 2), Pattern(MyOp(x_, 2))))
{'x': MyOp(0, 1)}
```

The argument of commutative operations are automatically sorted. Note that patterns with commutative operations can have multiple matches, because their arguments can be reordered arbitrarily.

```
>>> list(match(MyOp(1, 2), Pattern(MyOp(x_, z_))))
[{'x': 2, 'z': 1}, {'x': 1, 'z': 2}]
```

We can use the `CustomConstraint` class to create a constraint that checks whether `a` is smaller than `b`:

```
a_lt_b = CustomConstraint(lambda a, b: a < b)
```

The lambda function gets called with the variable substitutions based on their name. The order of arguments is not important and it is possible to only use a subset of the variables in the pattern. With this constraint we can define a replacement rule that basically describes bubble sort:

```
>>> pattern = Pattern([h___, b_, a_, t___], a_lt_b)
>>> rule = ReplacementRule(pattern,
                lambda a, b, h, t: [*h, a, b, *t])
```

Operation	Symbol	Arity	Properties
Multiplication	$\times$	variadic	associative
Addition	$+$	variadic	associative, commutative
Transposition	T	unary	
Inversion	-1	unary	
Inversion and Transposition	$-T$	unary	

TABLE 1: *Linear Algebra Operations*

The replacement function gets called with all matched variables as keyword arguments and needs to return the replacement. This replacement rule can be used to sort a list when applied repeatedly with `replace_all`:

```
>>> replace_all([1, 4, 3, 2], [rule])
[1, 2, 3, 4]
```

Sequence variables can also be used to match subsequences that match a constraint. For example, we can use the this feature to find all subsequences of integers that sum up to 5. In the following example, we use anonymous wildcards which have no name and are hence not part of the match substitution:

```
>>> x_sums_to_5 = CustomConstraint(
...                         lambda x: sum(x) == 5)
>>> pattern = Pattern([___, x__, ___], x_sums_to_5)
>>> list(match([1, 2, 3, 1, 1, 2], pattern))
[{'x': (2, 3)}, {'x': (3, 1, 1)}]
```

More examples can be found in MatchPy's documentation [Kre17a].

Application Example: Finding matches for a BLAS kernel

BLAS is a collection of optimized routines that can compute specific linear algebra operations efficiently [LHKK79], [DCHH88], [DCHD90]. As an example, assume we want to match all subexpressions of a linear algebra expression which can be computed by the ?TRMM BLAS routine. These have the form $\alpha \times op(A) \times B$ or $\alpha \times B \times op(A)$ where $op(A)$ is either the identity function or transposition, and A is a triangular matrix. For this example, we leave out all variants where $\alpha \neq 1$.

In order to model the linear algebra expressions, we use the operations shown in Table 1. In addition, we have special symbol subclasses for scalars, vectors and matrices. Matrices also have a set of properties, e.g. they can be triangular, symmetric, square, etc. For those patterns we also use a special kind of dot variable which is restricted to only match a specific kind of symbol. Finally, we construct the patterns using sequence variables to capture the remaining operands of the multiplication:

```
A_ = Wildcard.symbol('A', Matrix)
B_ = Wildcard.symbol('B', Matrix)
A_is_triangular = CustomConstraint(
  lambda A: 'triangular' in A.properties)

trmm_patterns = [
  Pattern(Times(h___, A_, B_, t___),
    A_is_triangular),
  Pattern(Times(h___, Transpose(A_), B_, t___),
    A_is_triangular),
  Pattern(Times(h___, B_, A_, t___),
    A_is_triangular),
  Pattern(Times(h___, B_, Transpose(A_), t___),
    A_is_triangular),
]
```

With these patterns, we can find all matches for the ?TRMM routine within a product. In this example, `M1`, `M2` and `M3` are matrices, but only `M3` is triangular:

```
>>> expr = Times(Transpose(M3), M1, M3, M2)
>>> for i, pattern in enumerate(trmm_patterns):
...     for substitution in match(expr, pattern):
...         print('{} with {}'.format(i, substitution))
0 with {A -> M3, B -> M2, t -> (), h -> ((M3)^T, M1)}
1 with {A -> M3, B -> M1, t -> (M3, M2), h -> ()}
2 with {A -> M3, B -> M1, t -> (M2), h -> ((M3)^T)}
```

As can be seen in the output, a total of three matches are found.

Design Considerations

There are plenty of implementations of syntactic matching and the algorithms are well known. Implementing pattern matching for MatchPy poses some challenges such as associativity and commutativity.

Associativity/Sequence variables

Associativity enables arbitrary grouping of arguments for matching: For example, `1 + a + b` matches `1 + x_` with $\{x \mapsto a+b\}$ because we can group the arguments as `1 + (a + b)`. Basically, when regular variables are arguments of an associative function, they behave like sequence variables. Both can result in multiple distinct matches for a single pattern. In contrast, for syntactic patterns there is always at most one match. This means that the matching algorithm needs to be non-deterministic to explore all potential matches for associative terms or terms with sequence variables. We employ backtracking with the help of Python generators to enable this. Associative matching is NP-complete [BKN87].

Commutativity

Matching commutative terms is difficult because matches need to be found independent of the argument order. Commutative matching has been shown to be NP-complete, too [BKN87]. It is possible to find all matches by matching all permutations of the subjects arguments against all permutations of the pattern arguments. However, with this naive approach, a total of $n!m!$ combinations have to be matched where n is the number of subject arguments and m the number of pattern arguments. It is likely that most of these combinations do not match or yield redundant matches.

Instead, we interpret the arguments as a multiset, i.e. an orderless collection that allows repetition of elements. Also, we use the following order for matching the subterms of a commutative term:

1) Constant arguments
2) Matched variables, i.e. variables that already have a value assigned in the current substitution
3) Non-variable arguments
4) Repeat step 2
5) Regular variables
6) Sequence variables

Each of those steps reduces the search space for successive steps. This also means that if one step finds no match, the remaining steps do not have to be performed. Note that steps 3, 5 and 6 can yield multiple matches and backtracking is employed to check every combination. Since step 6 is the most involved, it is described in more detail in the next section.

Sequence Variables in Commutative Functions

The distribution of n subjects subterms onto m sequence variables within a commutative function symbol can yield up to m^n distinct solutions. Enumerating all of the solutions is accomplished by generating and solving several linear Diophantine equations. As an example, lets assume we want to match `f(a, b, b, b)` with `f(x___, y___, y___)` where `f` is commutative. This means that the possible distributions are given by the non-negative integer solutions of these equations:

$$\begin{aligned} 1 &= x_a + 2y_a \\ 3 &= x_b + 2y_b \end{aligned}$$

x_a determines how many times `a` is included in the substitution for `x`. Because `y___` requires at least one term, we have the additional constraint $y_a + y_b \geq 1$. The only possible solution $x_a = x_b = y_b = 1 \wedge y_a = 0$ corresponds to the match substitution $\{x \mapsto (a,b), y \mapsto (b)\}$.

Extensive research has been done on solving linear Diophantine equations and linear Diophantine equation systems [Wei60], [Bon67], [Lam88], [CF89], [AHL00]. In our case the equations are actually independent expect for the additional constraints for plus variables. Also, the non-negative solutions can be found more easily. We use an adaptation of the algorithm used in SymPy which recursively reduces any linear Diophantine equation to equations of the form $ax + by = d$. Those can be solved efficiently with the Extended Euclidian algorithm [MVO96]. Then the solutions for those can be combined into a solution for the original equation.

All coefficients in those equations are likely very small since they correspond to the multiplicity of sequence variables. Similarly, the number of variables in the equations is usually small as they map to sequence variables. The constant is the multiplicity of a subject term and hence also usually small. Overall, the number of distinct equations that are solved is small and the solutions are cached. This reduces the impact of the sequence variables on the overall run time.

Optimizations

Since most applications for pattern matching repeatedly match a fixed set of patterns against multiple subjects, we implemented many-to-one matching for MatchPy. The goal of many-to-one matching is to utilize similarities between patterns to match them more efficiently. In this section, we give a brief overview of the many-to-one matching algorithm used by MatchPy. Full details can be found in the master thesis [Kre17b].

Many-to-one Matching

MatchPy includes two additional algorithms for matching: `ManyToOneMatcher` and `DiscriminationNet`. Both enable matching multiple patterns against a single subject much faster than matching each pattern individually using `match`. The latter can only be used for syntactic patterns and implements a state-of-the-art deterministic discrimination net. A discrimination net is a data structure similar to a decision tree or a finite automaton [Chr93], [Grä91], [NWE97]. The `ManyToOneMatcher` utilizes a generalized form of non-deterministic discrimination nets that support sequence variables and associative function symbols. Furthermore, as elaborated in the next section, it can also match commutative terms.

In Figure 1, an example for a non-deterministic discrimination net is shown. It contains three patterns that match Python lists:

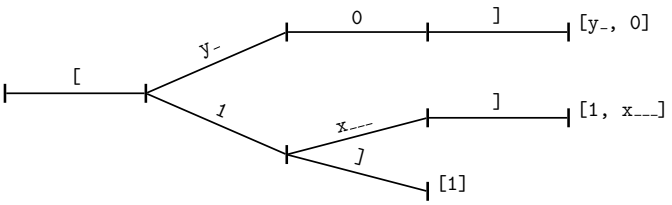


Fig. 1: *Example Discrimination Net.*

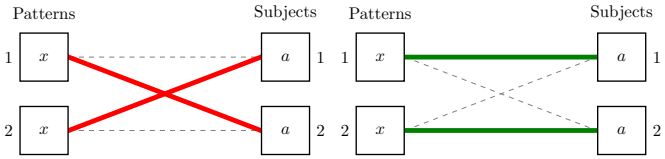


Fig. 2: *Example for Order in Bipartite Graph.*

One matches the list that consists of a single 1, the second one matches a list with exactly two elements where the last element is 0, and the third pattern matches any list where the first element is 1. Note, that these patterns can also match nested lists, e.g. the second pattern would also match `[[2, 1], 0]`.

Matching starts at the root and proceeds along the transitions. Simultaneously, the subject is traversed in preorder and each symbol is checked against the transitions. Only transitions matching the current subterm can be used. Once a final state is reached, its label gives a list of matching patterns. For non-deterministic discrimination nets, all possibilities need to be explored via backtracking. The discrimination net allows to reduce the matching costs, because common parts of different pattern only need to be matched once. For non-matching transitions, their whole subtree is pruned and all the patterns are excluded at once, further reducing the match cost.

In Figure 1, for the subject `[1, 0]`, there are two paths and therefore two matching patterns: `[y_, 0]` matches with $\{y \mapsto 1\}$ and `[1, x___]` matches with $\{x \mapsto 0\}$. Both the `y`-transition and the `1`-transition can be used in the second state to match a `1`.

Compared to existing discrimination net variants, we added transitions for the end of a compound term to support variadic functions. Furthermore, we added support for both associative function symbols and sequence variables. Finally, our discrimination net supports transitions restricted to symbol classes (i.e. `Symbol` subclasses) in addition to the ones that match just a specific symbol. We decided to use a non-deterministic discrimination net instead of a deterministic one, since the number of states of the later would grow exponentially with the number of patterns. While the `DiscriminationNet` also has support for sequence variables, in practice the net became to large to use with just a dozen patterns.

Commutative Many-to-one Matching

Many-to-one matching for commutative terms is more involved. We use a nested `CommutativeMatcher` which in turn uses another `ManyToOneMatcher` to match the subterms. Our approach is similar to the one used by Bachmair and Kirchner in their respective works [BCR+95], [KM01]. We match all the subterms of the commutative function in the subject with a many-to-one matcher constructed from the subpatterns of the commutative function in the pattern (except for sequence variables, which are handled separately). The resulting matches form a bipartite graph, where one set of nodes consists of the subject subterms and the other contains all the pattern subterms. Two nodes are connected by an edge iff the pattern matches the subject. Such an edge is also labeled with the match substitution(s). Finding an overall match is then accomplished by finding a maximum matching in this graph. However, for the matching to be valid, all the substitutions on its edges must be compatible, i.e. they cannot have contradicting replacements for the same variable. We use the Hopcroft-Karp algorithm [HK73] to find an initial maximum matching. However, since we are also interested in all matches and the initial matching might have incompatible substitutions, we use the algorithm described by Uno, Fukuda and Matsui [FM94], [Uno97] to enumerate all maximum matchings.

To avoid yielding redundant matches, we extended the bipartite graph by introducing a total order over its two node sets. This enables determining whether the edges of a matching maintain the order induced by the subjects or whether some of the edges "cross". Formally, for all edge pairs $(p,s),(p',s') \in M$ we require $(s \equiv s' \wedge p > p') \implies s > s'$ to hold where M is the matching, s,s' are subjects, and p,p' are patterns. An example of this is given in Figure 2. The order of the nodes is indicated by the numbers next to them. The only two maximum matchings for this particular match graph are displayed. In the left matching, the edges with the same subject cross and hence this matching is discarded. The other matching is used because it maintains the order. This ensures that only unique matches are yielded. Once a matching for the subpatterns is obtained, the remaining subject arguments are distributed to sequence variables in the same way as for one-to-one matching.

Experiments

To evaluate the performance of MatchPy, we conducted experiments on an Intel Core i5-2500K 3.3 GHz CPU with 8GB of RAM. Our focus is on relative performance of one-to-one and many-to-one matching rather than the absolute performance.

Linear Algebra

The operations for the linear algebra problem are shown in Table 1. The patterns all match BLAS kernels similar to the example pattern which was previously described. The pattern set consists of 199 such patterns. Out of those, 61 have an addition as outermost operation, 135 are patterns for products, and 3 are patterns for single matrices. A lot of these patterns only differ in terms of constraints, e.g. there are ten distinct patterns matching $A \times B$ with different constraints on the two matrices. By removing the sequence variables from the product patterns, these pattern can be made syntactic when ignoring the multiplication's associativity. In the following, we refer to the set of patterns with sequence variables as `LinAlg` and the set of syntactic product patterns as `Syntactic`.

The subjects were randomly generated such that matrices had random properties and each factor could randomly be transposed/inverted. The number of factors was chosen according to a normal distribution with $\mu = 5$. The total subject set consisted of 70 random products and 30 random sums. Out of the pattern set, random subsets were used to examine the influence of the pattern set size on the matching time. Across multiple subsets and repetitions per subject, the mean match and setup times were measured. Matching was performed both with the `match`

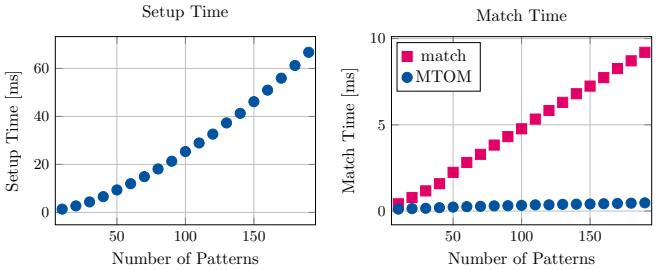


Fig. 3: *Timing Results for* `LinAlg`.

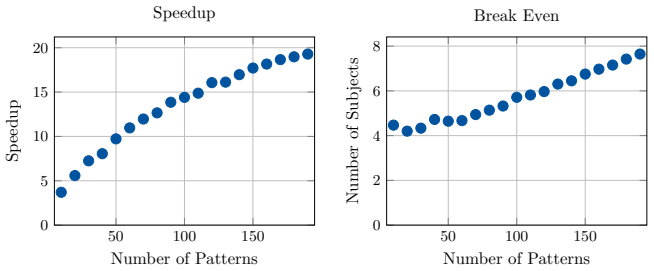


Fig. 4: *Comparison for* `LinAlg`.

function and the `ManyToOneMatcher` (MTOM). The results are displayed in Figure 3.

As expected, both setup and match times grow with the pattern set size. The growth of the many-to-one match time is much slower than the one for one-to-one matching. This is also expected since the simultaneous matching is more efficient. However, the growth of setup time for the many-to-one matcher beckons the question whether the speedup of the many-to-one matching is worth it.

Figure 4 depicts both the speedup and the break even point for many-to-one matching for `LinAlg`. The first graph indicates that the speedup of many-to-one matching increases with larger pattern sets. But in order to profit from that speedup, the setup cost of many-to-one matching must be amortized. Therefore, the second graph shows the break even point for many-to-one matching in terms of number of subjects. If for a given number of patterns and subjects the corresponding point is above the line, then many-to-one matching is overall faster. In this example, when matching more than eight times, many-to-one matching is overall always faster than one-to-one matching.

For the syntactic product patterns we compared the `match` function, the `ManyToOneMatcher` (MTOM) and the `DiscriminationNet` (DN). Again, randomly generated subjects were used. The resulting speedups and break even points are displayed in Figure 5.

In this case, the discrimination net is the fastest overall

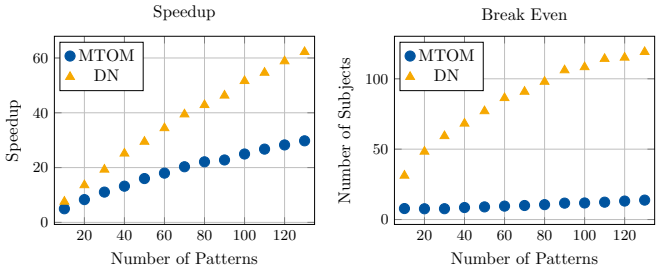


Fig. 5: *Comparison for* `Syntactic`.

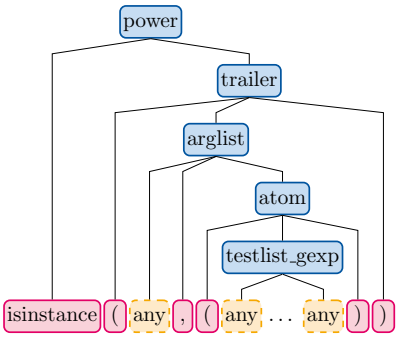


Fig. 6: *AST of the* `isinstance` *pattern.*

reaching a speedup of up to 60. However, because it also has the highest setup time, it only outperforms the many-to-one matcher after about 100 subjects for larger pattern set sizes. In practice, the discrimination net is likely the best choice for syntactic patterns, as long as the discrimination net does not grow to large. In the worst case, the size of the discrimination net can grow exponentially in the number of patterns.

Abstract Syntax Trees

Python includes a tool to convert code from Python 2 to Python 3. It is part of the standard library package `lib2to3` which has a collection of "fixers" that each convert one of the incompatible cases. To find matching parts of the code, those fixers use pattern matching on the abstract syntax tree (AST). Such an AST can be represented in the MatchPy data structures. We converted some of the patterns used by `lib2to3` both to demonstrate the generality of MatchPy and to evaluate the performance of many-to-one matching. Because the fixers are applied one after another and can modify the AST after each match, it would be difficult to use many-to-one matching for `lib2to3` in practice.

The following is an example of such a pattern:

```
power<
    'isinstance'
    trailer< '(' arglist< any ',' atom< '('
        args=testlist_gexp< any+ >
    ')' > > ')' >
>
```

It matches an `isinstance` expression with a tuple as second argument. Its tree structure is illustrated in Figure 6. The corresponding fixer cleans up duplications generated by previous fixers. For example `isinstance(x, (int, long))` would be converted by another fixer into `isinstance(x, (int, int))`, which in turn is then simplified to `isinstance(x, int)` by this fixer.

Out of the original 46 patterns, 36 could be converted to MatchPy patterns. Some patterns could not be converted, because they contain features that MatchPy does not support yet. The features include negated subpatterns (e.g. `not atom<'(' [any] ')'>`) and subpatterns that allow an aritrary number of repetitions (e.g. `any (',' any)+`).

Furthermore, some of the AST patterns contain alternative or optional subpatterns, e.g. `power<'input' args=trailer<'(' [any] ')'>>`. These features are also not directly supported by MatchPy, but they can be replicated by using multiple patterns. For those `lib2to3` patterns, all combinations of the alternatives were generated and added as invividual patterns. This resulted in about 1200 patterns for the many-to-one matcher that completely cover the original 36 patterns.

For the experiments, we used a file that combines the examples from the unittests of `lib2to3` with about 900 non-empty lines. We compared the set of 36 patterns with the original matcher and the 1200 patterns with the many-to-one matcher. A total of about 560 matches are found. Overall, on average, our many-to-one matcher takes 0.7 seconds to find all matches, while the matcher from `lib2to3` takes 1.8 seconds. This yields a speedup of approximately 2.5. However, the construction of the many-to-one matcher takes 1.4 seconds on average. However, this setup cost will be amortized by the faster matching for sufficiently large ASTs. The setup time can also mostly be eliminated by saving the many-to-one matcher to disk and loading it once required.

Compared to the one-to-one matching in MatchPy, the many-to-one matching achieves a speedup of about 60. This is due to the fact that for any given subject less than 1% of patterns match. By taking into account the setup time of the many-to-one matcher, the break even point for it is at about 200 subjects.

Conclusions

We have presented MatchPy, a pattern matching library for Python with support for sequence variables and associative/commutative functions. This library includes algorithms and data structures for both one-to-one and many-to-one matching. Because non-syntactic pattern matching is NP-hard, in the worst case the pattern matching times grows exponentially with the length of the pattern. Nonetheless, our experiments on real world examples indicate that many-to-one matching can give a significant speedup over one-to-one matching. However, the employed discrimination nets come with a one-time construction cost which needs to be amortized to benefit from their speedup. In our experiments, the break even point for many-to-one matching was always reached well within the typical number of subjects for the respective application. Therefore, many-to-one matching is likely to result in a compelling speedup in practice.

For syntactic patterns, we also compared the syntactic discrimination net with the many-to-one matcher. As expected, discrimination nets are faster at matching, but also have a significantly higher setup time. Furthermore, the number of states can grow exponentially with the number of patterns, making them unsuitable for some pattern sets. Overall, if applicable, discrimination nets offer better performance than a many-to-one matcher.

Which pattern matching algorithm is the fastest for a given application depends on many factors. Hence, it is not possible to give a general recommendation. Yet, the more subjects are matched against the same pattern set, the more likely it is that many-to-one outperforms one-to-one matching. In the experiments, a higher number of patterns lead to an increase of the speedup of many-to-one matching. In terms of the size of the many-to-one matcher, the growth of the net was sublinear in our experiments and still feasible for large pattern sets. The efficiency of using many-to-one matching also heavily depends on the actual pattern set, i.e. the degree of similarity and overlap between the patterns.

Future Work

We plan on extending MatchPy with more powerful pattern matching features to make it useful for an even wider range of applications. The greatest challenge with additional features is likely to implement them for many-to-one matching. In the following, we discuss some possibilities for extending the library.

Additional pattern features

In the future, we plan to implement similar functionality to the `Repeated`, `Sequence`, and `Alternatives` functions from Mathematica. These provide another level of expressive power which cannot be fully replicated with the current feature set of MatchPy. Another useful feature are context variables as described by Kutsia [Kut06]. They allow matching subterms at arbitrary depths which is especially useful for structures like XML. With context variables, MatchPy's pattern matching would be as powerful as XPath [RDS17] or CSS selectors [RJE17] for such structures. Similarly, function variables which can match a function symbol would also be useful for those applications.

Integration

Currently, in order to use MatchPy, existing data structures must be adapted to provide their children via an iterator. Where that is not possible, for example because the data structures are provided by a third party library, translation functions need to be applied. Also, some native data structures such as dicts are currently not supported directly. Therefore, it would be useful, to have a better way of using existing data structures with MatchPy.

In particular, easy integration with SymPy is an important goal, since it is a popular tool for working with symbolic mathematics. SymPy already implements a form of pattern matching which is less powerful than MatchPy. It lacks support for sequence variables, symbol wildcards and constraints. Each constant symbol in SymPy can have properties that allow it to be commutative or non-commutative. One benefit of this approach is easier modeling of linear algebra multiplication, where matrices and vectors do not commute, but scalars do. Better integration of MatchPy with SymPy would provide the users of SymPy with more powerful pattern matching tools. However, Matchpy would require selective commutativity to be fully compatible with SymPy. Also, SymPy supports older Python versions, while MatchPy requires Python 3.6.

Performance

If pattern matching is a major part of an application, its running time can significantly impact the overall speed. Reimplementing parts of MatchPy as a C module would likely result in a substantial speedup. Alternatively, adapting part of the code to Cython could be another option to increase the speed [BBS09], [WLØ09]. Furthermore, generating source code for a pattern set similar to parser generators for formal grammars could improve matching performance. While code generation for syntactic pattern matching has been the subject of various works [Aug85], [FM01], [Mar08], [MRV03], its application with the extended feature set of MatchPy is another potential area of future research. Also, additonal research on the viability of pattern matching with increasingly complex and large subjects or patterns is desirable. Parallelizing many-to-one matching is also a possibility to increase the overall speed which is worth exploring.

Functional pattern matching

Since Python does not have pattern matching as a language feature, MatchPy could be extended to provide a syntax similar to other functional programming languages. However, without a switch statement as part of the language, there is a limit to the syntax of this pattern expression. The following is an example of what such a syntax could look like:

```
with match(f(a, b)):
    if case(f(x_, y_)):
        print("x={}, y={}".format(x, y)))
    elif case(f(z_)):
        ....
```

There are already several libraries for Python which implement such a functionality for syntactic patterns and native data structures (e.g. MacroPy [Hao14], patterns [Sch14] or PyPatt [Jen15]). However, the usefulness of this feature needs further evaluation.

REFERENCES

[AHL00] Karen Aardal, Cor A. J. Hurkens, and Arjen K. Lenstra. Solving a system of linear diophantine equations with lower and upper bounds on the variables. *Mathematics of Operations Research*, 25(3):427–442, August 2000. doi:10.1287/moor.25.3.427.12219.

[Aug85] Lennart Augustsson. Compiling pattern matching. In *Proc. of a Conference on Functional Programming Languages and Computer Architecture*, pages 368–381, New York, NY, USA, 1985. Springer-Verlag New York, Inc.

[BBS09] Stefan Behnel, Robert W. Bradshaw, and Dag Sverre Seljebotn. Cython tutorial. In Gaël Varoquaux, Stéfan van der Walt, and Jarrod Millman, editors, *Proceedings of the 8th Python in Science Conference*, pages 4–14, Pasadena, CA USA, 2009.

[BCR93] Leo Bachmair, Ta Chen, and I. V. Ramakrishnan. *Associative-commutative Discrimination Nets*, pages 61–74. 1993. doi:10.1007/3-540-56610-4_56.

[BCR+95] Leo Bachmair, Ta Chen, I. V. Ramakrishnan, Siva Anantharaman, and Jacques Chabin. Experiments with associative-commutative discrimination nets. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence - Volume 1*, IJCAI'95, pages 348–354. Morgan Kaufmann, 1995.

[BKN87] Dan Benanav, Deepak Kapur, and Paliath Narendran. Complexity of matching problems. *Journal of Symbolic Computation*, 3(1):203–216, February 1987. doi:10.1016/S0747-7171(87)80027-5.

[BN98] Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, New York, NY, USA, 1998.

[Bon67] James Bond. Calculating the general solution of a linear diophantine equation. *The American Mathematical Monthly*, 74(8):955, October 1967. doi:10.2307/2315274.

[BW84] Alan Bundy and Lincoln Wallen. *Discrimination Net*, pages 31–31. 1984. doi:10.1007/978-3-642-96868-6_59.

[CF89] Michael Clausen and Albrecht Fortenbacher. Efficient solution of linear diophantine equations. *Journal of Symbolic Computation*, 8(1-2):201–216, July 1989. doi:10.1016/s0747-7171(89)80025-2.

[Chr93] Jim Christian. Flatterms, discrimination nets, and fast term rewriting. *Journal of Automated Reasoning*, 10(1):95–113, 1993. doi:10.1007/BF00881866.

[DCHD90] J. J. Dongarra, Jeremy Du Croz, Sven Hammarling, and I. S. Duff. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 16(1):1–17, March 1990. doi:10.1145/77626.79170.

[DCHH88] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Richard J. Hanson. An extended set of FORTRAN basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 14(1):1–17, March 1988. doi:10.1145/42288.42291.

[Eke95] S. M. Eker. Associative-commutative matching via bipartite graph matching. *The Computer Journal*, 38(5):381–399, May 1995. doi:10.1093/comjnl/38.5.381.

[FM94] K. Fukuda and T. Matsui. Finding all the perfect matchings in bipartite graphs. *Applied Mathematics Letters*, 7(1):15–18, January 1994. doi:10.1016/0893-9659(94)90045-0.

[FM01] Fabrice Le Fessant and Luc Maranget. Optimizing pattern matching. In *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming - ICFP '01*. Association for Computing Machinery (ACM), 2001. doi:10.1145/507635.507641.

[Grä91] Albert Gräf. Left-to-right tree pattern matching. In *Proceedings of the 4th International Conference on Rewriting Techniques and Applications*, RTA-91, pages 323–334, New York, NY, USA, 1991. Springer-Verlag New York, Inc.

[Hao14] Li Haoyi. Macropy. https://github.com/lihaoyi/macropy, 2014. URL: https://github.com/lihaoyi/macropy.

[HK73] John E. Hopcroft and Richard M. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM Journal on Computing*, 2(4):225–231, December 1973. doi:10.1137/0202019.

[Jen15] Grant Jenks. Pypatt. https://pypi.python.org/pypi/pypatt, 2015. URL: https://pypi.python.org/pypi/pypatt.

[KL91] E. Kounalis and D. Lugiez. Compilation of pattern matching with associative-commutative functions. In *TAPSOFT '91*, pages 57–73. Springer Nature, 1991. doi:10.1007/3-540-53982-4_4.

[KM01] Hélène Kirchner and Pierre-Etienne Moreau. Promoting rewriting to a programming language: A compiler for non-deterministic rewrite programs in associative-commutative theories. *International Journal of Foundations of Computer Science*, 11(2):207–251, March 2001. doi:10.1142/S0129054101000412.

[Kre17a] Manuel Krebber. Matchpy documentation. http://matchpy.readthedocs.io/, 06 2017. URL: http://matchpy.readthedocs.io/.

[Kre17b] Manuel Krebber. Non-linear associative-commutative many-to-one pattern matching with sequence variables. Master's thesis, 2017. arXiv:1705.00907.

[Kut06] Temur Kutsia. Context sequence matching for XML. *Electronic Notes in Theoretical Computer Science*, 157(2):47–65, May 2006. doi:10.1016/j.entcs.2005.12.045.

[Lam88] J. L. Lambert. Finding a partial solution to a linear system of equations in positive integers. *Computers & Mathematics with Applications*, 15(3):209–212, 1988. doi:10.1016/0898-1221(88)90172-1.

[LHKK79] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for fortran usage. *ACM Transactions on Mathematical Software*, 5(3):308–323, September 1979. doi:10.1145/355841.355847.

[LM94] D. Lugiez and J. L. Moysset. Tree automata help one to solve equational formulae in AC-theories. *Journal of Symbolic Computation*, 18(4):297–318, October 1994. doi:10.1006/jsco.1994.1049.

[Mar08] Luc Maranget. Compiling pattern matching to good decision trees. In *Proceedings of the 2008 ACM SIGPLAN Workshop on ML - ML '08*. Association for Computing Machinery (ACM), 2008. doi:10.1145/1411304.1411311.

[MRV03] Pierre-Etienne Moreau, Christophe Ringeissen, and Marian Vittek. *A Pattern Matching Compiler for Multiple Target Languages*, pages 61–76. 2003. doi:10.1007/3-540-36579-6_5.

[MSP+17] Aaron Meurer, Christopher P. Smith, Mateusz Paprocki, Ondřej Čertík, Sergey B. Kirpichev, Matthew Rocklin, AMiT Kumar, Sergiu Ivanov, Jason K. Moore, Sartaj Singh, Thilina Rathnayake, Sean Vig, Brian E. Granger, Richard P. Muller, Francesco Bonazzi, Harsh Gupta, Shivam Vats, Fredrik Johansson, Fabian Pedregosa, Matthew J. Curry, Andy R. Terrel, Štěpán Roučka, Ashutosh Saboo, Isuru Fernando, Sumith Kulal, Robert Cimrman, and Anthony Scopatz. Sympy: Symbolic computing in python. *Peerj Computer Science*, 3:e103, January 2017. doi:10.7717/peerj-cs.103.

[MVO96] Alfred J. Menezes, Scott A. Vanstone, and Paul C. Van Oorschot. *Handbook of Applied Cryptography*. CRC Press, Inc., Boca Raton, FL, USA, 1st edition, 1996.

[NWE97] Nadia Nedjah, Colin D. Walter, and Stephen E. Eldridge. *Optimal Left-to-right Pattern-matching Automata*, pages 273–286. 1997. doi:10.1007/BFb0027016.

[Pö16] Jan Pöschko. Mathics. http://mathics.github.io/, October 2016. URL: http://mathics.github.io/.

[RDS17] Jonathan Robie, Michael Dyck, and Josh Spiegel. XML path language (XPath) 3.1. W3C recommendation, W3C, March 2017.

[RJE17] Florian Rivoal, Tab Atkins Jr., and Elika Etemad. CSS snapshot 2017. W3C note, W3C, January 2017.

[Sch14] Alexander Schepanovski. patterns. https://github.com/Suor/patterns, 2014. URL: https://github.com/Suor/patterns.

[Uno97] Takeaki Uno. Algorithms for enumerating all perfect, maximum and maximal matchings in bipartite graphs. In *Algorithms and Computation*, pages 92–101. Springer Nature, 1997. doi:10.1007/3-540-63890-3_11.

[Wei60] Robert Weinstock. Greatest common divisor of several integers and an associated linear diophantine equation. *The Amer-*

ican Mathematical Monthly, 67(7):664, August 1960. doi:10.2307/2310105.

[WLØ09] Ilmar M. Wilbers, Hans Petter Langtangen, and Åsmund Ødegård. Using cython to speed up numerical python programs. In *Proceedings of Mekit'09*, pages 495–512, 2009.

[WR16] Inc. Wolfram Research. Mathematica, 2016. URL: https://www.wolfram.com.

pulse2percept: A Python-based simulation framework for bionic vision

Michael Beyeler[‡§¶*], Geoffrey M. Boynton[‡], Ione Fine[‡], Ariel Rokem[¶§]

https://youtu.be/KxsNAa-P2X4

Abstract—By 2020 roughly 200 million people worldwide will suffer from photoreceptor diseases such as retinitis pigmentosa and age-related macular degeneration, and a variety of retinal sight restoration technologies are being developed to target these diseases. One technology, analogous to cochlear implants, uses a grid of electrodes to stimulate remaining retinal cells. Two brands of retinal prostheses are currently approved for implantation in patients with late stage photoreceptor disease. Clinical experience with these implants has made it apparent that the vision restored by these devices differs substantially from normal sight. To better understand the outcomes of this technology, we developed *pulse2percept*, an open-source Python implementation of a computational model that predicts the perceptual experience of retinal prosthesis patients across a wide range of implant configurations. A modular and extensible user interface exposes the different building blocks of the software, making it easy for users to simulate novel implants, stimuli, and retinal models. We hope that this library will contribute substantially to the field of medicine by providing a tool to accelerate the development of visual prostheses.

Index Terms—bionic vision, retinal implant, pulse2percept, prosthesis

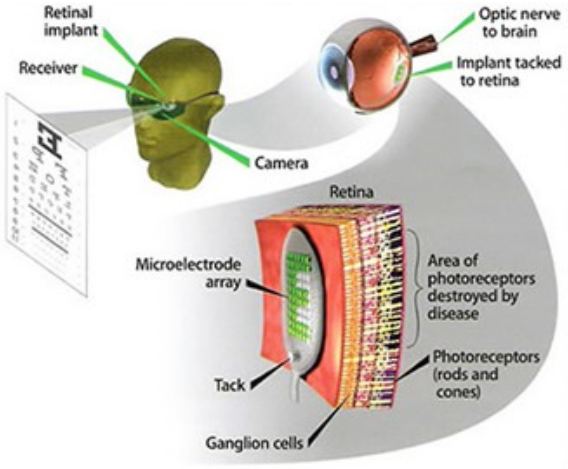


Fig. 1: *Electronic retinal prosthesis. Light from the visual scene is captured by an external camera and transformed into electrical pulses delivered through microelectrodes to stimulate the retina.*

Introduction

Two frequent causes of blindness in the developed world are age-related macular degeneration (AMD) and retinitis pigmentosa (RP) [BBB+84], [Gro04]. Both of these diseases have a hereditary component, and are characterized by a progressive degeneration of photoreceptors in the retina that lead to gradual loss of vision.

Microelectronic retinal prostheses have been developed in an effort to restore sight to RP and AMD patients. Analogous to cochlear implants, these devices function by electrically stimulating surviving retinal neurons in order to evoke neuronal responses that are transmitted to the brain and interpreted by patients as visual percepts (Fig. 1). Two of these devices are already approved for commercial use, and a number of other companies have either started or are planning to start clinical trials of devices in the near future. Other types of technologies, such as optogenetics and genetic modification are also areas of active research. Blinded individuals may potentially be offered a wide range of sight restoration options within a decade [FCL15].

One major challenge in the development of retinal prostheses is predicting what patients will see when they use their devices. Interactions between implant electronics and the underlying neurophysiology cause nontrivial perceptual distortions in both space and time [FB15], [BRBFss] that severely limit the quality of the generated visual experience.

Our goal was to develop a simulation framework that could describe the visual percepts of retinal prosthesis patients over space and time. We refer to these simulations as 'virtual patients', analogous to the virtual prototyping that has proved so useful in other complex engineering applications.

Here we present an open-source implementation of these models as part of *pulse2percept*, a BSD-licensed Python-based simulation framework [BSD17] that relies on the NumPy and SciPy stacks, as well as contributions from the broader Python community. Based on the detailed specification of a patient's implant configuration, and given a desired electrical stimulus, the model predicts the perceptual distortions experienced by 'virtual patients' over both space and time.

We hope that this library will contribute substantially to the field of medicine by providing a tool to accelerate the development of visual prostheses. Researchers may use this tool to improve stimulation protocols of existing implants or to aid development of future devices. In addition, this tool might guide government agen-

* *Corresponding author: mbeyeler@uw.edu*
‡ *Department of Psychology, University of Washington*
§ *Institute for Neuroengineering, University of Washington*
¶ *eScience Institute, University of Washington*

cies, such as the FDA and Medicare, in making reimbursement decisions. Furthermore, this tool can be used to guide patients and doctors in their decision as to when or whether to be implanted, and which device to select.

The remainder of this paper is organized as follows: We start by introducing the neuroscience background necessary to understand the interactions between implant electronics and the underlying neurophysiology. We then detail the computational model that underlies *pulse2percept*, before we give a simple usage example and go into implementation details. We then review our solutions to various technical challenges, and conclude by discussing the broader impact for this work for computational neuroscience and neural engineering communities in more detail.

Background

The first steps in seeing begin in the retina, where a mosaic of photoreceptors converts incoming light into electrochemical signals that encode the intensity of light as a function of position (two dimensions), wavelength, and time [Rod98]. The electrochemical signal is passed on to specialized neuronal circuits consisting of a variety of cell types (such as bipolar, amacrine, and horizontal cells), which act as feature detectors for basic sensory properties, such as spatial contrast and temporal frequency. These sensory features are then encoded in parallel across approximately 1.5 million retinal ganglion cells, which form the output layer of the retina. Each ganglion cell relays the electrical signal to the brain via a long axon fiber that passes from the ganglion cell body to the optic nerve and on to the brain.

Diseases such as RP and AMD are characterized by a progressive degeneration of photoreceptors, gradually affecting other layers of the retina [HPdJ+99], [MNS08]. In severe end-stage RP, roughly 95% of photoreceptors, 20% of bipolar cells, and 70% of ganglion cells degenerate [SHd+97]. In addition, the remaining cells undergo corruptive re-modeling in late stages of the disease [MJWS03], [MJ03], so that little or no useful vision is retained.

Microelectronic retinal prostheses have been developed in an effort to restore sight to individuals suffering from RP or AMD. Analogous to cochlear implants, the goal of retinal prostheses is to electrically stimulate surviving bipolar or ganglion cells in order to evoke neuronal responses that are interpreted by the brain as visual percepts. The electrical stimulus delivers charge to the cell membrane that depolarizes the neuron and opens voltage-sensitive ion channels. This bypasses the natural presynaptic neurotransmitter excitation and causes the activated neurons to stimulate their postsynaptic targets. Therefore, selective spatiotemporal modulation of retinal neurons with an array of electrodes should allow a prosthesis to coordinate retinal activity in place of natural photoreceptor input [WWH16].

Several types of retinal prostheses are currently in development. These vary in their user interface, light-detection method, signal processing, and microelectrode placement within the retina (for a recent review see [WWH16]). As far as our model is concerned, the critical factor is the placement of the microelectrodes on the retina (Fig. 2). The three main locations for microelectrode implant placement are *epiretinal* (i.e., on top of the retinal surface, above the ganglion cells), *subretinal* (i.e., next to the bipolar cells in the space of the missing photoreceptors), and *suprachoroidal* (i.e., between the choroid and the sclera). Each of these approaches is similar in that light from the visual scene is captured via a camera and transformed into electrical pulses delivered through electrodes to stimulate the retina.

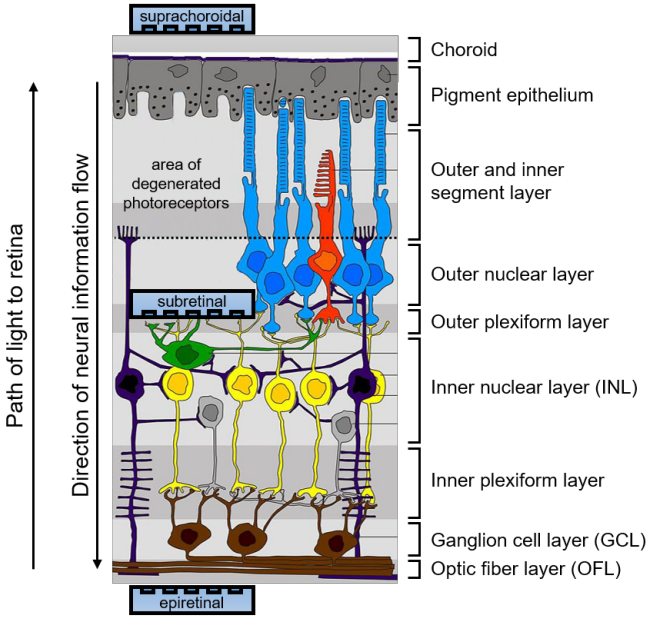


Fig. 2: *Diagram of the retina and common locations of retinal prosthesis microelectrode arrays. Retinitis pigmentosa causes severe photoreceptor degeneration. Epiretinal electrode arrays are placed in the vitreous, next to the optic fiber layer (OFL). Subretinal arrays are placed by creating a space between the choroid and remaining retinal tissue. Suprachoroidal arrays are placed behind the choroid.* pulse2percept *allows for simulation of processing in the inner nuclear layer (INL), ganglion cell layer (GCL), and optic fiber layer (OFL). Based on "Retina layers" by Peter Hartmann, CC BY-SA 3.0.*

As mentioned above, two devices are currently approved for commercial use and are being implanted in patients across the US and Europe: the Argus II device (epiretinal, Second Sight Medical Products Inc., [dCDH+16]) and the Alpha-IMS system (subretinal, Retina Implant AG, [SBSB+15]). At the same time, a number of other companies have either started or are planning to start clinical trials in the near future, potentially offering a wide range of sight restoration options for the blind within a decade [FCL15].

However, clinical experience with existing retinal prostheses makes it apparent that the vision provided by these devices differs substantially from normal sight. Interactions between implant electronics and the underlying neurophysiology cause nontrivial perceptual distortions in both space and time [FB15], [BRBFss] that severely limit the quality of the generated visual experience. For example, stimulating a single electrode rarely produces the experience of a 'dot' of light, instead leading to percepts that vary drastically in shape, varying in description from 'blobs', to 'streaks' and 'half-moons'. The percept produced by stimulating a single electrode with a continuous pulse train also fades over time, usually disappearing over a course of seconds. As a result, patients do not report seeing an interpretable world. One patient describes it as like: *"... looking at the night sky where you have millions of twinkly lights that almost look like chaos"* [Pre15].

Previous work by our group has focused on development of computational models to describe some of these distortions for a small number of behavioral observations in either space [NFH+12] or time [HGW+09]. Here we present a model that can describe spatial distortions, temporal nonlinearities, and spatiotemporal interactions reported across a wide range of conditions, devices, and patients.

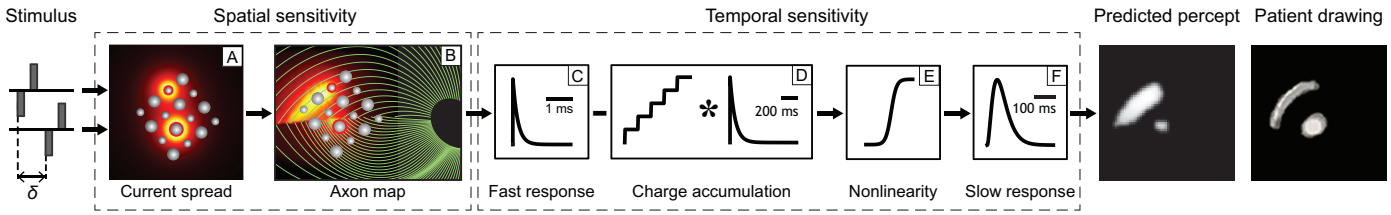


Fig. 3: *Full model cascade. A simulated electrical stimulus is processed by a series of linear filtering and nonlinear processing steps that model the spatial (A, B) and temporal sensitivity (C-F) of the retinal tissue. An Argus I device is shown (16 electrodes of 260 or 520 microns diameter arranged in a checkerboard pattern). The output of the model is a prediction of the visual percept in both space and time (example frame shown), which can be compared to human patients' drawings.*

Computational Model of Bionic Vision

Analogous to models of cochlear implants [Sha89], the goal of our computational model is to approximate, via a number of linear filtering and nonlinear processing steps, the neural computations that convert electrical pulse trains across multiple electrodes into a perceptual experience in space and time.

Model parameters were chosen to fit data from a variety of behavioral experiments in patients with prosthetic devices. For example, in threshold experiments patients were asked to report whether or not they detected a percept. Across many trials, the minimum stimulation current amplitude needed to reliably detect the presence of a percept on 80% of trials was found. This threshold was measured across a variety of pulse trains that varied across dimensions such as frequency, duty cycle, and duration. In other experiments patients reported the apparent brightness or size of percepts on a rating scale. In others patients drew the shapes of the percepts evoked by stimulation. The model has been shown to generalize across individual electrodes, patients, and devices, as well as across different experiments. Detailed methods of how the model was validated can be found in [HGW+09], [NFH+12], [BRBFss]. Here we provide a brief overview of the model cascade.

The full model cascade for an Argus I epiretinal prosthesis is illustrated in Fig. 3. The Argus I device simulated here consists of electrodes of 260 μm and 520 μm diameter, arranged in a checkerboard pattern (Fig. 3 A). In this example, input to the model is a pair of simulated pulse trains phase-shifted by δ ms, which are delivered to two individual simulated electrodes.

The first stages of the model describe the spatial distortions resulting from interactions between the electronics and the neuroanatomy of the retina. We assume that the current density caused by electrical stimulation decreases as a function of distance from the edge of the electrode [ABK+08]:

$$c(d) = \frac{\alpha}{\alpha + d^n} \tag{1}$$

where d is the 3D Euclidean distance to the electrode edge, $\alpha = 14000$ and the exponent $n = 1.69$. Current fields for two stimulated electrodes are shown, Fig. 3 A (the hotter the color, the higher the current).

Due to the fact that epiretinal implants sit on top of the optic fiber layer (Fig. 2), they do not only stimulate ganglion cell bodies but also ganglion cell axons. It is critical to note that, perceptually, activating an axon fiber that passes under a stimulated electrode is indistinguishable from the percept that would be elicited by activating the corresponding ganglion cell *body*. The produced visual percept will appear in the spatial location in visual space for which the corresponding ganglion cell and axon usually encodes information. Ganglion cells send their axons on highly stereotyped pathways to the optic disc (green lines in Fig. 3 B), which have been mathematically described [JNS+09]. As a result, electrical stimulation of axon fibers leads to predictable visual 'streaks' or 'comet trails' that are aligned with the axonal pathways.

We therefore convert the spatial map of current densities into a tissue activation map by accounting for axonal stimulation. We model the sensitivity of axon fibers as decreasing exponentially as a function of distance from the corresponding ganglion cell bodies. The resulting tissue activation map across the retinal surface is shown as a heatmap in Fig. 3 B (the hotter the color, the larger the amount of tissue stimulation).

The remaining stages of the model describe temporal nonlinearities. Every pixel of the tissue activation map is modulated over time by the applied electrical pulse train in order to predict a perceived brightness value that varies over time. This involves applying a series of linear filtering (Fig. 3 C, D, and F) and nonlinear processing (Fig. 3 E) steps in the time domain that are designed to approximate the processing of visual information within the retina and visual cortex.

Linear responses in Fig. 3 C, D, and F are modeled as temporal low-pass filters, or 'leaky integrators', using gamma functions of order n:

$$\delta(t, n, \tau) = \frac{\exp(-t/\tau)}{\tau(n-1)!}\left(\frac{t}{\tau}\right)^{n-1} \tag{2}$$

where t is time, n is the number of identical, cascading stages, and τ is the time constant of the filter.

The first temporal processing step convolves the timeseries of tissue activation strengths $f(t)$ at a particular spatial location with a one-stage gamma function ($n = 1$, time constant $\tau_1 = 0.42$ ms) to model the impulse response function of retinal ganglion cells (Fig. 3 C):

$$r_1(t) = f(t) * \delta(t, 1, \tau_1), \tag{3}$$

where $*$ denotes convolution.

Behavioral data suggests that the system becomes less sensitive as a function of accumulated charge. This is implemented by calculating the amount of accumulating charge at each point of time in the stimulus, $c(t)$, and by convolving this accumulation with a second one-stage gamma function ($n = 1$, time constant $\tau_2 = 45.3$ ms; Fig. 3 D). The output of this convolution is scaled by a factor $\varepsilon_1 = 8.3$ and subtracted from r_1 (Eq. 3):

$$r_2(t) = r_1(t) - \varepsilon_1\big(c(t) * \delta(t, 1, \tau_2)\big). \tag{4}$$

The response $r_2(t)$ is then passed through a stationary nonlinearity (Fig. 3 E) to model the nonlinear input-output relationship of ganglion cell spike generation:

$$r_3(t) = r_2(t)\frac{\alpha}{1 + \exp\frac{\delta - \max_t r_2(t)}{s}} \tag{5}$$

where $\alpha = 14$ (asymptote), $s = 3$ (slope), and $\delta = 16$ (shift) are chosen to match the observed psychophysical data.

Finally, the response $r_3(t)$ is convolved with another low-pass filter described as a three-stage gamma function ($n = 3$, $\tau_3 = 26.3$ ms) intended to model slower perceptual processes in the brain (Fig. 3 F):

$$r_4(t) = \varepsilon_2 r_3(t) * \delta(t, 3, \tau_3), \quad (6)$$

where $\varepsilon_2 = 1000$ is a scaling factor used to scale the output to subjective brightness values in the range [0, 100].

The output of the model is thus a movie of brightness values that corresponds to the predicted perceptual experience of the patient. An example percept generated is shown on the right-hand side of Fig. 3 ('predicted percept', brightest frame in the movie).

Implementation and Results

Code Organization

The *pulse2percept* project seeks a trade-off between optimizing for computational performance and ease of use. To facilitate ease of use, we organized the software as a standard Python package, consisting of the following primary modules:

- `api`: a top-level Application Programming Interface.
- `implants`: implementations of the details of different retinal prosthetic implants. This includes Second Sight's Argus I and Argus II implants, but can easily be extended to feature custom implants (see Section Extensibility).
- `retina`: implementation of a model of the retinal distribution of nerve fibers, based on [JNS+09], and an implementation of the temporal cascade of events described in Eqs. 2-6. Again, this can easily be extended to custom temporal models (see Section Extensibility).
- `stimuli`: implementations of commonly used electrical stimulation protocols, including methods for translating images and movies into simulated electrical pulse trains. Again, this can easily be extended to custom stimulation protocols (see Section Extensibility).
- `files`: a means to load/store data as images and videos.
- `utils`: various utility and helper functions.

Basic Usage

Here we give a minimal usage example to produce the percept shown on the right-hand side of Fig. 3.

Convention is to import the main `pulse2percept` module as `p2p`. Throughout this paper, if a class is referred to with the prefix `p2p`, it means this class belongs to the main pulse2percept library (e.g., `p2p.retina`):

```
import pulse2percept as p2p
```

`p2p.implants`: Our goal was to create electrode implant objects that could be configured in a highly flexible manner. As far as placement is concerned, an implant can be placed at a particular location on the retina (`x_center`, `y_center`) with respect to the fovea (in microns), and rotated as you see fit (`rot`):

```
import numpy as np
implant = p2p.implants.ArgusI(x_center=-800,
                               y_center=0,
                               h=80,
                               rot=np.deg2rad(35))
```

Here, we make use of the `ArgusI` array type, which provides predefined values for array type ('epiretinal') and electrode diameters. In addition, the distance between the array and the retinal tissue can be specified via the height parameter (`h`), either on a per-electrode basis (as a list) or using the same value for all electrodes (as a scalar).

The electrodes within the implant can also be specified. An implant is a wrapper around a list of `p2p.implants.Electrode` objects, which are accessible via indexing or iteration (e.g., via `[for i in implant]`). This allows for specification of electrode diameter, position, and distance to the retinal tissue on a per-electrode basis. Once configured, every `Electrode` object in the implant can also be assigned a name (in the Argus I implant, they are A1 - A16; corresponding to the names that are commonly used by Second Sight Medical Products Inc.). The first electrode in the implant can be accessed both via its index (`implant[0]`) and its name (`implant['A1']`).

Once the implant is created, it can be passed to the simulation framework. This is also where you specify the back end (currently supported are 'serial', 'joblib' [Job16], and 'dask' [Das16]):

```
sim = p2p.Simulation(implant, engine='joblib',
                     num_jobs=8)
```

The simulation framework provides a number of setter functions for the different layers of the retina. These allow for flexible specification of optional settings, while abstracting the underlying functionality.

`p2p.retina`: This includes the implementation of a model of the retinal distribution of nerve fibers, based on [JNS+09], and implementations of the temporal cascade of events described in Eqs. 2-6.

Things that can be set include the spatial sampling rate of the optic fiber layer (`ssample`) as well as the spatial extent of the retinal patch to be simulated (given by the corner points `[xlo, ylo]` and `[xhi, yhi]`). If the coordinates of the latter are not given, a patch large enough to fit the specified electrode array will be automatically selected:

```
ssample = 100  # microns
num_axon_samples = 801
sim.set_optic_fiber_layer(ssample=ssample,
                          num_axon_samples=801)
```

Similarly, for the ganglion cell layer we can choose one of the pre-existing cascade models and specify a temporal sampling rate:

```
tsample = 0.005 / 1000  # seconds
sim.set_ganglion_cell_layer('Nanduri2012',
                            tsample=tsample)
```

As its name suggest, `'Nanduri2012'` implements the model detailed in [NFH+12]. Other options include `'Horsager2009'` [HGW+09] and `'latest'`.

It's also possible to specify your own (custom) model, see Section Extensibility below.

`p2p.stimuli`: A stimulation protocol can be specified by assigning stimuli from the `p2p.stimuli` module to specific electrodes. An example is to set up a pulse train of particular stimulation frequency, current amplitude and duration. Because of safety considerations, all real-world stimuli must be balanced biphasic pulse trains (i.e., they must have a positive and negative phase of equal area, so that the net current delivered to the tissue sums to zero).

It is possible to specify a pulse train for each electrode in the implant as follows:

```
# Stimulate two electrodes, others set to zero
stim = []
for elec in implant:
```

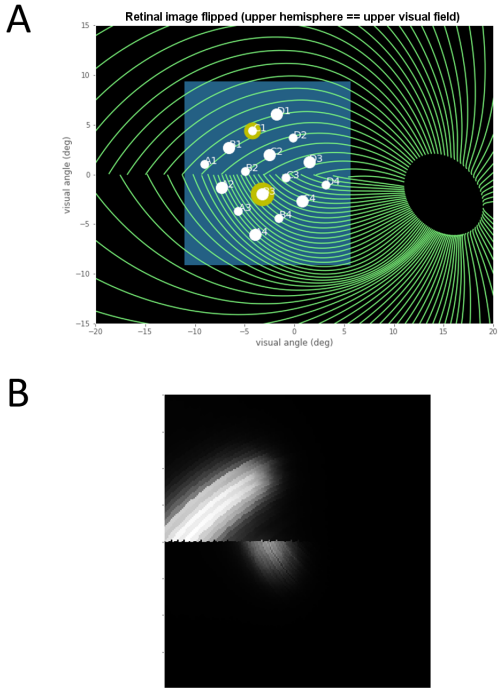


Fig. 4: *Model input/output generated by the example code. (A) An epiretinal Argus I array is placed near the fovea, and two electrodes ('C1' and 'B3') are stimulated with 50 Hz, 20 uA pulse trains. The plot is created by lines 34-36. Note that the retinal image is flipped, so that the upper hemisphere corresponds to the upper visual field. (B) Predicted visual percept (example frame shown). The plot is created by line 41.*

```
19     if elec.name == 'C1' or elec.name == 'B3':
20         # 50 Hz, 20 uA, 0.5 sec duration
21         pt = p2p.stimuli.PulseTrain(tsample,
22                                     freq=50.0,
23                                     amp=20.0,
24                                     dur=0.5)
25     else:
26         pt = p2p.stimuli.PulseTrain(tsample, freq=0)
27     stim.append(pt)
```

However, since implants are likely to have electrodes numbering in the hundreds or thousands, this method will rapidly become cumbersome when assigning pulse trains across multiple electrodes. Therefore, an alternative is to assign pulse trains to electrodes via a dictionary:

```
28 stim = {
29     'C1': p2p.stimuli.PulseTrain(tsample, freq=50.0,
30                                  amp=20.0, dur=0.5)
31     'B3': p2p.stimuli.PulseTrain(tsample, freq=50.0,
32                                  amp=20.0, dur=0.5)
33 }
```

At this point, we can highlight the stimulated electrodes in the array:

```
34 import matplotlib.pyplot as plt
35 %matplotlib inline
36 sim.plot_fundus(stim)
```

The output can be seen in Fig. 4 A.

Finally, the created stimulus serves as input to `sim.pulse2percept`, which is used to convert the pulse trains into a percept. This allows users to simulate the predicted percepts for simple input stimuli, such as stimulating a pair of electrodes, or more complex stimuli, such as stimulating a grid of electrodes in the shape of the letter A.

At this stage in the model it is possible to consider which retinal layers are included in the temporal model, by selecting from the following list (see Fig. 2 for a schematic of the anatomy):

- `'OFL'`: optic fiber layer (OFL), where ganglion cell axons reside,
- `'GCL'`: ganglion cell layer (GCL), where ganglion cell bodies reside, and
- `'INL'`: inner nuclear layer (INL), where bipolar cells reside.

A list of retinal layers to be included in the simulation is then passed to the `pulse2percept` method:

```
37 # From pulse train to percept
38 percept = sim.pulse2percept(stim, tol=0.25,
39                              layers=['GCL', 'OFL'])
```

This allows the user to run simulations that include only the layers relevant to a particular simulation. For example, axonal stimulation and the resulting axon streaks can be ignored by omitting `'OFL'` from the list. By default, all three supported layers are included.

Here, the output `percept` is a `p2p.utils.TimeSeries` object that contains the time series data in its `data` container. This time series consists of brightness values (arbitrary units in [0, 100]) for every pixel in the percept image.

Alternatively, it is possible to retrieve the brightest (mean over all pixels) frame of the time series:

```
40 frame = p2p.get_brightest_frame(percept)
```

Then we can plot it with the help of Matplotlib (Fig. 4 B):

```
41 plt.imshow(frame, cmap='gray')
```

`p2p.files`: *pulse2percept* offers a collection of functions to convert the `p2p.utils.TimeSeries` output into a movie file via scikit-video [Sci17] and ffmpeg [FFm10].

For example, a percept can be stored to an MP4 file as follows:

```
42 # Save movie at 15 frames per second
43 p2p.files.save_video(percept, 'mypercept.mp4',
44                       fps=15)
```

For convenience, *pulse2percept* provides a function to load a video file and convert it to the `p2p.utils.TimeSeries` format:

```
45 # Load video as TimeSeries of size (M x N x T),
46 # M: height, N: width, T: number of frames
47 video = p2p.files.load_video('mypercept.mp4')
```

Analogously, *pulse2percept* also provides functionality to go directly from images or videos to electrical stimulation on an electrode array:

```
48 from_img = p2p.stimuli.image2pulsetrain('myimg.jpg',
49                                          implant)
50 from_vid = p2p.stimuli.video2pulsetrain('mymov.avi',
51                                          implant)
```

These functions are based on functionality provided by scikit-image [S v14] and scikit-video [Sci17], respectively, and come with a number of options to specify whether brightness should be encoded as pulse train amplitude or frequency, at what frame rate to sample the movie, whether to maximize or invert the contrast, and so on.

Extensibility

As described above, our software is designed to allow for implants, retinal models, and pulse trains to be customized. We provide extensibility mainly through mechanisms of class inheritance.

Retinal Implants: Creating a new implant involves inheriting from `p2p.implants.ElectrodeArray` and providing a property `etype`, which is the electrode type (e.g., `'epiretinal'`, `'subretinal'`):

```
52 import pulse2percept as p2p
53
54 class MyImplant(p2p.implants.ElectrodeArray):
55
56     def __init__(self, etype):
57         """Custom electrode array
58
59         Parameters
60         ----------
61         etype : str
62             Electrode type, {'epiretinal',
63             'subretinal'}
64         """
65         self.etype = etype
66         self.num_electrodes = 0
67         self.electrodes = []
```

Then new electrodes can be added by utilizing the `add_electrodes` method of the base class:

```
68 myimplant = MyImplant('epiretinal')
69 r = 150  # electrode radius (um)
70 x, y = 10, 20  # distance from fovea (um)
71 h = 50  # height from retinal surface (um)
72 myimplant.add_electrodes(r, x, y, h)
```

Retinal cell models: Any new ganglion cell model is described as a series of temporal operations that are carried out on a single pixel of the image. It must provide a property called `tsample`, which is the temporal sampling rate, and a method called `model_cascade`, which translates a single-pixel pulse train into a single-pixel percept over time:

```
73 class MyGanglionCellModel(p2p.retina.TemporalModel):
74     def model_cascade(self, in_arr, pt_list, layers,
75                       use_jit):
76         """Custom ganglion cell model
77
78         Parameters
79         ----------
80         in_arr : array_like
81             2D array <#layers x #time points> of
82             effective current values at a single
83             pixel location.
84         pt_list : list
85             List of pulse train `data` containers.
86         layers : list
87             List of retinal layers to simulate.
88             Choose from:
89             - 'OFL': optic fiber layer
90             - 'GCL': ganglion cell layer
91             - 'INL': inner nuclear layer
92         use_jit : bool
93             If True, applies just-in-time (JIT)
94             compilation to expensive computations
95             for additional speedup (requires Numba)
96         """
97         return p2p.utils.TimeSeries(self.tsample,
98                                     in_arr[0, :])
```

This method can then be passed to the simulation framework:

```
99 mymodel = MyGanglionCellModel()
100 sim.set_ganglion_cell_layer(mymodel)
```

It will then automatically be selected as the implemented ganglion cell model when `sim.pulse2percept` is called.

Stimuli: The smallest stimulus building block provided by *pulse2percept* consists of a single pulse of either positive current (anodic) or negative current (cathodic), which can be created via `p2p.stimuli.MonophasicPulse`. However, as described above, any real-world stimulus must consist of biphasic pulses with zero net current. A single biphasic pulse can be created via `p2p.stimuli.BiphasicPulse`. A train of such pulses can be created via `p2p.stimuli.PulseTrain`. This setup gives the user the opportunity to build their own stimuli by creating pulse trains of varying amplitude, frequency, and interpulse intervals.

In order to define new pulse shapes and custom stimuli, the user can either inherit from any of these stimuli classes or directly from `p2p.utils.TimeSeries`. For example, a biphasic pulse can be built from two monophasic pulses as follows:

```
101 from pulse2percept.stimuli import MonophasicPulse
102
103 class MyBiphasicPulse(p2p.utils.TimeSeries):
104
105     def __init__(self, pdur, tsample):
106         """A charge-balanced pulse with a cathodic
107            and anodic phase
108
109         Parameters
110         ----------
111         tsample : float
112             Sampling time step in seconds.
113         pdur : float
114             Single-pulse phase duration (seconds).
115         """
116         on = MonophasicPulse('anodic', pdur, tsample,
117                              0, pdur)
118         off = MonophasicPulse('cathodic', pdur,
119                               tsample, 0, pdur)
120         on.append(off)
121         utils.TimeSeries.__init__(self, tsample, on)
```

Implementation Details

pulse2percept's main technical challenge is computational cost: the simulations require a fine subsampling of space, and span several orders of magnitude in time. In the space domain the software needs to be able to simulate electrical activation of individual retinal ganglion cells on the order of microns. In the time domain the model needs to be capable of dealing with pulse trains containing individual pulses on the sub-millisecond time scale that last over several seconds.

Like the brain, we solve this problem through parallelization. Spatial interactions are confined to an initial stage of processing (Fig. 3 A, B), after which all spatial computations are parallelized using two back ends (Joblib [Job16] and Dask [Das16]), with both multithreading and multiprocessing options.

However, even after parallelization, computing the temporal response remains a computational bottleneck. Initial stages of the temporal model require convolutions of arrays (e.g., Eqs. 2 and 3) that describe responses of the model at high temporal resolution (sampling rates on the order of 10 microseconds) for pulse trains lasting for at least 500 milliseconds. These numerically-heavy sections of the code are sped up using a conjunction of three strategies. First, as described above, any given electrode generally only stimulates a subregion of the retina. As a consequence, when only a few electrodes are active, we can often obtain substantial speed savings by ignoring pixels which are not significantly stimulated by any electrode (see tolerance parameter `tol` on line 38 of the example code). Second, electrical stimulation is often carried out at relatively low pulse train frequencies of less than 50 Hz. Since the individual pulses within the pulse train are usually very short (~75-450 microseconds), input pulse trains tend to be sparse. We can exploit this fact to speed up computation time by avoiding direct convolution with the entire time-series whenever possible,

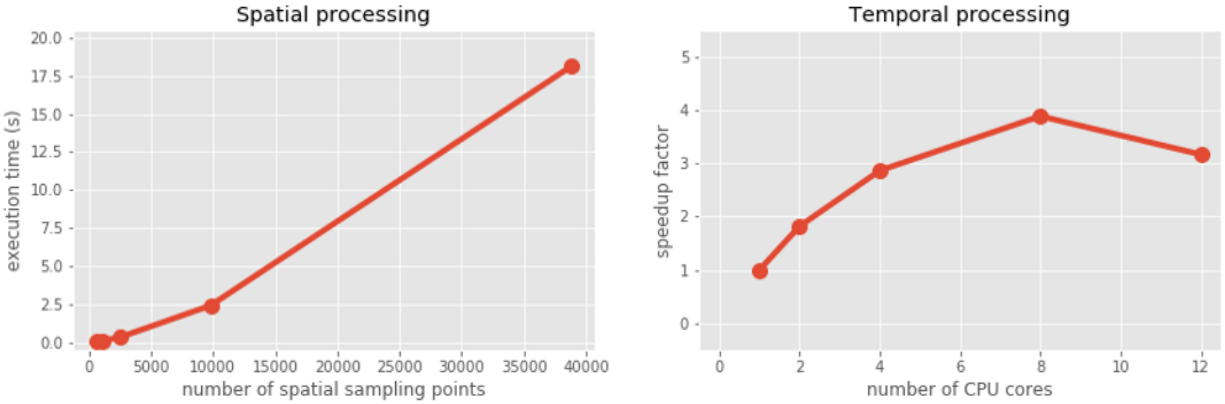


Fig. 5: *Computational performance. (A) Compute time to generate an 'effective stimulation map' is shown as a function of the number of spatial sampling points used to characterize the retina. (B) Speedup factor (serial execution time / parallel execution time) is shown as a function of the number of CPU cores. Execution times were collected for the an Argus II array (60 electrodes) simulating the letter 'A' (roughly 40 active electrodes, 20 Hz/20 uA pulse trains) over a period of 500 ms (`tsample` was 10 microseconds, `ssample` was 50 microns). Joblib and Dask parallelization back ends gave similar results.*

and instead relying on a custom-built sparse convolution function. Preprocessing of sparse pulse train arrays allows us to carry out temporal convolution only for those parts of the time-series that include nonzero current amplitudes. Finally, these convolutions are sped up wih LLVM-base compilation implemented using Numba [LPS15].

Computational Performance

We measured computational performance of the model for both spatial and temporal processing using a 12-core Intel Core i7-5930K operating at 3.50 GHz (64GB of RAM).

The initial stages of the model scale as a function of the number of spatial sampling points in the retina, as shown in Fig. 5 A. Since the spatial arrangement of axonal pathways does not depend on the stimulation protocol or retinal implant, *pulse2percept* automatically stores and re-uses the generated spatial map depending on the specified set of spatial parameters.

The remainder of the model is carried out in parallel, with the resulting speedup factor shown in Fig. 5 B. Here, the speedup factor is calculated as the ratio of single-core execution time and parallel execution time. On this particular machine, the maximum speedup factor is obtained with eight cores, above which the simulation shifts from being CPU bound to being memory bound, leading to a decrease in speedup. At its best, simulating typical stimulation of an Argus II over a timecourse of 500 milliseconds (at 50 microns spatial resolution and 10 ms temporal resolution) required 79 seconds of execution time. According to line profiling, most of the time is spent executing the slow convolutions (Fig. 3 D, F), thus heavily relying on the computational performance of the SciPy implementation of the Fast Fourier Transform. Due to the current design constraints (see Discussion), the software is better suited for rapid prototyping rather than real-time execution - although we aim to alleviate this in the future through GPU parallelization (via CUDA [KPL+12] and Dask [Das16]) and cluster computing (via Spark [Apa16]).

Software availability and development

All code can be found at https://github.com/uwescience/pulse2percept, with up-to-date source code documentation available at https://uwescience.github.io/pulse2percept. In addition, the latest stable release is available on the Python Package Index and can be installed using pip:

```
$ pip install pulse2percept
```

The library's test suite can be run as follows:

```
$ py.test --pyargs pulse2percept --doctest-modules
```

All code presented in this paper is current as of the v0.2 release.

Discussion

We presented here an open-source, Python-based framework for modeling the visual processing in retinal prosthesis patients. This software generates a simulation of the perceptual experience of individual prosthetic users - a 'virtual patient'.

The goal of *pulse2percept* is to provide open-source simulations that can allow any user to evaluate the perceptual experiences likely to be produced across both current and future devices. Specifically, the software is designed to meet four software design specifications:

- *Ease of use*: The intended users of this simulation include researchers and government officials who collect or assess perceptual data on prosthetic implants. These researchers are likely to be MDs rather than computer scientists, and might therefore lack technical backgrounds in computing. In the future, we plan for *pulse2percept* to become the back end of a web application similar to [KDM+ss].
- *Modularity*: As research continues in this field, it is likely that the underlying computational models converting electrical stimulation to patient percepts will improve. The modular design of the current release makes it easy to update individual components of the model.
- *Flexibility*: *pulse2percept* allows for rapid prototyping and integration with other analysis or visualization libraries from the Python community. It allows users to play with parameters, and use the ones that fit their desired device. Indeed, within most companies the specifications of implants currently in design is closely guarded intellectual property.
- *Extensibility*: The software can easily be extended to include custom implants, stimulation protocols, and retinal models.

As a result of these design considerations, *pulse2percept* has a number of potential uses.

Device developers can use virtual patients to get an idea of how their implant will perform even before a physical prototype has been created. This is reminiscent of the practice of virtual prototyping in other engineering fields. It becomes possible to make predictions about the perceptual consequences of individual design considerations, such as specific electrode geometries and stimulation protocols. As a result, virtual patients provide a useful tool for implant development, making it possible to rapidly predict vision across different implant configurations. We are currently collaborating with two leading manufacturers to validate the use of this software for both of these purposes.

Virtual patients can also play an important role in the wider community. To a naive viewer, simulations of prosthetic vision currently provided by manufacturers and the press might provide misleading visual outcomes, because these simulations do not take account of the substantial distortions in space and time that are observed by actual patients. On our website we provide example stimulations of real-world vision based on the *pulse2percept* virtual patient.

Prosthetic implants are expensive technology - costing roughly $100k per patient. Currently, these implants are reimbursed on a trial-by-trial basis across many countries in Europe, and are only reimbursed in a subset of states in the USA. Hence our simulations can help guide government agencies such as the FDA and Medicare in reimbursement decisions. Most importantly, these simulations can help patients, their families, and doctors make an informed choice when deciding at what stage of vision loss a prosthetic device would be helpful.

Acknowledgments

Supported by the Washington Research Foundation Funds for Innovation in Neuroengineering and Data-Intensive Discovery (M.B.), by a grant from the Gordon & Betty Moore Foundation and the Alfred P. Sloan Foundation to the University of Washington eScience Institute Data Science Environment (M.B. and A.R.), and by the National Institutes of Health (NEI EY-012925 to G.M.B., EY-014645 to I.F.). Research credits for cloud computing were provided by Amazon Web Services.

References

[ABK+08] AK Ahuja, MR Behrend, M Kuroda, MS Humayun, and JD Weiland. An *in vitro* model of a retinal prosthesis. *IEEE Transactions on Biomedical Engineering*, 55(6):1744–1753, 2008.

[Apa16] Apache Spark Development Team. *Apache Spark: a fast and general cluster computing system for Big Data*, 2016. URL: https://spark.apache.org/releases/spark-release-2-0-0.html.

[BBB+84] CH Bunker, EL Berson, WC Bromley, RP Hayes, and TH Roderick. Prevalence of retinitis pigmentosa in maine. *American Journal of Ophthalmology*, 97:357–365, 1984.

[BRBFss] M Beyeler, A Rokem, GM Boynton, and I Fine. Learning to see again: Biological constraints on cortical plasticity and the implications for sight restoration technologies. *Journal of Neural Engineering*, in press. doi:10.1088/1741-2552/aa795e.

[BSD17] Berkeley Software Distribution (BSD) 3-clause "New" or "Revised" License, version 3, 2017. URL: https://github.com/uwescience/pulse2percept/blob/master/LICENSE.

[Das16] Dask Development Team. *Dask: Library for dynamic task scheduling*, 2016. URL: http://dask.pydata.org.

[dCDH+16] L da Cruz, JD Dorn, MS Humayun, G Dagnelie, J Handa, P-O Barale, J-A Sahel, PE Stanga, F Hafezi, and AB Safran et al. Five-year safety and performance results from the argus {II} retinal prosthesis system clinical trial. *Ophthalmology*, 123(10):2248 – 2254, 2016.

[FB15] I Fine and GM Boynton. Pulse trains to percepts: the challenge of creating a perceptually intelligible world with sight recovery technologies. 370(1677), 2015.

[FCL15] I Fine, C Cepko, and MS Landy. Vision Research special issue: Sight restoration: Prosthetics, optogenetics and gene therapy. *Vision Research*, 111:115–123, 2015.

[FFm10] FFmpeg Development Team. *FFmpeg: A complete, cross-platform solution to record, convert and stream audio and video*, 2010. URL: http://www.ffmpeg.org.

[Gro04] The Eye Diseases Prevalence Research Group*. Prevalence of age-related macular degeneration in the United States. *Archives of Ophthalmology*, 122(4):564–572, 2004.

[HGW+09] A Horsager, SH Greenwald, JD Weiland, MS Humayun, RJ Greenberg, MJ McMahon, GM Boynton, and I Fine. Predicting visual sensitivity in retinal prosthesis patients. *Investigative Ophthalmology & Visual Science*, 50(4):1483, 2009.

[HPdJ+99] M S Humayun, M Prince, E de Juan, Jr, Y Barron, M Moskowitz, I B Klock, and A H Milam. Morphometric analysis of the extramacular retina from postmortem eyes with retinitis pigmentosa. *Investigative Ophthalmology & Visual Science*, 40(1):143, 1999.

[JNS+09] NM Jansonius, J Nevalainen, B Selig, LM Zangwill, PA Sample, WM Budde, JB Jonas, WA Lagrèze, PJ Airaksinen, and R Vonthein et al. A mathematical description of nerve fiber bundle trajectories and their variability in the human retina. *Vision research*, 49(17):2157–2163, 2009.

[Job16] Joblib Development Team. *Joblib*, 2016. URL: http://pythonhosted.org/joblib/.

[KDM+ss] A Keshavan, E Datta, IM McDonough, CR Madan, K Jordan, and RG Henry. Mindcontrol: A web application for brain segmentation quality control. *NeuroImage*, in press. doi:10.1016/j.neuroimage.2017.03.055.

[KPL+12] A Klöckner, N Pinto, Y Lee, B Catanzaro, P Ivanov, and A Fasih. PyCUDA and PyOpenCL: A Scripting-Based Approach to GPU Run-Time Code Generation. *Parallel Computing*, 38(3):157–174, 2012.

[LPS15] SK Lam, A Pitrou, and S 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.

[MJ03] RE Marc and BW Jones. Retinal remodeling in inherited photoreceptor degenerations. *Molecular Neurobiology*, 28:139–147, 2003.

[MJWS03] RE Marc, BW Jones, CB Watt, and E Strettoi. Neural remodeling in retinal degeneration. *Progress in Retinal and Eye Research*, 22:607–655, 2003.

[MNS08] F Mazzoni, E Novelli, and E Strettoi. Retinal ganglion cells survive and maintain normal dendritic morphology in a mouse model of inherited photoreceptor degeneration. *Journal of Neuroscience*, 28:14282–14292, 2008.

[NFH+12] D Nanduri, I Fine, A Horsager, GM Boynton, MS Humayun, RJ Greenberg, and JD Weiland. Frequency and amplitude modulation have different effects on the percepts elicited by retinal stimulation. *Investigative Ophthalmology & Visual Science*, 53(1):205–214, 2012.

[Pre15] Pioneer Press. 'bionic eye' helps duluth man make out his dark world. *Twin Cities Pioneer Press*, 2015.

[Rod98] RW Rodieck. *The first steps in seeing*. Sinauer Associates, 1998.

[S v14] S van der Walt, JL Schönberger, J Nunze-Iglesias, F Boulogne, JD Warner, N Yager, E Gouillart, T Yu and scikit-image contributors. scikit-image: Image processing in Python. *PeerJ*, 2(e453), 2014.

[SBSB+15] K Stingl, KU Bartz-Schmidt, D Besch, CK Chee, CL Cottriall, F Gekeler, M Groppe, TL Jackson, RE MacLaren, and A Koitschev et al. Subretinal visual implant alpha {IMS} – clinical trial interim report. *Vision Research*, 111, Part B:149 – 160, 2015.

[Sci17] Scikit-Video Development Team. *Scikit-Video: Video processing in Python*, 2017. URL: http://www.scikit-video.org.

[Sha89] RV Shannon. A model of threshold for pulsatile electrical stimulation. *Hearing Research*, 40:197–2014, 1989.

[SHd+97] A Santos, MS Humayun, E de Juan Jr., RJ Greenburg, MJ Marsh, IB Klock, and AH Milam. Preservation of the inner retina in retinitis pigmentosa. *Archives of Ophthalmology*, 115:511–515, 1997.

[WWH16] JD Weiland, ST Walston, and MS Humayun. Electrical stimulation of the retina to produce artificial vision. *Annual Review of Vision Science*, 2:273–294, 2016.

Optimised finite difference computation from symbolic equations

Michael Lange[‡*], Navjot Kukreja[‡], Fabio Luporini[‡], Mathias Louboutin[¶], Charles Yount[§], Jan Hückelheim[‡], Gerard J. Gorman[‡]

https://youtu.be/KinmqFTEs94

Abstract—Domain-specific high-productivity environments are playing an increasingly important role in scientific computing due to the levels of abstraction and automation they provide. In this paper we introduce Devito, an open-source domain-specific framework for solving partial differential equations from symbolic problem definitions by the finite difference method. We highlight the generation and automated execution of highly optimized stencil code from only a few lines of high-level symbolic Python for a set of scientific equations, before exploring the use of Devito operators in seismic inversion problems.

Index Terms—Finite difference, domain-specific languages, symbolic Python

Introduction

Domain-specific high-productivity environments are playing an increasingly important role in scientific computing. The level of abstraction and automation provided by such frameworks not only increases productivity and accelerates innovation, but also allows the combination of expertise from different specialised disciplines. This synergy is necessary when creating the complex software stack needed to solve leading edge scientific problems, since domain specialists as well as high performance computing experts are required to fully leverage modern computing architectures. Based on this philosophy we introduce Devito [Lange17], an open-source domain-specific framework for solving partial differential equations (PDE) from symbolic problem definitions by the finite difference method.

Symbolic computation, where optimized numerical code is automatically derived from a high-level problem definition, is a powerful technique that allows domain scientists to focus on algorithmic development rather than implementation details. For this reason Devito exposes an API based on Python (SymPy) [Meurer17] that allow users to express equations symbolically, from which it generates and executes optimized stencil code via just-in-time (JIT) compilation. Using latest advances in stencil compiler research, Devito thus provides domain scientists with the ability to quickly and efficiently generate high-performance kernels from only a few lines of Python code, making Devito composable with existing open-source software.

While Devito was originally developed for seismic imaging workflows, the automated generation and optimization of stencil codes can be utilised for a much broader set of computational problems. Matrix-free stencil operators based on explicit finite difference schemes are widely used in industry and academic research, although they merely represent one of many approaches to solving PDEs [Baba16], [Liu09], [Rai91]. In this paper we therefore limit our discussion of numerical methods and instead focus on the ease with which these operators can be created symbolically. We give a brief overview of the design concepts and key features of Devito and demonstrate its API using a set of classic examples from computational fluid dynamics (CFD). Then we will discuss the use of Devito in an example of a complex seismic inversion algorithm to illustrate its use in practical scientific applications and to showcase the performance achieved by the auto-generated and optimised code.

Background

The attraction of using domain-specific languages (DSL) to solve PDEs via a high-level mathematical notation is by no means new and has led to various purpose-built software packages and compilers dating back to 1962 [Iverson62], [Cardenas70], [Umetani85], [Cook88], [VanEngelen96]. Following the emergence of Python as a widely used programming language in scientific research, embedded DSLs for more specialised domains came to the fore, most notably the FEniCS [Logg12] and Firedrake [Rathgeber16] frameworks, which both implement the unified Form Language (UFL) [Alnaes14] for the symbolic definition of finite element problems in the weak form. The increased level of abstraction that such high-level languages provide decouples the problem definition from its implementation, thus allowing domain scientists and mathematicians to focus on more advanced methods, such as the automation of adjoint models as demonstrated by Dolfin-Adjoint [Farrell13].

The performance optimization of stencil computation on regular cartesian grids for high-performance computing applications has also received much attention in computer science research [Datta08], [Brandvik10], [Zhang12], [Henretty13], [Yount15]. The primary focus of most stencil compilers or DSLs, however, is the optimization of synthetic problems which often limits their

* Corresponding author: michael.lange@imperial.ac.uk
‡ Imperial College London
¶ The University of British Columbia
§ Intel Corporation

applicability for practical scientific applications. The primary consideration here is that most realistic problems often require more than just a fast and efficient PDE solver, which entails that symbolic DSLs embedded in Python can benefit greatly from native interoperability with the scientific Python ecosystem.

Design and API

The primary objective of Devito is to enable the quick and effective creation of highly optimised finite difference operators for use in a realistic scientific application context. As such, its design is centred around composability with the existing Python software stack to provide users with the tools to dynamically generate optimised stencil computation kernels and to enable access to the full scientific software ecosystem. In addition, to accommodate the needs of "real life" scientific applications, a secondary API is provided that enables users to inject custom expressions, such as boundary conditions or sparse point interpolation routines, into the generated kernels.

The use of SymPy as the driver for the symbolic generation of stencil expressions and the subsequent code-generation are at the heart of the Devito philosophy. While SymPy is fully capable of auto-generating low-level C code for pre-compiled execution from high-level symbolic expressions, Devito is designed to combine these capabilities with automatic performance optimization based on the latest advances in stencil compiler technology. The result is a framework that is capable of automatically generating and optimising complex stencil code from high-level symbolic definitions.

The Devito API is based around two key concepts that allow users to express finite difference problems in a concise symbolic notation:

- **Symbolic data objects:** Devito's high-level symbolic objects behave like `sympy.Function` objects and provide a set of shorthand notations for generating derivative expressions, while also managing user data. The rationale for this duality is that many stencil optimization algorithms rely on data layout changes, mandating that Devito needs to be in control of data allocation and access.
- **Operator:** An `Operator` creates, compiles and executes a single executable kernel from a set of SymPy expressions. The code generation and optimization process involves various stages and accepts a mixture of high-level and low-level expressions to allow the injection of customised code.

Fluid Dynamics Examples

In the following section we demonstrate the use of the Devito API to implement two examples from classical fluid dynamics, before highlighting the role of Devito operators in a seismic inversion context. Both CFD examples are based in part on tutorials from the introductory blog "CFD Python: 12 steps to Navier-Stokes"[1] by the Lorena A. Barba group. We have chosen the examples in this section for their relative simplicity to concisely illustrate the capabilities and API features of Devito. For a more complete discussion on numerical methods for fluid flows please refer to [Peiro05].

1. http://lorenabarba.com/blog/cfd-python-12-steps-to-navier-stokes/

Linear Convection

We will demonstrate a basic Devito operator definition based on a linear two-dimensional convection flow (step 5 in the original tutorials)[2]. The governing equation we are implementing here is:

$$\frac{\partial u}{\partial t} + c\frac{\partial u}{\partial x} + c\frac{\partial u}{\partial y} = 0 \tag{1}$$

A discretised version of this equation, using a forward difference scheme in time and a backward difference scheme in space might be written as

$$u_{i,j}^{n+1} = u_{i,j}^{n} - c\frac{\Delta t}{\Delta x}(u_{i,j}^{n} - u_{i-1,j}^{n}) - c\frac{\Delta t}{\Delta y}(u_{i,j}^{n} - u_{i,j-1}^{n}) \tag{2}$$

where the subscripts i and j denote indices in the space dimensions and the superscript n denotes the index in time, while Δt, Δx, Δy denote the spacing in time and space dimensions respectively.

The first thing we need is a function object with which we can build a timestepping scheme. For this purpose Devito provides so-called `TimeData` objects that encapsulate functions that are differentiable in space and time. With this we can derive symbolic expressions for the backward derivatives in space directly via the `u.dxl` and `u.dyl` shorthand expressions (the `l` indicates "left" or backward differences) and the shorthand notation `u.dt` provided by `TimeData` objects to derive the forward derivative in time.

```
from devito import *

c = 1.
u = TimeData(name='u', shape=(nx, ny))

eq = Eq(u.dt + c * u.dxl + c * u.dyl)

[In] print eq
[Out] Eq(-u(t, x, y)/s + u(t + s, x, y)/s
        + 2.0*u(t, x, y)/h - 1.0*u(t, x, y - h)/h
        - 1.0*u(t, x - h, y)/h, 0)
```

The above expression results in a `sympy.Equation` object that contains the fully discretised form of Eq. 1, including placeholder symbols for grid spacing in space (`h`, assuming $\Delta x = \Delta y$) and time (`s`). These spacing symbols will be resolved during the code generation process, as described in the code generation section. It is also important to note here that the explicit generation of the space derivatives `u_dx` and `u_dy` is due to the use of a backward derivative in space to align with the original example. A similar notation to the forward derivative in time (`u.dt`) will soon be provided.

In order to create a functional `Operator` object, the expression `eq` needs to be rearranged so that we may solve for the unknown $u_{i,j}^{n+1}$. This is easily achieved by using SymPy's `solve` utility and the Devito shorthand `u.forward` which denotes the furthest forward stencil point in a time derivative ($u_{i,j}^{n+1}$).

```
from sympy import solve

stencil = solve(eq, u.forward)[0]

[In] print(stencil)
[Out] (h*u(t, x, y) - 2.0*s*u(t, x, y)
     + s*u(t, x, y - h) + s*u(t, x - h, y))/h
```

The above variable `stencil` now represents the RHS of Eq. 2, allowing us to construct a SymPy expression that updates $u_{i,j}^{n+1}$ and build a `devito.Operator` from it. When creating this operator

2. http://nbviewer.jupyter.org/github/opesci/devito/blob/master/examples/cfd/test_01_convection_revisited.ipynb

we also supply concrete values for the spacing terms `h` and `s` via an additional substitution map argument `subs`.

```
op = Operator(Eq(u.forward, stencil),
              subs={h: dx, s:dt})

# Set initial condition as a smooth function
init_smooth(u.data, dx, dy)

op(u=u, time=100)  # Apply for 100 timesteps
```

Using this operator we can now create a similar example to the one presented in the original tutorial by initialising the data associated with the symbolic function u, `u.data` with an initial flow field. However, to avoid numerical errors due to the discontinuities at the boundary of the original "hat function", we use the following smooth initial condition provided by [Krakos12], as depicted in Figure 1.

$$u_0(x,y) = 1 + u\left(\frac{2}{3}x\right) * u\left(\frac{2}{3}y\right)$$

The final result after executing the operator for $5s$ (100 timesteps) is depicted in Figure 2. The result shows the expected displacement of the initial shape, in accordance with the prescribed velocity ($c = 1.0$), closely mirroring the displacement of the "hat function" in the original tutorial. It should also be noted that, while the results show good agreement with expectations by visual inspection, they do not represent an accurate solution to the linear convection equation. In particular, the low order spatial discretisation introduces numerical diffusion that causes a decrease in the peak velocity. This is a well-known issue that could be addressed with more sophisticated solver schemes as discussed in [LeVeque92].

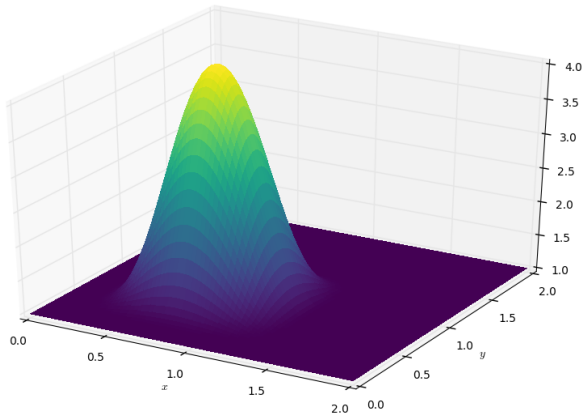


Fig. 1: *Initial condition of* `u.data` *in the 2D convection example.*

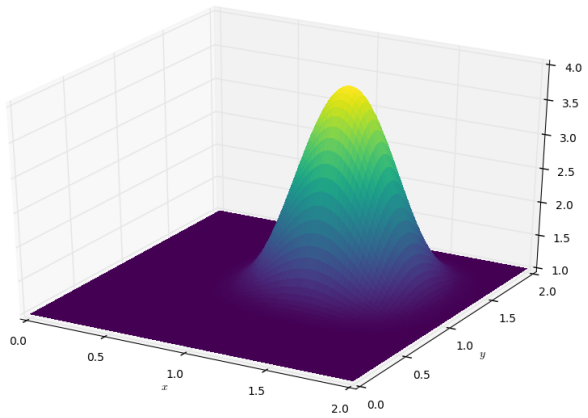


Fig. 2: *State of* `u.data` *after 100 timesteps in convection example.*

Laplace equation

The above example shows how Devito can be used to create finite difference stencil operators from only a few lines of high-level symbolic code. However, the previous example only required a single variable to be updated, while more complex operators might need to execute multiple expressions simultaneously, for example to solve coupled PDEs or apply boundary conditions as part of the time loop. For this reason `devito.Operator` objects can be constructed from multiple update expressions and allow mutiple expression formats as input.

Nevertheless, boundary conditions are currently not provided as part of the symbolic high-level API. For exactly this reason, Devito provides a low-level, or "indexed" API, where custom SymPy expressions can be created with explicitly resolved grid accesses to manually inject custom code into the auto-generation toolchain. This entails that future extensions to capture different types of boundary conditions can easily be added at a later stage.

To illustrate the use of the low-level API, we will use the Laplace example from the original CFD tutorials (step 9), which implements the steady-state heat equation with Dirichlet and Neuman boundary conditions[3]. The governing equation for this problem is

$$\frac{\partial^2 p}{\partial x^2} + \frac{\partial^2 p}{\partial y^2} = 0 \tag{3}$$

The rearranged discretised form, assuming a central difference scheme for second derivatives, is

$$p^n_{i,j} = \frac{\Delta y^2 (p^n_{i+1,j} + p^n_{i-1,j}) + \Delta x^2 (p^n_{i,j+1} + p^n_{i,j-1})}{2(\Delta x^2 + \Delta y^2)} \tag{4}$$

Using a similar approach to the previous example, we can construct the SymPy expression to update the state of a field p. For demonstration purposes we will use two separate function objects of type `DenseData` in this example, since the Laplace equation does not contain a time-dependence. The shorthand expressions `pn.dx2` and `pn.dy2` hereby denote the second derivatives in x and y.

```
# Create two separate symbols with space dimensions
p = DenseData(name='p', shape=(nx, ny),
              space_order=2)
pn = DenseData(name='pn', shape=(nx, ny),
               space_order=2)

# Define equation and solve for center point in `pn`
eq = Eq(a * pn.dx2 + pn.dy2)
stencil = solve(eq, pn)[0]
# The update expression to populate buffer `p`
eq_stencil = Eq(p, stencil)
```

Just as the original tutorial, our initial condition in this example is $p = 0$ and the flow will be driven by the boundary conditions

$$\begin{aligned} p &= 0 \quad && \text{at } x = 0 \\ p &= y \quad && \text{at } x = 2 \\ \frac{\partial p}{\partial y} &= 0 \quad && \text{at } y = 0,\ 1 \end{aligned}$$

To implement these BCs we can utilise the `.indexed` property that Devito symbols provide to get a symbol of type `sympy.IndexedBase`, which in turn allows us to use matrix indexing notation (square brackets) to create symbols of type `sympy.Indexed` instead of `sympy.Function`. This notation allows users to hand-code stencil expressions using explicit relative grid indices, for example `p[x, y] - p[x-1, y] / h` for the discretized backward derivative $\frac{\partial u}{\partial x}$. The symbols `x` and `y` hereby represent the respective problem dimensions and cause the expression to be executed over the entire data dimension, similar to Python's `:` operator.

The Dirichlet BCs in the Laplace example can thus be implemented by creating a `sympy.Eq` object that assigns either fixed values or a prescribed function, such as the utility symbol `bc_right` in our example, along the left and right boundary of the domain. To implement the Neumann BCs we again follow the original tutorial by assigning the second grid row from the top and bottom boundaries the value of the outermost row. The resulting SymPy expressions can then be used alongside the state update expression to create our `Operator` object.

```
# Create an additional symbol for our prescibed BC
bc_right = DenseData(name='bc_right', shape=(nx, ),
                     dimensions=(x, ))
bc_right.data[:] = np.linspace(0, 1, nx)

# Create explicit boundary condition expressions
bc = [Eq(p.indexed[x, 0], 0.)]
bc += [Eq(p.indexed[x, ny-1], bc_right.indexed[x])]
bc += [Eq(p.indexed[0, y], p.indexed[1, y])]
bc += [Eq(p.indexed[nx-1, y], p.indexed[nx-2, y])]

# Build operator with update and BC expressions
op = Operator(expressions=[eq_stencil] + bc,
              subs={h: dx, a: 1.})
```

After building the operator, we can now use it in a time-independent convergence loop that minimizes the L^1 norm of p. However, in this example we need to make sure to explicitly exchange the role of the buffers `p` and `pn`. This can be achieved by supplying symbolic data objects via keyword arguments when invoking the operator, where the name of the argument is matched against the name of the original symbol used to create the operator.

The convergence criterion for this example is defined as the relative error between two iterations and set to $\|p\|^1 < 10^{-4}$. The corresponding initial condition and the resulting steady-state solution, depicted in Figures 3 and 4 respectively, agree with the original tutorial implementation. It should again be noted that the chosen numerical scheme might not be optimal to solve steady-state problems of this type, since implicit methods are often preferred.

```
l1norm = 1
counter = 0
while l1norm > 1.e-4:
    # Determine buffer order
    if counter % 2 == 0:
        _p, _pn = p, pn
    else:
        _p, _pn = pn, p

    # Apply operator
    op(p=_p, pn=_pn)

    # Compute L1 norm
    l1norm = (np.sum(np.abs(_p.data[:])
              - np.abs(_pn.data[:]))
              / np.sum(np.abs(_pn.data[:])))
    counter += 1
```

Seismic Inversion Example

The primary motivating application behind the design of Devito is the solution of seismic exploration problems that require highly optimised wave propagation operators for forward modelling and

3. http://nbviewer.jupyter.org/github/opesci/devito/blob/master/examples/cfd/test_05_laplace.ipynb

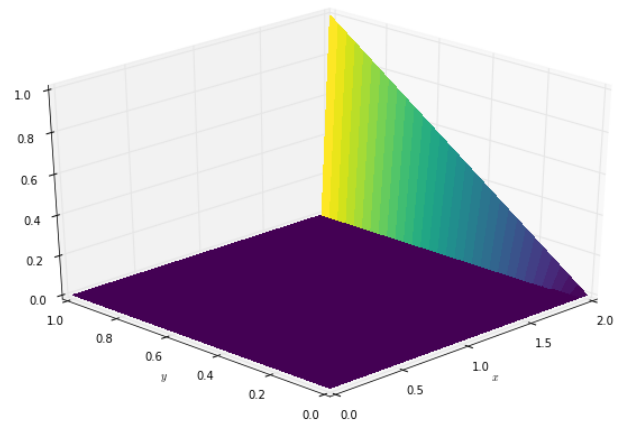


Fig. 3: *Initial condition of* `p.data` *in the 2D Laplace example.*

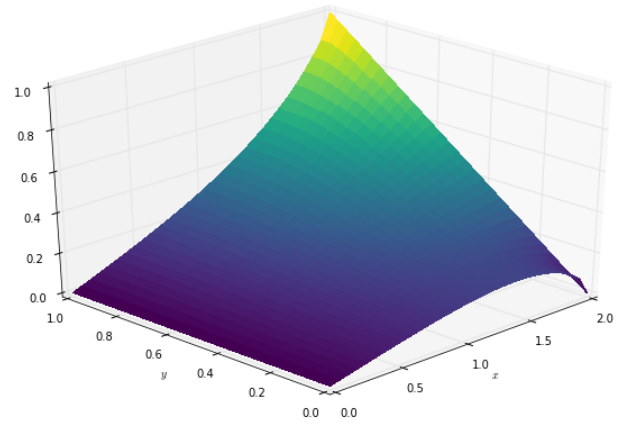


Fig. 4: *State of* `p.data` *after convergence in Laplace example.*

adjoint-based inversion. Obviously, the speed and accuracy of the generated kernels are of vital importance. Moreover, the ability to efficiently define rigorous forward modelling and adjoint operators from high-level symbolic definitions also implies that domain scientists are able to quickly adjust the numerical method and discretisation to the individual problem and hardware architecture [Louboutin17a].

In the following example we will show the generation of forward and adjoint operators for the acoustic wave equation and verify their correctness using the so-called *adjoint test* [Virieux09][4]. This test, also known as *dot product test*, verifies that the implementation of an adjoint operator indeed computes the conjugate transpose of the forward operator.

The governing wave equation for the forward operator is defined as

$$m\frac{\partial^2 u}{\partial t^2} + \eta\frac{\partial u}{\partial t} - \nabla^2 u = q$$

where u denotes the pressure wave field, m is the square slowness, q is the source term and η denotes the spatially varying dampening factor used to implement an absorbing boundary condition.

On top of fast stencil operators, seismic inversion kernels also rely on sparse point interpolation to inject the modelled wave as a point source (q) and to record the pressure at individual point locations. To accommodate this, Devito provides another symbolic data type `PointData`, which allows the generation of sparse-point interpolation expressions using the "indexed" low-level API. These symbolic objects provide utility routines `pt.interpolate(expression)` and `pt.inject(field, expression)` to create symbolic expressions that perform linear interpolation between the sparse points and the cartesian grid for insertion into `Operator` kernels. A separate set of explicit coordinate values is associated with the sparse point objects for this purpose in addition to the function values stored in the `data` property.

Adjoint Test

The first step for implementing the adjoint test is to build a forward operator that models the wave propagating through an isotropic medium, where the square slowness of the wave is denoted as *m*. Since `m`, as well as the boundary dampening function `eta`, is re-used between forward and adjoint runs the only symbolic data object we need to create here is the wavefield `u` in order to implement and rearrange our discretised equation `eqn` to form the update expression for `u`. It is worth noting that the `u.laplace` shorthand notation used here expands to the set of second derivatives in all spatial dimensions, thus allowing us to use the same formulation for two-dimensional and three-dimensional problems.

In addition to the state update of `u`, we are also inserting two additional terms into the forward modelling operator:

- `src_term` injects a pressure source at a point location according to a prescribed time series stored in `src.data` that is accessible in symbolic form via the symbol `src`. The scaling factor in `src_term` is coded by hand but can be automatically inferred.
- `rec_term` adds the expression to interpolate the wavefield `u` for a set of "receiver" hydrophones that measure the propagated wave at varying distances from the source for every time step. The resulting interpolated point data will be stored in `rec.data` and is accessible to the user as a NumPy array.

```
def forward(model, m, eta, src, rec, order=2):
    # Create the wavefeld function
    u = TimeData(name='u', shape=model.shape,
                 time_order=2, space_order=order)

    # Derive stencil from symbolic equation
    eqn = m * u.dt2 - u.laplace + eta * u.dt
    stencil = solve(eqn, u.forward)[0]
    update_u = [Eq(u.forward, stencil)]


    # Add source injection and receiver interpolation
    src_term = src.inject(field=u,
                          expr=src * dt**2 / m)
    rec_term = rec.interpolate(expr=u)

    # Create operator with source and receiver terms
    return Operator(update_u + src_term + rec_term,
                    subs={s: dt, h: model.spacing})
```

After building a forward operator, we can now implement the adjoint operator in a similar fashion. Using the provided symbols `m` and `eta`, we can again define the adjoint wavefield `v` and implement its update expression from the discretised equation. However, since the adjoint operator needs to operate backwards in time there are two notable differences:

4. http://nbviewer.jupyter.org/github/opesci/devito/blob/master/examples/seismic/tutorials/test_01_modelling.ipynb

- The update expression now updates the backward stencil point in the time derivative $v_{i,j}^{n-1}$, denoted as `v.backward`. In addition to that, the `Operator` is forced to reverse its internal time loop by providing the argument `time_axis=Backward`
- Since the acoustic wave equation is self-adjoint without dampening, the only change required in the governing equation is to invert the sign of the dampening term `eta * u.dt`. The first derivative is an antisymmetric operator and its adjoint minus itself.

Moreover, the role of the sparse point objects has now switched: Instead of injecting the source term, we are now injecting the previously recorded receiver values into the adjoint wavefield, while we are interpolating the resulting wave at the original source location. As the injection and interpolations are part of the kernel, we also insure that these two are adjoints of each other.

```
def adjoint(model, m, eta, srca, rec, order=2):
    # Create the adjoint wavefeld function
    v = TimeData(name='v', shape=model.shape,
                 time_order=2, space_order=order)

    # Derive stencil from symbolic equation
    # Note the inversion of the dampening term
    eqn = m * v.dt2 - v.laplace - eta * v.dt
    stencil = solve(eqn, u.forward)[0]
    update_v = [Eq(v.backward, stencil)]

    # Inject the previous receiver readings
    rec_term = rec.inject(field=v,
                          expr=rec * dt**2 / m)

    # Interpolate the adjoint-source
    srca_term = srca.interpolate(expr=v)

    # Create operator with source and receiver terms
    return Operator(update_v + rec_term + srca_term,
                    subs={s: dt, h: model.spacing},
                    time_axis=Backward)
```

Having established how to build the required operators we can now define the workflow for our adjoint example. For illustration purposes we are using a utility object `Model` that provides the core information for seismic inversion runs, such as the values for `m` and the dampening term `eta`, as well as the coordinates of the point source and receiver hydrophones. It is worth noting that the spatial discretisation and thus the stencil size of the operators is still fully parameterisable.

```
# Create the seismic model of the domain
model = Model(...)

# Create source with Ricker wavelet
src = PointData(name='src', ntime=ntime,
                ndim=2, npoint=1)
src.data[0, :] = ricker_wavelet(ntime)
src.coordinates.data[:] = source_coords

# Create empty set of receivers
rec = PointData(name='rec', ntime=ntime,
                ndim=2, npoint=101)
rec.coordinates.data[:] = receiver_coords

# Create empty adjoint source symbol
srca = PointData(name='srca', ntime=ntime,
                 ndim=2, npoint=1)
srca.coordinates.data[:] = source_coords

# Create symbol for square slowness
m = DenseData(name='m', shape=model.shape,
              space_order=order)
```

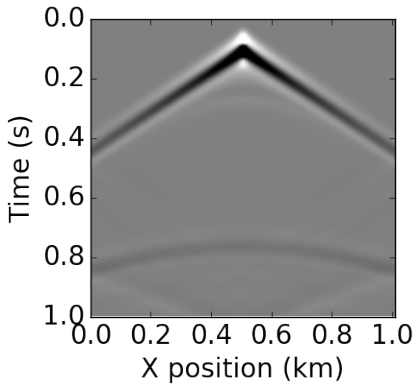


Fig. 5: *Shot record of the measured point values in* `rec.data` *after the forward run.*

```
m.data[:] = model  # Set m from model data


# Create dampening term from model
eta = DenseData(name='eta', shape=shape,
                space_order=order)
eta.data[:] = model.dampening

# Execute foward and adjoint runs
fwd = forward(model, m, eta, src, rec)
fwd(time=ntime)
adj = adjoint(model, m, eta, srca, rec)
adj(time=ntime)

# Test prescribed against adjoint source
adjoint_test(src.data, srca.data)
```

The adjoint test is the core definition of the adjoint of a linear operator. The mathematical correctness of the adjoint is required for mathematical adjoint-based optimizations methods that are only guarantied to converged with the correct adjoint. The test can be written as:

$$< src,\ adjoint(rec) >=< forward(src),\ rec >$$

The adjoint test can be used to verify the accuracy of the forward propagation and adjoint operators and has been shown to agree for 2D and 3D implementations [Louboutin17b]. The shot record of the data measured at the receiver locations after the forward run is shown in Figure 5.

Automated code generation

The role of the `Operator` in the previous examples is to generate semantically equivalent C code to the provided SymPy expressions, complete with loop constructs and annotations for performance optimization, such as OpenMP pragmas. Unlike many other DSL-based frameworks, Devito employs actual compiler technology during the code generation and optimization process. The symbolic specification is progressively lowered to C code through a series of passes manipulating abstract syntax trees (AST), rather than working with rigid templates. This software engineering choice has an invaluable impact on maintainability, extensibility and composability.

Following the initial resolution of explicit grid indices into the low-level format, Devito is able to apply several types of automated performance optimization throughout the code generation pipeline, which are grouped into two distinct sub-modules:

- **DSE - Devito Symbolic Engine:** The first set of optimization passes consists of manipulating SymPy equations with the aim to decrease the number of floating-point operations performed when evaluating a single grid point. This initial optimization is performed following an initial analysis of the provided expressions and consists of sub-passes such as common sub-expressions elimination, detection and promotion of time-invariants, and factorization of common finite-difference weights. These transformations not only optimize the operation count, but they also improve the symbolic processing and low-level compilation times of later processing stages.
- **DLE - Devito Loop Engine:** After the initial symbolic processing Devito schedules the optimised expressions in a set of loops by creating an Abstract Syntax Tree (AST). The loop engine (DLE) is now able to perform typical loop-level optimizations in mutiple passes by manipulating this AST, including data alignment through array annotations and padding, SIMD vectorization through OpenMP pragmas and thread parallelism through OpenMP pragmas. On top of that, loop blocking is used to fully exploit the memory bandwidth of a target architecture by increasing data locality and thus cache utilization. Since the effectiveness of the blocking technique is highly architecture-dependent, Devito can determine optimal block size through runtime auto-tuning.

Performance Benchmark

The effectiveness of the automated performance optimization performed by the Devito backend engines can be demonstrated using the forward operator constructed in the above example. The following performance benchmarks were run with for a three-dimensional grid of size $512 \times 512 \times 512$ with varying spatial discretisations resulting in different stencil sizes with increasing operational intensity (OI). The benchmark runs were performed on on a Intel(R) Xeon E5-2620 v4 2.1Ghz "Broadwell" CPU with a single memory socket and 8 cores per socket and the slope of the roofline models was derived using the Stream Triad benchmark [McCalpin95].

The first set of benchmark results, shown in Figure 6, highlights the performance gains achieved through loop-level optimizations. For these runs the symbolic optimizations were kept at a "basic" setting, where only common sub-expressions elimination is performed on the kernel expressions. Of particular interest are the performance gains achieved by increasing the loop engine mode from "basic" to "advanced", to insert loop blocking and explicit vectorization directives into the generated code. Due to the improved memory bandwidth utilization the performance increased to between 52% and 74% of the achievable peak. It is also worth noting that more aggressive optimization in the "speculative" DLE mode (directives for non-temporal stores and row-wise data alignment through additional padding) did not yield any consistent improvements due to the low OI inherent to the acoustic formulation of the wave equation and the subsequent memory bandwidth limitations of the kernel.

On top of loop-level performance optimizations, Figure 7 shows the achieved performance with additional symbolic optimizations and flop reductions enabled. While the peak performance shows only small effects from this set of optimizations due to the inherent memory bandwidth limitations of the kernel, it is interesting to note a reduction in operational intensity between

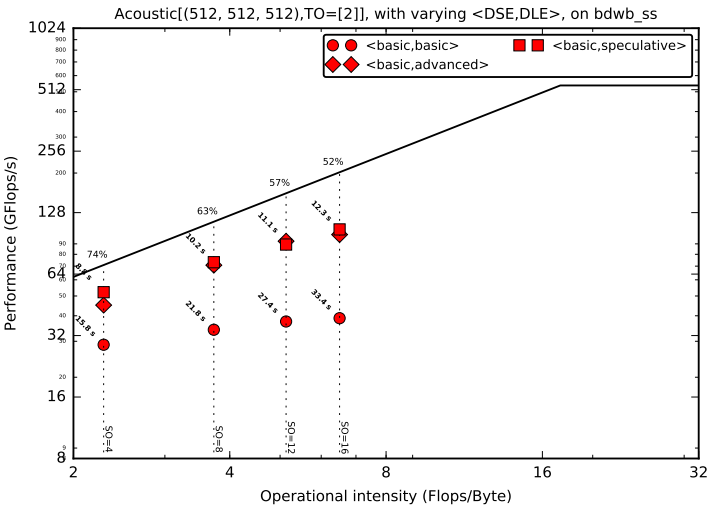


Fig. 6: *Performance benchmarks for loop-level optimizations with different spatial orders (SO). The symbolic optimisations (DSE) have been kept at level 'basic', while loop optimisation levels (DLE) vary.*

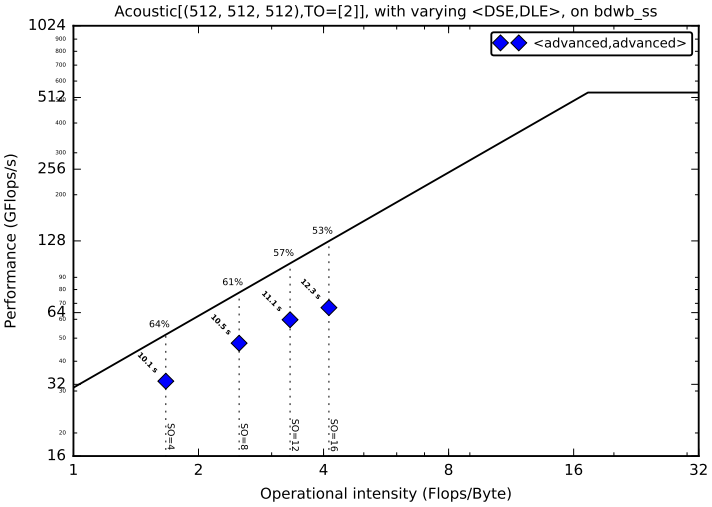


Fig. 7: *Performance benchmarks with full symbolic and loop-level optimizations for different spatial orders (SO).*

equivalent stencil sizes in Figures 6 and 7. This entails that, despite only marginal runtime changes, the generated code is performing less flops per stencil point, which is of vital importance for compute-dominated kernels with large OI [Louboutin17a].

Integration with YASK

As mentioned previously, Devito is based upon actual compiler technology with a highly modular structure. Each backend transformation pass is based on manipulating an input AST and returning a new, different AST. One of the reasons behind this software engineering strategy, which is clearly more challenging than a template-based solution, is to ease the integration of external tools, such as the YASK stencil optimizer [Yount16]. We are currently in the process of integrating YASK to complement the DLE, so that YASK may replace some (but not all) DLE passes.

The DLE passes are organized in a hierarchy of classes where each class represents a specific code transformation pipeline

based on AST manipulations. Integrating YASK becomes then a conceptually simple task, which boils down to three actions:

1) Adding a new transformation pipeline to the DLE.
2) Adding a new array type, to ease storage layout transformations and data views (YASK employs a data layout different than the conventional row-major format).
3) Creating the proper Python bindings in YASK so that Devito can drive the code generation process.

It has been shown that real-world stencil codes optimised through YASK may achieve an exceptionally high fraction of the attainable machine peak [Yount15], [Yount16]. Further, initial prototyping (manual optimization of Devito-generated code through YASK) revealed that YASK may also outperform the loop optimization engine currently available in Devito, besides ensuring seamless performance portability across a range of computer architectures. On the other hand, YASK is a C++ based framework that, unlike Devito, does not rely on symbolic mathematics and processing; in other words, it operates at a much lower level of abstraction. These observations, as well as the outcome of the initial prototyping phase, motivate the on-going Devito-YASK integration effort.

Discussion

In this paper we present the finite difference DSL Devito and demonstrate its high-level API to generate two fluid dynamics operators and a full seismic inversion example. We highlight the relative ease with which to create complex operators from only a few lines of high-level Python code while utilising highly optimised auto-generated C kernels via JIT compilation. On top of purely symbolic top-level API based on SymPy, we show how to utilise Devito's secondary API to inject custom expressions into the code generation toolchain to implement Dirichlet and Neumann boundary conditions, as well as the sparse-point interpolation routines required by seismic inversion operators.

Moreover, we demonstrate that Devito-generated kernels are capable of exploiting modern high performance computing architectures by achieving a significant percentage of machine peak. Devito's code-generation engines achieve this by automating well-known performance optimizations, as well as domain-specific optimizations, such as flop reduction techniques - all while maintaining full compatibility with the scientific software stack available through the open-source Python ecosystem.

Limitations and Future Work

The examples used in this paper have been chosen for their relative simplicity in order to concisely demonstrate the current features of the Devito API. Different numerical methods may be used to solve the presented examples with greater accuracy or achieve more realistic results. Nevertheless, finite difference methods play an important role and are widely used in academic and industrial research due to the relative ease of implementation, verification/validation and high computational efficiency, which is of particular importance for inversion methods that require fast and robust high-order PDE solvers.

The interfaces provided by Devito are intended to create high-performance operators with relative ease and thus increase user productivity. Several future extensions are planned to enhance the high-level API to further ease the construction of more complex operators, including explicit abstractions for symbolic boundary conditions, perfectly matched layer (PML) methods and staggered grids. Devito's secondary low-level API and use of several intermediate representations are intended to ease the gradual addition of new high-level features.

Moreover, the addition of YASK as an alternative backend will not only provide more advanced performance optimisation, but also an MPI infrastructure to allow Devito to utilise distribute computing environments. Further plans also exist for integration with linear and non-linear solver libraries, such as PETSc, to enable Devito to handle implicit formulations.

Acknowledgements

This work was financially supported in part by EPSRC grant EP/L000407/1 and the Imperial College London Intel Parallel Computing Centre. This research was carried out as part of the SINBAD project with the support of the member organizations of the SINBAD Consortium. Part of this work was supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Applied Mathematics and Computer Science programs under contract number DE-AC02-06CH11357.

References

[Alnaes14] M. S. Alnæs, A. Logg, K. B. Ølgaard, M. E. Rognes, and G. N. Wells, "Unified Form Language: a domain-specific language for weak formulations of partial differential equations", ACM Transactions on Mathematical Software (TOMS), vol. 40, no. 2, p. 9, 2014. https://dx.doi.org/10.1145/2566630

[Baba16] Y. Baba and V. Rakov, "The Finite-Difference Time Domain Method for Solving Maxwell's Equations", in "Electromagnetic Computation Methods for Lightning Surge Protection Studies", 2016, pp. 43–72, Wiley, ISBN 9781118275658. http://dx.doi.org/10.1002/9781118275658.ch3

[Brandvik10] T. Brandvik and G. Pullan, "Sblock: A framework for efficient stencil-based pde solvers on multi-core platforms", in "Proceedings of the 2010 10th IEEE International Conference on Computer and Information Technology", IEEE Computer Society, 2010, pp. 1181–1188. http://dx.doi.org/10.1109/CIT.2010.214

[Cardenas70] Cárdenas, A. F. and Karplus, W. J.: PDEL — a language for partial differential equations, Communications of the ACM, 13, 184–191, 1970.

[Cook88] Cook Jr, G. O.: ALPAL: A tool for the development of large-scale simulation codes, Tech. rep., Lawrence Livermore National Lab., CA (USA), 1988.

[Datta08] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick, "Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures", in Proceedings of the 2008 ACM/IEEE Conference on Supercomputing, IEEE Press, 2008, pp. 4:1–4:12. http://dl.acm.org/citation.cfm?id=1413370.1413375

[Farrell13] Farrell, P. E., Ham, D. A., Funke, S. W., and Rognes, M. E.: Automated Derivation of the Adjoint of High-Level Transient Finite Element Programs, SIAM Journal on Scientific Computing, 35, C369–C393, 2013. http://dx.doi.org/10.1137/120873558

[Henretty13] T. Henretty, R. Veras, F. Franchetti, L.-N. Pouchet, J. Ramanujam, and P. Sadayappan, "A stencil compiler for short-vector simd architectures," in Proceedings of the 27th International ACM Conference on International Conference on Supercomputing, ACM, 2013, pp. 13–24. http://doi.acm.org/10.1145/2464996.2467268

[Iverson62] Iverson, K.: A Programming Language, Wiley, 1962.

[Krakos12] J.A. Krakos, "Unsteady Adjoint Analysis for Output Sensitivity and Mesh Adaptation", PhD thesis, 2012. https://dspace.mit.edu/handle/1721.1/77133

[Lange17] Lange, M., Luporini, F., Louboutin, M., Kukreja, N., Pandolfo, V., Kazakas, P., Velesko, P., Zhang, S., Peng, P., and Gorman, G. Dylan McCormick. 2017, June 7. opesci/devito: Devito-3.0.1. Zenodo. https://doi.org/10.5281/zenodo.823172

[LeVeque92] LeVeque, R. J., "Numerical Methods for Conservation Laws", Birkhauser-Verlag (1992).

[Liu09] Y. Liu and M. K. Sen, “Advanced Finite-Difference Method for Seismic Modeling,” Geohorizons, Vol. 14, No. 2, 2009, pp. 5-16.

[Logg12] Logg, A., Mardal, K.-A., Wells, G. N., et al.: Automated Solution of Differential Equations by the Finite Element Method, Springer, doi:10.1007/978-3-642-23099-8, 2012.

[Louboutin17a] Louboutin, M., Lange, M., Herrmann, F. J., Kukreja, N., and Gorman, G.: Performance prediction of finite-difference solvers for different computer architectures, Computers Geosciences, 105, 148—157, https://doi.org/10.1016/j.cageo.2017.04.014, 2017.

[Louboutin17b] M. Louboutin, M. Lange, F. Luporini, N. Kukreja, F. Herrmann, P. Velesko, and G. Gorman: Code generation from symbolic finite-difference for geophysical exploration. In preparation for Geoscientific Model Development (GMD), 2017.

[McCalpin95] McCalpin, J. D., "Memory Bandwidth and Machine Balance in Current High Performance Computers", IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter, December 1995.

[Meurer17] Meurer A, Smith CP, Paprocki M, Čertík O, Kirpichev SB, Rocklin M, Kumar A, Ivanov S, Moore JK, Singh S, Rathnayake T, Vig S, Granger BE, Muller RP, Bonazzi F, Gupta H, Vats S, Johansson F, Pedregosa F, Curry MJ, Terrel AR, Roučka Š, Saboo A, Fernando I, Kulal S, Cimrman R, Scopatz A. (2017) SymPy: symbolic computing in Python. PeerJ Computer Science 3:e103 https://doi.org/10.7717/peerj-cs.103

[Peiro05] J. Peiró, S. Sherwin, "Finite Difference, Finite Element and Finite Volume Methods for Partial Differential Equations", in "Handbook of Materials Modeling, pp. 2415—2446, ISBN 978-1-4020-3286-8, 2005. http://dx.doi.org/10.1007/978-1-4020-3286-8_127.

[Rai91] M. M. Rai and P. Moin. 1991. "Direct simulations of turbulent flow using finite-difference schemes", J. Comput. Phys. 96, 1 (October 1991), 15-53. http://dx.doi.org/10.1016/0021-9991(91)90264-L

[Rathgeber16] Rathgeber, F., Ham, D. A., Mitchell, L., Lange, M., Luporini, F., McRae, A. T. T., Bercea, G., Markall, G. R., and Kelly, P. H. J.: "Firedrake: automating the finite element method by composing abstractions", ACM Trans. Math. Softw., 43(3):24:1–24:27, 2016. http://dx.doi.org/10.1145/2998441.

[Umetani85] Umetani, Y.: DEQSOL A numerical Simulation Language for Vector/Parallel Processors, Proc. IFIP TC2/WG22, 1985, 5, 147–164, 1985.

[VanEngelen96] R. Van Engelen, L. Wolters, and G. Cats, “Ctadel: A generator of multi-platform high performance codes for pde-based scientific applications,” in Proceedings of the 10th international conference on Supercomputing. ACM, 1996, pp. 86–93.

[Virieux09] Virieux, J. and Operto, S., "An overview of full-waveform inversion in exploration geophysics", GEOPHYSICS, 74, WCC1–WCC26, 2009. http://dx.doi.org/10.1190/1.3238367

[Yount15] C. Yount, "Vector Folding: Improving Stencil Performance via Multi-dimensional SIMD-vector Representation," 2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems, New York, NY, 2015, pp. 865-870. https://doi.org/10.1109/HPCC-CSS-ICESS.2015.27

[Yount16] C. Yount, J. Tobin, A. Breuer and A. Duran, "YASK — Yet Another Stencil Kernel: A Framework for HPC Stencil Code-Generation and Tuning," 2016 Sixth International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing (WOLFHPC), Salt Lake City, UT, 2016, pp. 30-39. https://doi.org/10.1109/WOLFHPC.2016.08

[Zhang12] Y. Zhang and F. Mueller, “Auto-generation and auto-tuning of 3d stencil codes on gpu clusters,” in Proceedings of the Tenth International Symposium on Code Generation and Optimization, ACM, 2012, pp. 155–164. http://doi.acm.org/10.1145/2259016.2259037

Python meets systems neuroscience: affordable, scalable and open-source electrophysiology in awake, behaving rodents

Narendra Mukherjee[¶*], Joseph Wachutka[¶], Donald B Katz[‡§]

Abstract—In-vivo electrophysiology, the recording of neurons in the brains of awake, behaving animals, is currently undergoing paradigm shifts. There is a push towards moving to open-source technologies that can: 1) be adjusted to specific experiments; 2) be shared with ease; and 3) more affordably record from larger numbers of electrodes simultaneously. Here we describe our construction of a system that satisfies these three desirable properties using the scientific Python stack and Linux. Using a Raspberry Pi to control experimental paradigms, we build a completely open-source, HDF5-based analysis (spike sorting) toolkit in Python. This toolkit can be easily parallelized and scales to incorporate increasing electrode counts and longer recordings. Our rig costs about $5000, an order of magnitude less than many comparable commercially available electrophysiology systems.

Index Terms—in-vivo electrophysiology, Python, open-source, HDF5, spike sorting

Introduction

The process of recording neural activity in awake, behaving animals (in-vivo extracellular electrophysiology, hereafter 'ephys') is key in systems neuroscience to understanding how the brain drives complex behaviors. Typically, this process involves voltage recordings from bundles of microwire electrodes (10-20 microns in diameter) surgically implanted into the brain regions of interest. Across hundreds of papers, ephys has increased our understanding of brain systems, function and behavior in a wide range of animal species from invertebrates (locusts and grasshoppers – [SJL03] [BHS15]) to fishes [CAK+16], birds [LIMP+16], rodents [JFS+07] and primates [GMHL05]. Ephys in awake, behaving animals provides an unprecedented view of the complex and highly variable neural dynamics that underlie accurate behavioral responses. It provides a unique degree of resolution at both the spatial and temporal (sub-millisecond) scales, yielding insights into brain structure and function ranging from the cellular [HB01] to the systems [HRB14] [GTJ99] levels.

* *Corresponding author: narendra@brandeis.edu*
¶ *Graduate Program in Neuroscience, Brandeis University, Waltham, MA, USA*
‡ *Department of Psychology, Brandeis University, Waltham, MA, USA*
§ *Volen National Center for Complex Systems, Brandeis University, Waltham, MA, USA*

The world of ephys hardware and software has classically been dominated by proprietary and closed-source technologies. These closed-source designs are, by default, not easily modifiable to suit specific experimental circumstances, and, like any closed-source technology, go against the philosophy of open science [SHNV15]. It is also harder for other investigators to replicate experimental results obtained by the use of such proprietary software, given that most calculations and operations happen *under-the-hood*, with underlying algorithms either being opaque or not technically accessible to researchers of all skill levels [IHGC12]. Furthermore, proprietary ephys hardware and software is prohibitively expensive, and poses a high 'barrier to entry' for neuroscientists starting to set up their laboratories or working under tight budgetary constraints in smaller schools — particularly those in nations in which research funding is scarce. Finally, the use of closed-source technologies has ensured that ephys hardware and software packages are slow to change. In fact, dominant ephys technology has been virtually unchanged for the last 20 years despite the fact that electronics and computing technology have taken giant strides forward in that time.

With reproducible and affordable science in mind, some ephys laboratories have recently started to explore open source ephys hardware and software [SHNV15]. The possible value of this move is manifold: new ephys hardware and software, apart from being open-source, affordable and reproducible, can easily 'scale' with growing experiment and data sizes. It is, therefore, much easier with open-source technology to follow the trend in modern ephys towards increasing 'channel counts' - recording from hundreds, or even thousands, of electrodes implanted in several different brain regions to better understand the inter-regional coordination that underlies brain function and animal behavior.

In this paper, we describe a completely open-source, Python-based hardware and software setup that we are currently using to study the role of gustatory (taste) cortex in taste-related learning and behavior in rats. We use a Raspberry Pi based system to coordinate the various stimulus control needs of our experiments. This includes the delivery of precise amounts of taste solutions to the animals [KSN02] and the optogenetic perturbation of the firing of neurons in the taste cortex with laser sources [LMRK16] [Pas11]. To handle the ephys signals, we use chips from Intan Technologies and a HDF5 and Python-based software setup for spike sorting (picking out action potentials from individual neurons) [Lew98] and analysis.

Starting with a brief description of the hardware we have constructed to control experimental paradigms, we will focus on describing the computations involved at every step of our spike sorting toolchain, highlighting software principles that make such an analysis setup: 1) scale with increased channel counts and longer recordings; and 2) easily parallelized on computing environments. Traditionally, manual approaches, closed-source software and heuristics abound in the electrophysiologist's spike sorting toolchain - these are time-consuming, error-prone and hard to replicate in a principled manner [WBVI+04]. We automate several key steps of the spike sorting pipeline with algorithms that have been suggested elsewhere [QNBS04] [FMK96] and describe the accessibility and ease-of-use that the scientific Python stack offers to electrophysiologists. Finally, we demonstrate the use of this system to record and analyze ephys data from 64 electrodes simultaneously in the taste cortex of rodents and point out future directions of improvement keeping the modern ephys experiment in mind.

Animal care, handling and surgeries

We use adult, female Long-Evans rats (300-325g) and adult mice (15-20g) in our experiments. They are prepared with surgically implanted bundles of microwire electrodes bilaterally in the gustatory (taste) cortex and intra-oral cannulae (IOCs) behind the cheek for delivering taste solutions. All animal care and experiments comply with the Brandeis University Institutional Animal Care and Use Committee (IACUC) guidelines. For more details on experimental protocols, see [SMV+16].

Raspberry Pi based behavior control system

We use a Raspberry Pi running Ubuntu-MATE to weave together the various behavioral paradigms of our experiments. This includes 1) delivering precise amounts of taste solutions to the animals via pressurized solenoid valves, 2) measuring the animals' licking responses with an analog-to-digital converter (ADC) circuit and 3) controlling laser sources for optogenetic perturbation. Most of these steps involve controlling the digital I/O pins (DIO) of the Pi – the Rpi.GPIO package provides convenient functions:

```
import RPi.GPIO as GPIO
# The BOARD mode allows referring to the GPIO pins
# by their number on the board
GPIO.setmode(GPIO.BOARD)
# Set port 1 as an output
GPIO.setup(1, GPIO.OUT)
# Send outputs to port 1
GPIO.output(1, 1)
GPIO.output(1, 0)
```

Electrode bundles and microdrives

We build electrode bundles with 32 nichrome-formvar microwires (0.0015 inch diameter, from a-msystems), a 200 μ fiber for optogenetics (optionally), and 3D printed microdrives. Our custom built drives cost about $50 and their designs are freely available for use and modification at the Katz lab website.

Electrophysiology hardware

We use an open-source ephys recording system from Intan Technologies for neural recordings. The RHD2000 series ephys recording headstages connect to electrode bundles implanted in the animal's brain and contain 32-128 amplifiers and ADCs. The Intan data acquisition system offers an open-source C++ based graphical interface that can record up to 512 electrodes (4 headstages) simultaneously at sampling rates of up to 30kHz/channel. This recording system is relatively robust to AC noise, because the electrode signals are digitized right on the headstage itself, but we additionally encase the animal's behavior and recording chamber in a Faraday cage constructed with standard aluminum insect netting.

Electrophysiology in systems neuroscience

In-vivo ephys is unique in systems neuroscience in the temporal and spatial view it provides into the role of the brain in generating accurate behavioral responses. Ephys typically involves the placement of a bundle [SMV+16] or spatially structured array [WRL+15] of electrodes in a brain region of interest. After the animal recovers from the surgical implantation of electrodes, its behavior in tightly controlled experimental paradigms is correlated with neural activity in the brain region being recorded from. The study of sensory systems (vision, somatosensation, olfaction, taste, etc) in the brain, for instance, involves an awake, behaving animal experiencing different sensory stimuli while ephys recordings are performed in the corresponding sensory cortex (or other involved regions). In addition, ephys electrodes are often implanted in multiple brain regions in the same animal in order to understand the role of inter-regional coordination in the animal's behavior.

In our lab, we study taste processing in adult mice and rats - Figure 1 shows a typical experimental setup. We surgically implant bundles of 64 microwire electrodes bilaterally (32 wires in each hemisphere) in the taste cortex (among many other regions). Our basic experimental paradigm involves the animal tasting solutions of different kinds (sweet - sucrose, salty - NaCl or bitter - quinine, for instance) while its behavioral responses to the tastes are being recorded [LMRK16]. All this while, we record electrical activity in the taste cortex using the implanted electrodes and eventually try to understand the animals behavior in the light of the activity of the neurons being recorded from.

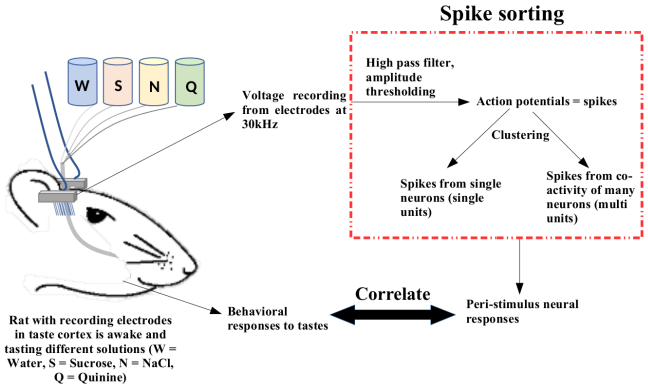


Fig. 1: *An example of a sensory systems experimental setup. The animal (rodent, primate, etc) experiences sensory stimuli (taste, in this case) while cortical (or other) neurons are being recorded. Eventually, the activity of the recorded population of neurons (also called units) is analyzed in the context of the animal's behavioral responses.*

The essential step in the analysis of ephys data, therefore, is to isolate (and identify) the activity of single neurons from the raw voltage recordings from the implanted electrodes. As shown in Figure 1, this involves high-pass filtering of the raw voltage

signals (see next section for more details) to identify putative action potentials (or '*spikes*'). These spikes can originate either from a single neuron or multiple neurons. We thus need to sort them into groups, based on how they are inferred to originate (spikes inferred to be from single neurons are called '*single units*' and those from multiple neurons are called '*multi units*'). This entire pipeline is, therefore, called '*spike sorting*'. Typically, we are able to isolate 10-40 neurons from our recordings with 64 electrodes - we then go on to correlate the responses of this population of recorded units with the animal's behavior in our experimental paradigms (see [SMV+16], [LMRK16] as examples, and Figure 1).

Scientific Python stack for data analysis – spike sorting

The recent push in ephys experiments towards increased channel counts and longer recordings poses significant data handling and analysis challenges. Each of the implanted electrodes needs to be sampled at frequencies in the range of 20-30kHz if it is to clearly render action potentials (the signature binary voltage waveforms, about 1ms in duration, that neurons produce when active – also called '*spikes*', hence the name '*spike sorting*'). In our experiments, we sample signals coming from 64 electrodes at 30kHz for up to 2 hours, generating datasets that total 10-30GB in size. Datasets of such sizes cannot be loaded into memory and processed in serial – there is evidently a need to convert the data to a format that allows access to specific parts of the data and can support a parallel computing framework.

The Hierarchical Data Format (HDF5) is ideal for dealing with such big numerical datasets. We use the Pytables package to build, structure and modify HDF5 files at every point in our spike sorting and analysis toolchain. Pytables allows data to be stored and extracted from HDF5 files in the convenient form of numpy arrays. We decided to use individual electrodes as storage and computation splits, storing the voltage recording from each electrode as a separate array in the HDF5 file with its analysis assigned to a separate process.

We adopt a semi-supervised approach to spike sorting, starting with a (parallelized) set of automated filtering and clustering steps that can be fine-tuned by the experimenter (who presumably comes equipped with expert knowledge about action potential shapes actually observed in the brain). Our setup therefore involves 3 distinct steps (all the code is available on Github):

1) Pre-processing (**blech_clust.py**) – Constructs a HDF5 file post-experiment with the raw binary data recorded by the Intan system, acquires the clustering parameters from the user and creates a shell file that runs the actual processing step in parallel.
2) Processing (**blech_process.py**) – Runs filtering and clustering steps on the voltage data from every electrode and plots out the results.
3) Post-processing (**blech_post_process.py**) – Removes raw recordings from the HDF5 file and compresses it, and then allows the user to sieve out real spikes from the putative spikes plotted in step 2.

Pre-processing

The pre-processing starts by building a HDF5 file for the ephys dataset with separate nodes for raw neural electrodes, digital inputs and outputs. This structuring of different aspects of the data into separate nodes is a recurrent feature of our toolchain. The Pytables library provides a convenient set of functions for this purpose:

```
# modified from blech_clust.py
import tables
# Create hdf5 file, and make group for raw data
hf5 = tables.open_file(hdf5_name[-1]+'.h5', 'w',
        title = hdf5_name[-1])
# Node for raw electrode data
hf5.create_group('/', 'raw')
# Node for digital inputs
hf5.create_group('/', 'digital_in')
#Node for digital outputs
hf5.create_group('/', 'digital_out')
hf5.close()
```

We have set up Pytables *extendable arrays* (EArrays) to read the electrode and digital input data saved by the Intan system. Extendable arrays are akin to standard Python lists in the sense that their size can be 'extended' as data is appended to them – unlike lists, however, they are a homogeneous data class and cannot store different types together. The Intan system saves all the data as integers in binary files and therefore, EArrays of type int (defined by IntAtom in Pytables) are perfect for this purpose. These EArrays can be constructed and filled as follows:

```
# Modified from create_hdf_arrays() in read_file.py
# Open HDF5 file with read and write permissions - r+
hf5 = tables.open_file(file_name, 'r+')
# 2 ports/headstages each with 32
# electrodes in our experiments
n_electrodes = len(ports)*32
# All the data is stored as integers
atom = tables.IntAtom()
# Create arrays for neural electrodes
for i in range(n_electrodes):
    el = hf5.create_earray('/raw',
                           'electrode%i' % i,
                            atom, (0,))
hf5.close()

# Modified from read_files() in read_file.py
# Open HDF5 file with read and write permissions - r+
hf5 = tables.open_file(file_name, 'r+')
# Fill data from electrode 1 on port A
# Electrode data are stored in binary files
# as 16 bit signed integers
# Filenames of binary files as defined
# by the Intan system
data = np.fromfile('amp-A-001.dat',
                   dtype = np.dtype('int16'))
hf5.flush()
hf5.close()
```

To facilitate the spike sorting process, we use the easygui package to integrate user inputs through a simple graphical interface. Finally, we use GNU Parallel [Tan11] to run filtering and clustering on every electrode in the dataset in a separate process. GNU Parallel is a great parallelization tool on .nix systems, and allows us to: 1) assign a minimum amount of RAM to every process and 2) resume failed processes by reading from a log file.

Processing

The voltage data from the electrodes are stored as signed integers in the HDF5 file in the pre-processing step – they need to be converted into actual voltage values (in microvolts) as floats. The datasheet of the Intan RHD2000 system gives the transformation as:

$$voltage(\mu V) = 0.195 * voltage(int)$$

Spikes are high frequency events that typically last for 1-1.5 ms – we therefore remove low frequency transients by bandpass

filtering the data in 300-3000 Hz using a 2-pole Butterworth filter as follows:

```
# Modified from get_filtered_electrode()
# in clustering.py
from scipy.signal import butter
from scipy.signal import filtfilt
m, n = butter(2, [300.0/(sampling_rate/2.0),
              3000.0/(sampling_rate/2.0)],
              btype = 'bandpass')
filt_el = filtfilt(m, n, el)
```

Depending on the position of the electrode in relation to neurons in the brain, action potentials appear as transiently large positive or negative deflections from the mean voltage detected on the electrode. Spike sorting toolchains thus typically impose an amplitude threshold on the voltage data to detect spikes. In our case (i.e., cortical neurons recorded extracellularly with microwire electrodes), the wide swath of action potentials appear as negative voltage deflections from the average – we therefore need to choose segments of the recording that go *below* a predefined threshold. The threshold we define is based on the median of the electrode's absolute voltage (for details, see [QNBS04]):

```
# Modified from extract_waveforms() in clustering.py
m = np.mean(filt_el)
th = 5.0*np.median(np.abs(filt_el)/0.6745)
pos = np.where(filt_el <= m - th)[0]
```

We treat each of these segments as a '*putative spike*'. We locate the minimum of each segment and slice out 1.5ms (0.5ms before the minimum, 1ms after = 45 samples at 30kHz) of data around it. These segments, having been recorded digitally, are eventually approximations of the actual analog signal with repeated samples. Even at the relatively high sampling rates that we use in our experiments, it is possible that these segments are significantly 'jittered' in time and their shapes do not line up exactly at their minima due to sampling approximation. In addition, due to a variety of electrical noise that seeps into such a recording, we pick up a large number of segments that have multiple troughs (or minima) and are unlikely to be action potentials. To deal with these issues, we 'dejitter' the set of potential spikes by interpolating their shapes (using scipy.interpolate.interp1d), up-sampling them 10-fold using the interpolation, and finally picking just the segments that can be lined up by their unique minimum.

This set of 450-dimensional putative spikes now needs to be sorted into two main groups: one that consists of actual action potentials recorded extracellularly and the other that consists of noise (this is high-frequency noise that slips in despite the filtering and amplitude thresholding steps). In addition, an electrode can record action potentials from multiple neurons - the group consisting of real spikes, therefore, needs to be further sorted into one or more groups depending upon the number of neurons that were recorded on the electrode. We start this process by first splitting up the set of putative spikes into several *clusters* by fitting a Gaussian Mixture Model (GMM) [Lew98]. GMM is an unsupervised clustering technique that assumes that the data originate from several different groups, each defined by a Gaussian distribution (in our case over the 450 dimensions of the putative spikes). Classifying the clusters that the GMM picks as noise or real spikes is eventually a subjective decision (explained in the post-processing section). The user picks the best solution with their expert knowledge in the manual part of our semi-automated spike sorting toolchain (which is potentially time cosuming for recordings with large numbers of electrodes, see *Discussion* for more details).

Each putative spike waveform picked by the procedure above consists of 450 samples after interpolation – there can be more than a million such waveforms in a 2 hour recording from each electrode. Fitting a GMM in such a high dimensional space is both processor time and memory consuming (and can potentially run into the curse-of-dimensionality). We therefore reduce the dimensionality of the dataset by picking the first 3 components produced through principal component analysis (PCA) [BS14] using the scikit-learn package [PVG+11]. These principal components, however, are known to depend mostly on the amplitude-induced variance in shapes of recorded action potential waveforms – to address this possibility, we scale each waveform by its energy (modified from [FMK96]), defined as follows, before performing the PCA:

$$Energy = \frac{1}{n}\sqrt{\sum_{i=1}^{450} X_i^2}$$

where $X_i = i^{th}$ component of the waveform

Finally, we feed in the energy and maximal amplitude of each waveform as features into the GMM in addition to the first 3 principal components. Using scikit-learn's GMM API, we fit GMMs with cluster numbers varying from 2 to a user-specified maximum number (usually 7 or 8). Each of these models is fit to the data several times (usually 10) and the best fit is chosen according to the Bayesian Information Criterion (BIC) [BK10].

The clustering results need to be plotted for the user to be able to pick action potentials from the noise in the post-processing step. The most important in these sets of plots are the actual waveforms of the spikes clustered together by the GMM and the distribution of their inter-spike-intervals (ISIs) (more details in the post-processing step). Plotting the waveforms of the putative spikes in every cluster produced by the GMM together, however, is the most memory-expensive step of our toolchain. Each putative spike is 1.5ms (or 45 samples) long, and there can be tens of thousands of spikes in every cluster (see Figures 2, 3). For a 2 hour recording with 64 electrodes, the plotting step with matplotlib [Hun07] can consume up to 6GB of memory although the PNG files that are saved to disk are only of the order of 100KB. High memory consumption during plotting also limits the possibility of applying this spike sorting framework to recordings that are several hours long – as a potential substitute, we have preliminarily set up a live plotting toolchain using Bokeh that can be used during the post-processing step. We are currently trying to work out a more memory-efficient plotting framework, and any suggestions to that end are welcome.

Post-processing

Once the parallelized processing step outlined above is over, we start the post-processing step by first deleting the raw electrode recordings (under the 'raw' node) and compressing the HDF5 file using ptrepack as follows:

```
# Modified from blech_post_process.py
hf5.remove_node('/raw', recursive = True)
# Use ptrepack with compression level = 9 and
# compression library = blosc
os.system("ptrepack --chunkshape=auto --propindexes
          --complevel=9 --complib=blosc " + hdf5_name
          + " " + hdf5_name[:-3] + "_repacked.h5")
```

The logic of the post-processing step revolves around allowing the user to look at the GMM solutions for the putative spikes from every electrode, pick the solution that best splits the noise and

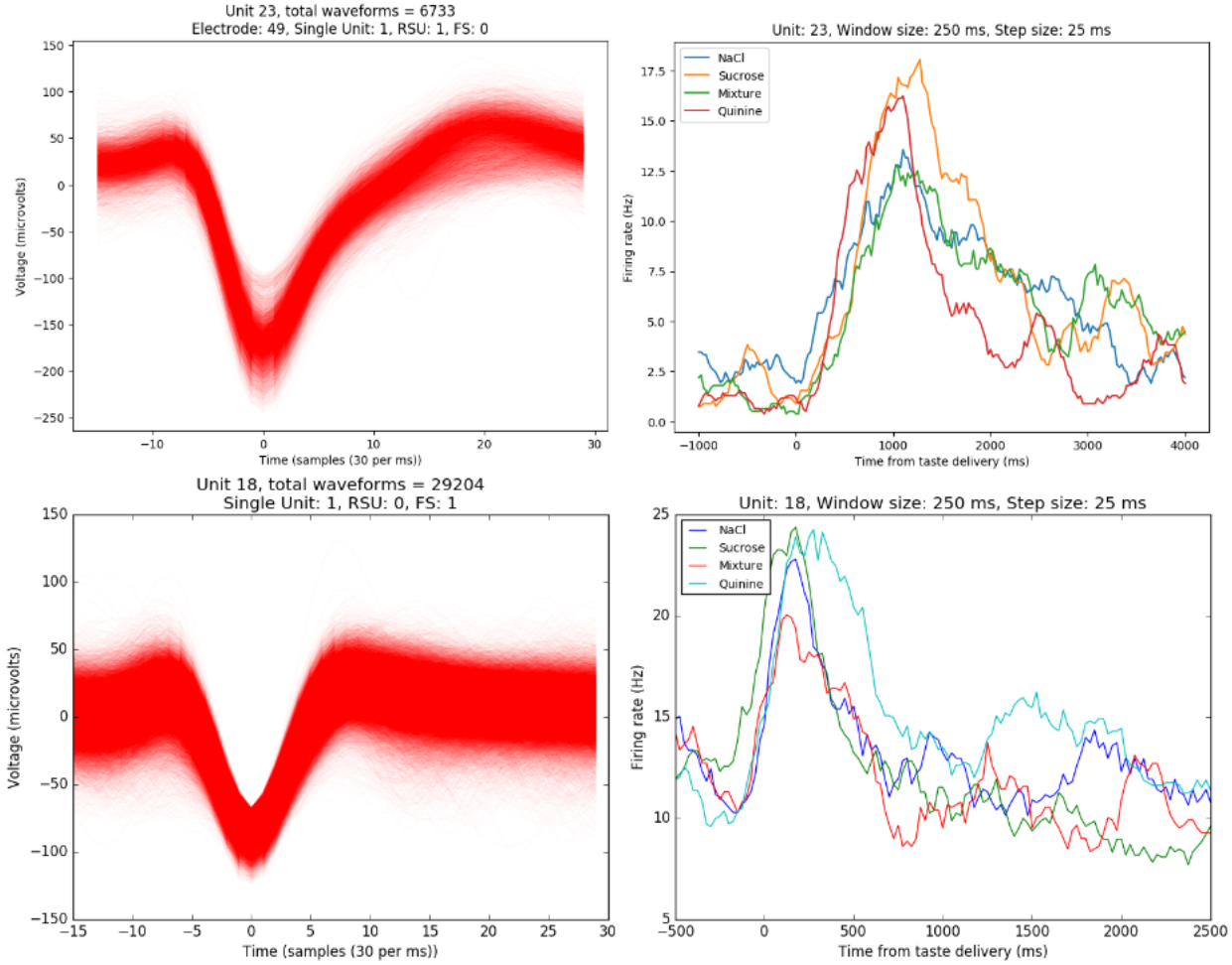


Fig. 2: *Two types of single units isolated from taste cortex recordings. Spike waveforms on the left, and responses to the taste stimuli on the right.* ***Top-left:*** *Spikes waveforms of a regular spiking unit (RSU) - 45 samples (1.5ms) on the time/x axis. Note the 2 inflection points as the spikes go back to baseline from their minimum - this is characteristic of the shape of RSUs. RSUs represent the activity of excitatory cortical pyramidal neurons on ephys records - these spikes are slow and take about 1ms (20-30 samples) to go back up to baseline from their minimum (with 2 inflection points).* ***Bottom-left:*** *Spike waveforms of a fast spiking unit (FS) - 45 samples (1.5ms) on the time/x axis. Compare to the spike waveforms of the RSU in the top-left figure and note that this unit has narrower/faster spikes that take only 5-10 samples (1/3 ms) to go back up to baseline from their minimum. FSs represent the activity of (usually inhibitory) cortical interneurons on ephys records.* ***Top-Right:*** *Peri-stimulus time histogram (PSTH) - Plot of the activity of the RSU around the time of stimulus (taste) delivery (0 on the time/x axis). Note the dramatic increase in firing rate (spikes/second) that follows taste delivery.* ***Bottom-Right:*** *Peri-stimulus time histogram (PSTH) - Plot of the activity of the FS around the time of stimulus (taste) delivery (0 on the time/x axis). Note the dramatic increase in firing rate (spikes/second) that follows taste delivery. Also compare to the PSTH of the RSU in the figure above and note that the FS has a higher firing rate (more spikes) than the RSU. 0.1M Sodium Chloride (NaCl), 0.15M Sucrose, 1mM Quinine-HCl and a 50:50 mixture of 0.1M NaCl and 0.15M Sucrose were used as the taste stimuli.*

spike clusters, and choose the cluster numbers that corresponds to spikes. The GMM clustering step, being unsupervised in nature, can sometimes put spikes from two (or more) separate neurons (with very similar energy-scaled shapes, but different amplitudes) in the same cluster or split the spikes from a single neuron across several clusters. In addition, the actual action potential waveform observed on an electrode depends on the timing of the activity of the neurons in its vicinity – co-active neurons near an electrode can additively produce spike waveforms that have smaller amplitude and are noisier (called 'multi' units) (Figure 3) than single, isolated neurons (called 'single' units, Figure 2). Therefore, we set up utilities to merge and split clusters in the post-processing step – users can choose to merge clusters when the spikes from a single neuron have been distributed across clusters or split (with a GMM clustering using the same features as in the processing step) a single cluster if it contains spikes from separate neurons.

HDF5, once again, provides a convenient format to store the single and multi units that the user picks from the GMM results. We make a 'sorted_units' node in the file to which units are added in the order that they are picked by the user. In addition, we make a 'unit_descriptor' table that contains metadata about the units that are picked – these metadata are essential in all downstream analyses of the activity of the neurons in the dataset. To set up such a table through Pytables, we first need to create a class describing the datatypes that the columns of the table will hold and then use this class as the description while creating the table.

```
# Modified from blech_post_process.py
# Define a unit_descriptor class to be used
# to add things (anything!) about the sorted
```

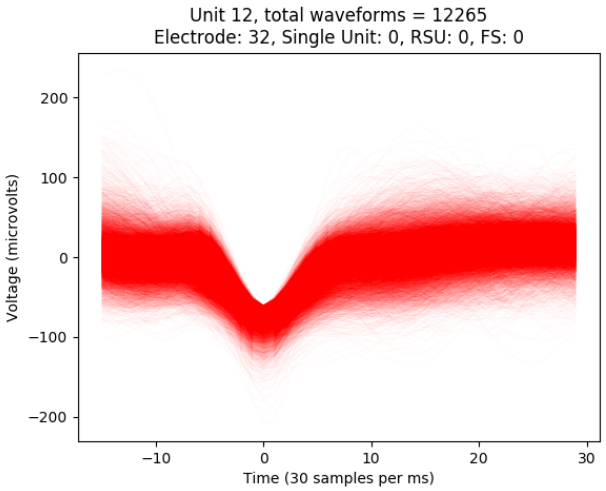


Fig. 3: *A multi unit - 45 samples (1.5ms) on the time/x axis. Compare to the single units in Figure 2 and note that these spikes have smaller amplitudes and are noisier. Multi units are produced by the co-activity of multiple neurons near the electrode.*

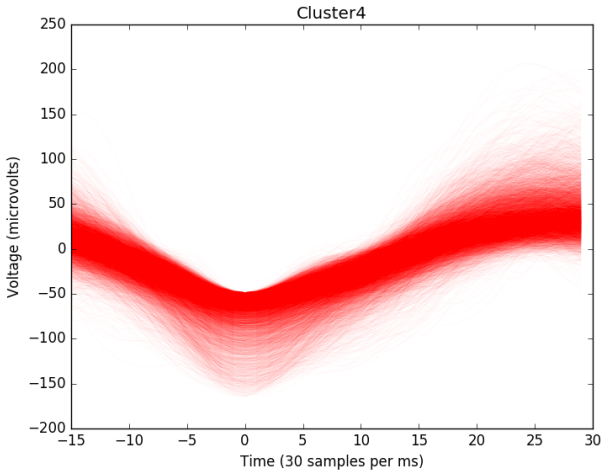


Fig. 4: *A noise cluster - 45 samples (1.5ms) on the time/x axis. This is high frequency noise that seeps in despite the filtering and thresholding steps used in the processing step. Compare to the single units in Figure 2 and multi unit in Figure 3 and note that these waveforms are much smoother and do not have the characteristics of a unit.*

```
# units to a pytables table
class UnitDescriptor(tables.IsDescription):
    electrode_number = tables.Int32Col()
    single_unit = tables.Int32Col()
    regular_spiking = tables.Int32Col()
    fast_spiking = tables.Int32Col()

# Make a table describing the sorted units.
# If unit_descriptor already exists, just open it up
try:
    table = hf5.create_table('/', 'unit_descriptor',
                             description = UnitDescriptor)
except Exception:
    table = hf5.root.unit_descriptor
```

Cortical neurons (including gustatory cortical neurons that we record from in our experiments) fall into two major categories – 1) excitatory pyramidal cells that define cortical layers and have long range connections across brain regions, and 2) inhibitory interneurons that have short range connections. In ephys records, pyramidal cells produce relatively large and slow action potentials at rates ranging from 5-20 Hz (spikes/s) (Figure 2, top). Interneurons, on the other hand, have much higher spiking rates (usually from 25-50Hz, and sometimes up to 70 Hz) and much faster (and hence, narrower) action potentials (Figure 2, bottom). Therefore, in the unit_descriptor table, we save the type of cortical neuron that the unit corresponds to in addition to the electrode number it was located on and whether its a single unit. In keeping with classical ephys terminology, we refer to putative pyramidal neuron units as 'regular spiking units (RSU)' and interneuron units as 'fast spiking units (FS)' [MCLP85] [HLVH+13]. In addition, anatomically, pyramidal cells are much larger and more abundant than interneurons in cortical regions [YEH11] [AFY+13] [PTF+17] – expectedly, in a typical gustatory cortex recording, 60-70% of the units we isolate are RSUs. This classification of units is in no way restrictive – new descriptions can simply be added to the UnitDescriptor class to account for recordings in a sub-cortical region that contains a different electrophysiological unit.

Apart from the shape of the spikes (look at Figures 2, 3, 4 to compare spikes and typical noise) in a cluster, the distribution of their inter-spike-intervals (ISIs) (plotted in the processing step) is another important factor in differentiating single units from multi units or noise. Due to electrochemical constraints, after every action potential, neurons enter a '*refractory period*' - most neurons cannot produce another spike for about 2ms. We, therefore, advise a relatively conservative ISI threshold while classifying single units – in our recordings, we designate a cluster as a single unit only if <0.01% (<1 in 10000) spikes fall within 2ms of another spike.

Finally, we consider the possibility that since the processing of the voltage data from each electrode happens independently in a parallelized manner, we might pick up action potentials from the same neuron on different electrodes (if they are positioned close to each other). We, therefore, calculate '*similarity*' between every pair of units in the dataset – this is the percentage of spikes in a unit that are within 1ms of spikes in a different unit. This metric should ideally be very close to 0 for two distinct neurons that are spiking independently – in our datasets, we consider units that have similarity greater than 20% as the same neuron and discard one of them from our downstream analysis. To speed up this analysis, especially for datasets that have 20-40 neurons each with >10000 spikes, we use Numba's just-in-time compilation (JIT) feature:

```
# Modified from blech_units_distance.py
from numba import jit
@jit(nogil = True)
def unit_distance(this_unit_times, other_unit_times):
    this_unit_counter = 0
    other_unit_counter = 0
    for i in range(len(this_unit_times)):
            for j in range(len(other_unit_times)):
                    if np.abs(this_unit_times[i]
                              - other_unit_times[j])
                              <= 1.0:
                             this_unit_counter += 1
                             other_unit_counter += 1
    return this_unit_counter, other_unit_counter
```

Discussion

In-vivo extracellular electrophysiology in awake, behaving animals provides a unique spatiotemporal glimpse into the activity

of populations of neurons in the brain that underlie the animals' behavioral responses to complex stimuli. Recording, detecting, analyzing and isolating action potentials of single neurons in a brain region in an awake animal poses a variety of technical challenges, both at the hardware and software levels. Rodent and primate electrophysiologists have classically used proprietary hardware and software solutions in their experiments – these closed-source technologies are expensive, not suited to specific experimental contexts and hard to adapt to sharing and collaboration. The push towards open, collaborative and reproducible science has spurred calls for affordable, scalable open-source experimental setups. In this paper, we have outlined a Raspberry Pi and scientific Python-based solution to these technical challenges and described its successful use in electrophysiological and optogenetic experiments in the taste cortex of awake mice and rats. Our setup can scale as data sizes grow with increasingly longer recordings and larger number of electrodes, and costs ~$5000 (compared to up to $100k for a comparable proprietary setup).

Our approach uses the HDF5 data format, which allows us to organize all of the data (and their associated metadata) under specific nodes in the same file. This approach has several advantages over traditional practices of organizing ephys data. Firstly, HDF5 is a widely used cross-platform data format that has convenient APIs in all major programming languages. Secondly, having all the data from an experimental session in the same file (that can be easily compressed – we use ptrepack in the post-processing step) makes data sharing and collaboration easier. Thirdly, HDF5 files allow quick access to desired parts of the data during analysis – as a consequence, larger than memory workflows can easily be supported without worrying about the I/O overhead involved. Lastly, in our setup, we splice the storage and processing of the data by individual electrodes – this allows us to run the processing step in parallel on several electrodes together bringing down processing time significantly.

The standard approach of picking units in ephys studies involves arbitrary, user-defined amplitude threshold on spike waveforms during ephys recordings and manually drawing polygons around spikes from a putative unit in principal component (PC) space. This process is very time consuming for the experimenter and is prone to human errors. Our semi-automated approach to spike sorting is faster and more principled than the standard approach - we automate both these steps of the traditional spike sorting toolchain by using an amplitude threshold that depends on the median voltage recorded on an electrode and clustering putative spikes with a Gaussian Mixture Model (GMM). The user's expertise only enters the process in the last step of our workflow — they label the clusters picked out by the GMM as noise, single unit or multi unit based on the shapes of the spike waveforms and their ISI distributions. As the number of electrodes in an electrophysiological recording is already starting to run into the hundreds and thousands, there is a need to automate this last manual step as well – this can be achieved by fitting supervised classifiers to the units (and their types) picked out manually in a few training datasets. As the waveforms of spikes can depend upon the brain region being recorded from, such an approach would likely have to applied to every brain region separately.

During the pre-processing step, we restrict our setup to pick only *negative* spikes – those in which the voltage deflection goes *below* a certain threshold. While most extracellular spikes will appear as negative voltage deflections (due to the fact that they are being mostly recorded from outside the axons of neurons), sometimes an electrode, depending on the brain region, ends up being close enough to the cell body of a neuron to record positive spikes. Our pre-processing step requires only trivial modifications to include positive deflections '*above*' a threshold as spikes as well.

The use of the HDF5 format and the ease of supporting larger-than-memory workflows allows our toolchain to scale to longer recordings and increased electrode counts. However, as explained previously, plotting all the spike waveforms in a cluster together during the processing step using matplotlib is a major memory bottleneck in our workflow. We are working on still more efficient workarounds, and have devised a live plotting setup with Bokeh (that plots 50 waveforms at a time) that can be used during post processing instead. In addition, recordings running for several hours (or days) have to account for the change in spike waveforms induced by '*electrode drift*' - the electrode moves around in the fluid medium of the brain with time. The live plotting module is potentially useful in such longer recordings as well – it can be used to look at spikes recorded in small windows of time (30 minutes say) to see if their shapes change with time.

We are currently attempting to fold our Python based ephys analysis setup into the format of a Python package that can be used by electrophysiologists (using the Intan recording system) to analyze their data with ease on a shared computing resource or on personal workstations. We think that using the scientific Python stack will make previously hidden *under-the-hood* spike sorting principles clearer to the average electrophysiologist, and will make implementing downstream analyses on these data easier.

Acknowledgements

This work was supported by National Institutes of Health (NIH) grants R01 DC006666-00 and R01 DC007703-06 to DBK. NM was supported by the Howard Hughes Medical Institute (HHMI) International Graduate Fellowship through the duration of this work. The National Science Foundation's (NSF) Extreme Science and Engineering Discovery Environment (XSEDE) supported the computational needs for this work through grant IBN170002 to DBK.

We would like to thank Dr. Francesco Pontiggia for helping us solidify many of our data handling and computing ideas and Dr. Jian-You Lin for being the first independent tester of our toolchain. NM would additionally like to thank Shrabastee Banerjee for providing many hours of insights on statistical and programming ideas and pushing for this work to be written up in the first place.

Declaration of interest

The authors declare no competing financial interests.

References

[AFY+13] Kazunori Adachi, Satoshi Fujita, Atsushi Yoshida, Hiroshi Sakagami, Noriaki Koshikawa, and Masayuki Kobayashi. Anatomical and electrophysiological mechanisms for asymmetrical excitatory propagation in the rat insular cortex: In vivo optical imaging and whole-cell patch-clamp studies. *Journal of Comparative Neurology*, 521(7):1598–1613, 2013.

[BHS15] Mit Balvantray Bhavsar, Ralf Heinrich, and Andreas Stumpner. Multielectrode recordings from auditory neurons in the brain of a small grasshopper. *Journal of Neuroscience Methods*, 256:63 – 73, 2015. URL: http://www.sciencedirect.com/science/article/pii/S0165027015003155, doi:https://doi.org/10.1016/j.jneumeth.2015.08.024.

[BK10] Harish S Bhat and Nitesh Kumar. On the derivation of the bayesian information criterion. *School of Natural Sciences, University of California*, 2010.

[BS14] Rasmus Bro and Age K Smilde. Principal component analysis. *Analytical Methods*, 6(9):2812–2831, 2014.

[CAK+16] Ming-Yi Chou, Ryunosuke Amo, Masae Kinoshita, Bor-Wei Cherng, Hideaki Shimazaki, Masakazu Agetsuma, Toshiyuki Shiraki, Tazu Aoki, Mikako Takahoko, Masako Yamazaki, Shin-ichi Higashijima, and Hitoshi Okamoto. Social conflict resolution regulated by two dorsal habenular subregions in zebrafish. *Science*, 352(6281):87–90, 2016. URL: http://science.sciencemag.org/content/352/6281/87, arXiv:http://science.sciencemag.org/content/352/6281/87.full.pdf, doi:10.1126/science.aac9508.

[FMK96] Michale S. Fee, Partha P. Mitra, and David Kleinfeld. Automatic sorting of multiple unit neuronal signals in the presence of anisotropic and non-gaussian variability. *Journal of Neuroscience Methods*, 69(2):175 – 188, 1996. URL: http://www.sciencedirect.com/science/article/pii/S0165027096000507, doi:https://doi.org/10.1016/S0165-0270(96)00050-7.

[GMHL05] Asif A. Ghazanfar, Joost X. Maier, Kari L. Hoffman, and Nikos K. Logothetis. Multisensory integration of dynamic faces and voices in rhesus monkey auditory cortex. *Journal of Neuroscience*, 25(20):5004–5012, 2005. URL: http://www.jneurosci.org/content/25/20/5004, arXiv:http://www.jneurosci.org/content/25/20/5004.full.pdf, doi:10.1523/JNEUROSCI.0799-05.2005.

[GTJ99] H Gurden, J-P Tassin, and TM Jay. Integrity of the mesocortical dopaminergic system is necessary for complete expression of in vivo hippocampal–prefrontal cortex long-term potentiation. *Neuroscience*, 94(4):1019–1027, 1999.

[HB01] Arnold J. Heynen and Mark F. Bear. Long-term potentiation of thalamocortical transmission in the adult visual cortex in vivo. *Journal of Neuroscience*, 21(24):9801–9813, 2001. URL: http://www.jneurosci.org/content/21/24/9801, arXiv:http://www.jneurosci.org/content/21/24/9801.full.pdf.

[HLVH+13] Keith B Hengen, Mary E Lambo, Stephen D Van Hooser, Donald B Katz, and Gina G Turrigiano. Firing rate homeostasis in visual cortex of freely behaving rodents. *Neuron*, 80(2):335–342, 2013.

[HRB14] MacKenzie Allen Howard, John LR Rubenstein, and Scott C Baraban. Bidirectional homeostatic plasticity induced by interneuron cell death and transplantation in vivo. *Proceedings of the National Academy of Sciences*, 111(1):492–497, 2014.

[Hun07] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing In Science & Engineering*, 9(3):90–95, 2007. doi:10.1109/MCSE.2007.55.

[IHGC12] Darrel C Ince, Leslie Hatton, and John Graham-Cumming. The case for open computer programs. *Nature*, 482(7386):485–488, 2012.

[JFS+07] Lauren M Jones, Alfredo Fontanini, Brian F Sadacca, Paul Miller, and Donald B Katz. Natural stimuli evoke dynamic sequences of states in sensory cortical ensembles. *Proceedings of the National Academy of Sciences*, 104(47):18772–18777, 2007.

[KSN02] Donald B Katz, SA Simon, and Miguel AL Nicolelis. Taste-specific neuronal ensembles in the gustatory cortex of awake rats. *Journal of Neuroscience*, 22(5):1850–1857, 2002.

[Lew98] Michael S Lewicki. A review of methods for spike sorting: the detection and classification of neural action potentials. *Network: Computation in Neural Systems*, 9(4):R53–R78, 1998.

[LIMP+16] William A Liberti III, Jeffrey E Markowitz, L Nathan Perkins, Derek C Liberti, Daniel P Leman, Grigori Guitchounts, Tarciso Velho, Darrell N Kotton, Carlos Lois, and Timothy J Gardner. Unstable neurons underlie a stable learned behavior. *Nature Neuroscience*, 19(12):1665–1671, 2016.

[LMRK16] Jennifer X Li, Joost X Maier, Emily E Reid, and Donald B Katz. Sensory cortical activity is related to the selection of a rhythmic motor action pattern. *Journal of Neuroscience*, 36(20):5596–5607, 2016.

[MCLP85] David A McCormick, Barry W Connors, James W Lighthall, and David A Prince. Comparative electrophysiology of pyramidal and sparsely spiny stellate neurons of the neocortex. *Journal of neurophysiology*, 54(4):782–806, 1985.

[Pas11] Erika Pastrana. Optogenetics: controlling cell function with light. *Nature Methods*, 8(1):24, 2011.

[PTF+17] Yangfan Peng, Barreda Tomás, J Federico, Constantin Klisch, Imre Vida, and Jörg RP Geiger. Layer-specific organization of local excitatory and inhibitory synaptic connectivity in the rat presubiculum. *Cerebral Cortex*, 27(4):2435–2452, 2017.

[PVG+11] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[QNBS04] R Quian Quiroga, Zoltan Nadasdy, and Yoram Ben-Shaul. Unsupervised spike detection and sorting with wavelets and superparamagnetic clustering. *Neural computation*, 16(8):1661–1687, 2004.

[SHNV15] Joshua H Siegle, Gregory J Hale, Jonathan P Newman, and Jakob Voigts. Neural ensemble communities: open-source approaches to hardware for large-scale electrophysiology. *Current opinion in neurobiology*, 32:53–59, 2015.

[SJL03] Mark Stopfer, Vivek Jayaraman, and Gilles Laurent. Intensity versus identity coding in an olfactory system. *Neuron*, 39(6):991 – 1004, 2003. URL: http://www.sciencedirect.com/science/article/pii/S089662730300535X, doi:https://doi.org/10.1016/j.neuron.2003.08.011.

[SMV+16] Brian F Sadacca, Narendra Mukherjee, Tony Vladusich, Jennifer X Li, Donald B Katz, and Paul Miller. The behavioral relevance of cortical neural ensemble responses emerges suddenly. *Journal of Neuroscience*, 36(3):655–669, 2016.

[Tan11] O. Tange. Gnu parallel - the command-line power tool. *;login: The USENIX Magazine*, 36(1):42–47, Feb 2011. URL: http://www.gnu.org/s/parallel, doi:http://dx.doi.org/10.5281/zenodo.16303.

[WBVI+04] Frank Wood, Michael J Black, Carlos Vargas-Irwin, Matthew Fellows, and John P Donoghue. On the variability of manual spike sorting. *IEEE Transactions on Biomedical Engineering*, 51(6):912–918, 2004.

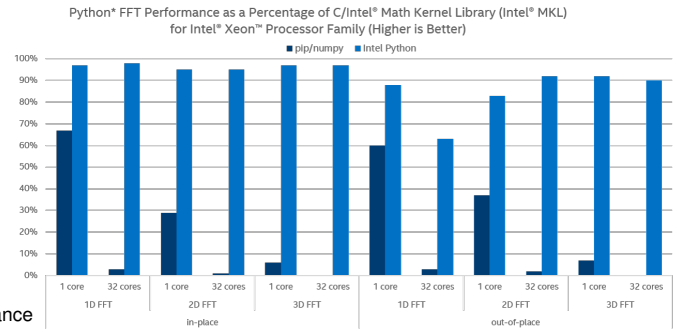
[WRL+15] Yingxue Wang, Sandro Romani, Brian Lustig, Anthony Leonardo, and Eva Pastalkova. Theta sequences are essential for internally generated hippocampal firing fields. *Nature neuroscience*, 18(2):282–288, 2015.

[YEH11] T Yokota, K Eguchi, and K Hiraba. Functional properties of putative pyramidal neurons and inhibitory interneurons in the rat gustatory cortex. *Cerebral Cortex*, 21(3):597–606, 2011.

Accelerating Scientific Python with Intel Optimizations

Oleksandr Pavlyk[‡*], Denis Nagorny[‡†], Andres Guzman-Ballen[‡†], Anton Malakhov[‡†], Hai Liu[‡†], Ehsan Totoni[‡†], Todd A. Anderson[‡†], Sergey Maidanov[‡†]

Abstract—It is well-known that the performance difference between Python and basic C code can be up 200x, but for numerically intensive code another speed-up factor of 240x or even greater is possible. The performance comes from software's ability to take advantage of CPU's multiple cores, single instruction multiple data (SIMD) instructions, and high performance caches. The article describes optimizations, included in Intel® Distribution for Python*, aimed to automatically boost performance of numerically intensive code. This paper is intended for Python programmers who want to get the most out of their hardware but do not have time or expertise to re-code their applications using techniques such as native extensions or Cython.



Index Terms—numpy,scipy,scikit-learn,numba,simd,parallel,optimization,performance

Introduction

Scientific software is usually algorithmically rich and compute intensive. The expressiveness of Python language as well as abundance of quality packages offering implementations of advanced algorithms allow scientists and engineers alike to code their software in Python. The ability of this software to solve realistic problems in a reasonable time is often hampered by inefficient use of hardware resources. Intel Distribution for Python [IDP] attempts to enable scientific Python community with optimized computational packages, such as NumPy*, SciPy*, Scikit-learn*, Numba* and PyDAAL across a range of Intel® processors, from Intel® Core™ CPUs to Intel® Xeon® and Intel® Xeon Phi™ processors. This paper offers a detailed report about optimization that went into the Intel® Distribution for Python*, which might be interesting for developers of SciPy tools.

Fast Fourier Transforms

Intel® Distribution for Python* offers a thin layered interface for the Intel® Math Kernel Library (Intel® MKL) that allows efficient access to native FFT optimizations from a range of NumPy and SciPy functions. The optimizations are provided for real and complex data types in both single and double precision. Update 2 improves performance of both one-dimensional and multi-dimensional transforms, for in-place and out-of-place modes of operation. As a result, Python performance may improve up to 60x over Update 1 and is now close to performance of native C/Intel MKL.

* *Corresponding author: Oleksandr.Pavlyk@intel.com*
‡ *Intel Corporation*
† *These authors contributed equally.*

Thanks to Intel® MKL's flexibility in its supports for arbitrarily strided input and output arrays[1] both one-dimensional and multi-dimensional complex Fast Fourier Transforms along distinct axes can be performed directly, without the need to copy the input into a contiguous array first (the cost of copying, whose complexity is $\mathcal{O}(n)$, is not negligible compared to the cost of computing the transform, whose complexity is $\mathcal{O}(n \log n)$, and copying, being memory bound, does not scale well with the number of available cores). Furthermore, input strides can be arbitrary, including negative or zero, as long strides remain an integer multiple of array's item size, otherwise a copy will be made.

The wrapper supports both in-place and out-of-place modes, enabling it to efficiently power both `numpy.fft` and `scipy.fftpack` submodules. In-place operations are only performed where possible.

Direct support for multivariate transforms along distinct array axis. Even when multivariate transform ends up being computed as iterations of one-dimensional transforms, all subsequent iterations are performed in place for efficiency.

The update also provides dedicated support for complex FFTs on real inputs, such as `np.fft.fft(real_array)`, by leveraging corresponding functionality in MKL[2].

Dedicated support for specialized real FFTs, which only store independent complex harmonics. Both `numpy.fft.rfft` and `scipy.fftpack.rfft` storage modes are natively supported via Intel® MKL.

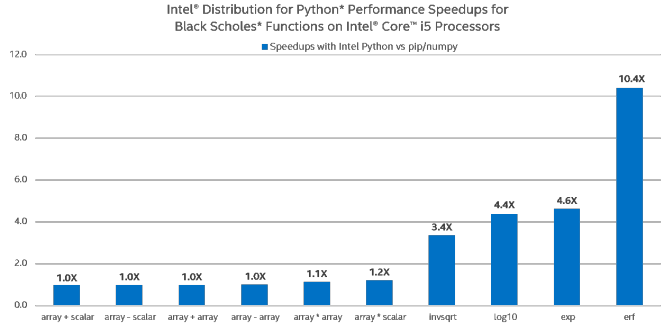
1. https://software.intel.com/en-us/mkl-developer-reference-c-dfti-input-strides-dfti-output-strides#10859C1F-7C96-4034-8E66-B671CE789AD6
2. https://software.intel.com/en-us/mkl-developer-reference-c-dfti-complex-storage-dfti-real-storage-dfti-conjugate-even-storage#CONJUGATE_EVEN_STORAGE

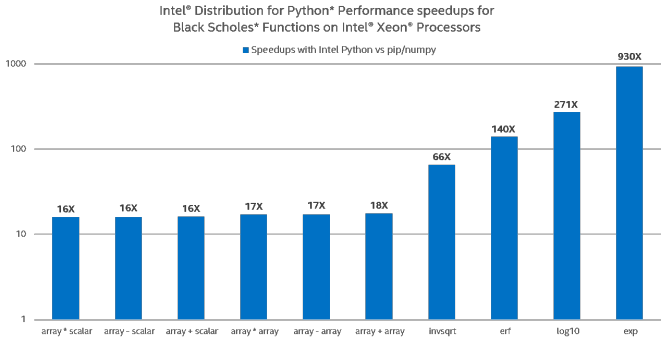
command	`fft(arg)`	`fft(arg, axis=0)`	`fft2(arg)`	`fftn(arg)`
`arg.shape`	$(3 \cdot 10^6,)$	*(1860, 1420)*	*(275, 274, 273)*	*(275, 274, 273)*
`arg.strides`	$(10 \cdot 16,)$	C-contiguous	F-contiguous	$(16, 274 \cdot 275 \cdot 16, 275 \cdot 16)$
repetitions	16	16	8	8
IDP 2017.0.3	0.162±0.01	0.113±0.01	8.87±0.08	0.86±0.01
IDP 2017.0.1	0.187±0.06	1.046±0.03	10.3±0.1	12.38±0.03
pip numpy	2.333±0.01	1.769±0.02	29.94±0.03	34.455±0.007

TABLE 1: *Table of total times of repeated executions of FFT computations using `np.fft` functions for arrays of complex doubles in different Python distributions on Intel (R) Xeon (R) E5-2698 v3 @ 2.30GHz with 64GB of RAM.*

command		`fft(arg)`	`fft(arg)`	`fft2(arg)`	`fft2(arg)`	`fftn(arg)`	`fftn(arg)`
`overwrite_x`		False	True	False	True	False	True
`arg.shape`		$(3 \cdot 10^6,)$	$(3 \cdot 10^6,)$	*(1860, 1420)*	*(1860, 1420)*	*(273, 274, 275)*	*(273, 274, 275)*
IDP 2017.0.3	cd	1.40±0.02	0.885±0.005	0.090±0.001	0.067±0.001	0.868±0.007	0.761±0.001
	cs	0.734±0.004	0.450±0.002	0.056±0.001	0.041±0.0002	0.326±0.003	0.285±0.002
IDP 2017.0.1	cd	1.77±0.02	1.760±0.012	2.208±0.004	2.219±0.002	22.77±0.38	22.7±0.5
	cs	5.79±0.14	5.75±0.02	1.996±0.1	2.258±0.001	27.12±0.05	26.8±0.25
pip numpy	cd	26.06±0.01	23.51±0.01	4.786±0.002	3.800±0.003	67.69±0.12	81.46±0.01
	cs	28.4±0.1	11.9±0.05	5.010±0.003	3.77±0.02	69.49±0.02	80.54±0.07

TABLE 2: *Table of times of repeated execution of `scipy.fftpack` functions with `overwrite_x=True` (in-place) and `overwrite_x=False` (out-of-place) on a C-contiguous arrays of complex double and complex singles.*

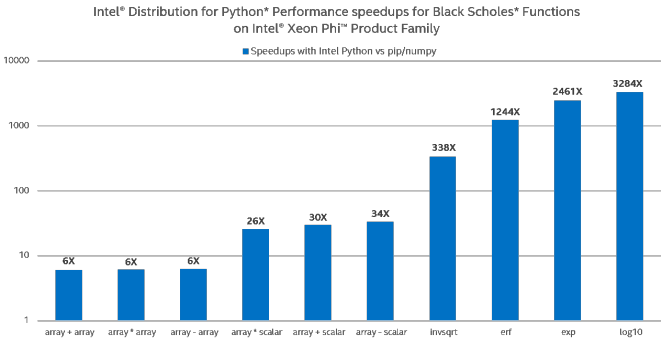




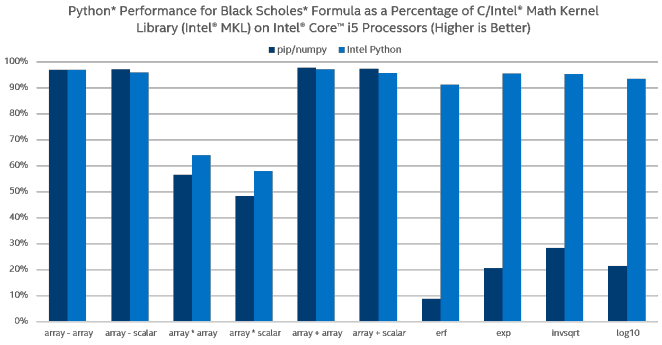
Arithmetic and transcendental expressions

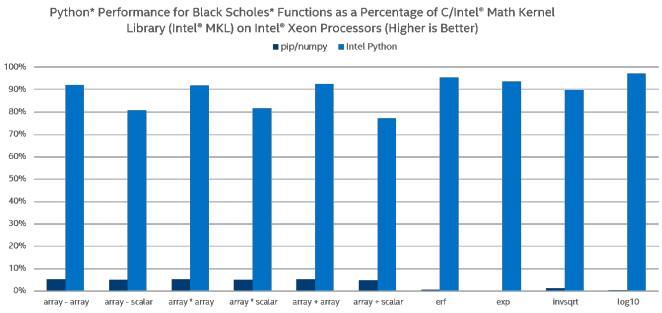
One of the great benefits of the Intel® Distribution for Python* is the performance boost gained from leveraging SIMD and multithreading in (select) NumPy's UMath arithmetic and transcendental operations across the range of Intel® CPUs, from Intel® Core™ to Intel® Xeon™ & Intel® Xeon Phi™. With stock Python as our baseline, we demonstrate the scalability of Intel® Distribution for Python* by using functions that are intensively used in financial math applications and machine learning:

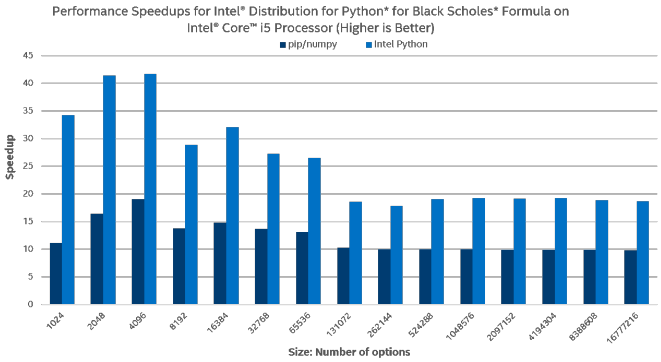
One can see that stock Python (pip-installed NumPy from PyPI) on Intel® Core™ i5 performs basic operations such as addition, subtraction, and multiplication just as well as Intel® Python, but not on Intel® Xeon™ and Intel® Xeon Phi™, where Intel® Distribution for Python* provides over 10x speedup. This can be explained by the fact that basic arithmetic operations in stock NumPy are hard-coded AVX intrinsics (and thus already leverage SIMD, but do not scale to other instruction set architectures (ISA), e.g. AVX-512). These operations in stock Python also do not leverage multiple cores (i.e. no multi-threading of loops under the hood of NumPy exist with such operations). Intel Python's implementation allows for this scalability by utilizing

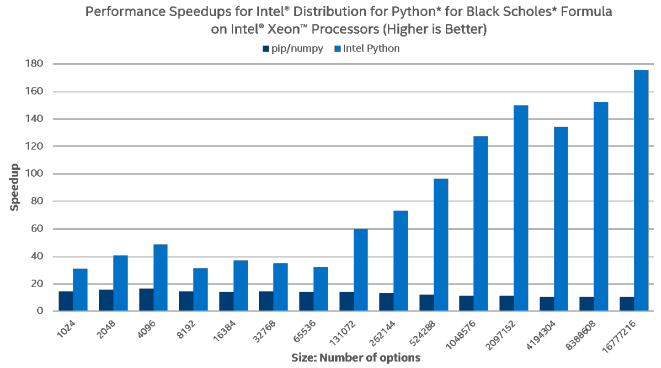


both respective Intel® MKL VML CPU-dispatched and multi-threaded primitives under the hood, and Intel® SVML intrinsics - a compiler-provided short vector math library that vectorizes math functions for both IA-32 and Intel® 64-bit architectures on supported operating systems. Depending on the problem size, NumPy will choose one of the two approaches. On small array sizes, Intel® SVML outperforms VML due to high library call overhead, but for larger problem sizes, VML's ability to both vectorize math









functions and multi-thread loops offsets the overhead.

Specifically, on Intel® Core™ i5 processor the Intel® Distribution for Python delivers greater performance in numerical evaluation of transcendental functions (log, exp, erf, etc.) due to utilization of both SIMD and multi-threading. We do not see any visible benefit of multi-threading basic operations (as shown on the graph) unless NumPy arrays are very large (not shown on the graph). On Intel® Xeon™ processor, the 10x-1000x boost is explained by leveraging both (a) AVX2 instructions to evaluate transcendentals and (b) multiple cores (32 in our setup). Even greater scalability of Intel® Xeon Phi™ relative to Intel® Xeon™ is explained by larger number of cores (64 in our setup) and wider vector registers.

The following charts provide another view of Intel® Distribution for Python performance versus stock Python on arithmetic and transcendental vector operations in NumPy by measuring how close UMath performance is to the respective native MKL call:

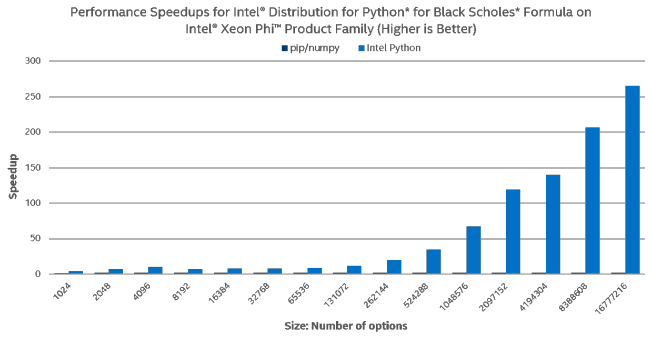
Again on Intel® Core™ i5 the stock Python performs well on basic operations (due to hard-coded AVX intrinsics and because multi-threading from Intel® Distribution for Python does not add much on basic operations) but does not scale on transcendentals (loops with transcendentals are not vectorized in stock Python). Intel® Distribution for Python delivers performance close to native speeds (90% of MKL) on relatively big problem sizes.

To demonstrate the benefits of vectorization and multi-threading in a real-world application, we chose to use the Black Scholes model, used to estimate the price of financial derivatives, specifically European vanilla stock options. A Python implementation of the Black Scholes formula gives an idea of how NumPy UMath optimizations can be noticed at the application level:

One can see that on Intel® Core™ i5 the Black Scholes Formula scales nicely with Intel Python on small problem sizes but does not perform well on bigger problem sizes, which is explained by small cache sizes. Stock Python does marginally scale due to leveraging AVX instructions on basic arithmetic operations, but it is a whole different story on Intel® Xeon™ and Intel® Xeon Phi™. Using Intel® Distribution for Python to execute the same Python code on server processors, much greater scalability on much greater problem sizes is observed. Intel® Xeon Phi™ scales better due to bigger number of cores and as expected, while the stock Python does not scale on server processors due to the lack of AVX2/AVX-512 support for transcendentals and no utilization of multiple cores.

Memory management optimizations

Update 2 introduces extensive optimizations in NumPy memory management operations. As a dynamic language, Python manages memory for the user. Memory operations, such as allocation, deallocation, copy, and move, affect performance of essentially all Python programs. Specifically, Update 2 ensures NumPy allocates arrays that are properly aligned in memory (their address is a multiple of a specific factor, usually 64) on Linux, so that NumPy and SciPy compute functions can benefit from respective aligned versions of SIMD memory access instructions. This is especially relevant for Intel® Xeon Phi™ processors. The most significant improvements in memory optimizations in Update 2



comes from replacing original memory copy and move operations with optimized implementations from Intel® MKL. The result: improved performance because these Intel® MKL routines are optimized for both a range of Intel® CPUs and multiple CPU cores.

Faster Machine Learning with Scikit-learn

Scikit-learn is well-known library that provides a lot of algorithms for many areas of machine learning. Having limited developer resources this project prefers universal solutions and proven algorithms. For performance improvement scikit-learn uses Cython and underlying BLAS/LAPACK libraries through SciPy and Numpy. OpenBLAS and MKL uses threaded based parallelism to utilize multicores of modern CPUs. Unfortunately BLAS/LAPACK's functions are too low level primitives and their usage is often not very efficient comparing to possible high-level parallelism. For high-level parallelism scikit-learn uses multiprocessing approach that is not very efficient from technical point of view. On the other hand Intel provides Intel® Data Analytics Acceleration Library (Intel® DAAL) that helps speed up big data analysis by providing highly optimized algorithmic building blocks for all stages of data analytics (preprocessing, transformation, analysis, modeling, validation, and decision making) in batch, online, and distributed processing modes of computation. It is originally written in C++ and provides Java and Python bindings. DAAL is heavily optimized for all Intel® Architectures including Intel® Xeon Phi™, but it is not at all clear how to use DAAL binding from Python. DAAL bindings for python are generated automatically and reflects original C++ API very closely. This makes its usage quite complicated because of its use of non pythonic idioms and scarce documentation.

In order to combine the power of well optimized native code with the familiar to machine learning community API the Intel Distribution for Python includes fruits of efforts of scikit-learn optimization. Thus beginning with version 2017.0.2 the Intel Distribution for Python includes scikit-learn with daal4sklearn submodule. Specifically, daal4sklearn optimizes Principal Component Analysis (PCA), Linear and Ridge Regressions, Correlation and Cosine Distances, and K-Means in scikit-learn using Intel® DAAL. Speedups may range from 1.5x to 160x.

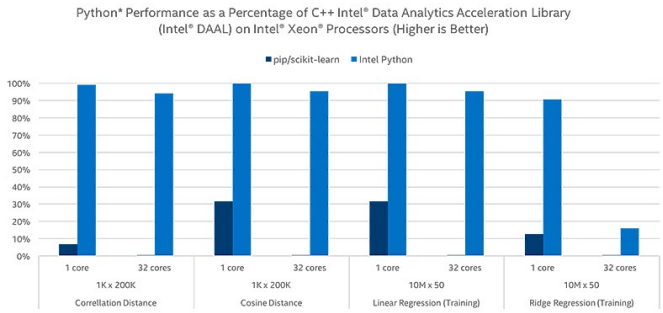
There is no direct matching between scikit-learn's and Intel® DAAL's APIs. Moreover, they aren't fully compatible for all inputs, therefore in those cases where daal4sklearn detects incompatibility it falls back to original sklearn's implementation.

Scikit-learn uses multiprocessing approach to parallelize computations. The unfortunate consequence of this choice may be a large memory footprint as each cloned process has access to its own copy of all input data. This precludes scikit-learn from effectivly utilizing many-cores architectures as Intel® Xeon Phi™ for big workloads. On the other hand DAAL internally uses multi-threading approach sharing the same data across all cores. This allows to DAAL to use less memory and to process bigger workloads which especially important for ML algorithms.

Daal4sklearn is enabled by default and provides a simple API to toggle these optimizations:

```
from sklearn.daal4sklearn import dispatcher
dispatcher.disable()
dispatcher.enable()
```

Several benchmarks [sklearn_benches] were prepared to demonstrate performance that can be achieved with Intel® DAAL. A



fragment from the benchmark used to measure performance of K-means is given below.

```
problem_sizes = [
        (10000,  2),  (10000,  25), (10000,  50),
        (50000,  2),  (50000,  25), (50000,  50),
        (100000, 2),  (100000, 25), (100000, 50)]
X={}
for rows, cols in problem_sizes:
    X[(rows, cols)] = rand(rows, cols)

kmeans = KMeans(n_clusters=10, n_jobs=args.proc)

@st_time
def train(X):
    kmeans.fit(X)

for rows, cols in problem_sizes:
    print (rows, cols, end=' ')
    X_local = X[(rows, cols)]
    train(X_local)
    print('')
```

Using all 32 cores of Intel® Xeon® processor E5-2698 v3 IDP's K-Means can be more than 50 times faster than the python included with Ubuntu 14.04. P below means the number of CPU cores used.

We compared the similar runs for other algorithms and normalized results by results obtained with DAAL in C++ without python to estimate overhead from python wrapping.

You can find some benchmarks [sklearn_benches]

Numba vectorization

Wikipedia defines SIMD as:

> Single instruction, multiple data (SIMD), is a class of parallel computers in Flynn's taxonomy. It describes computers with multiple processing elements that perform the same operation on multiple data points simultaneously. Thus, such machines exploit data level parallelism, but not concurrency: there are simultaneous (parallel) computations, but only a single process (instruction) at a given moment. Most modern CPU designs include SIMD instructions in order to improve the performance of multimedia use.

To utilize power of CPU's SIMD instructions compilers need to implement special optimization passes, so-called code vectorization. Modern optimizing compilers implement automatic vectorization - a special case of automatic parallelization, where a computer program is converted from a scalar implementation, which processes a single pair of operands at a time, to a vector implementation, which processes a single operation on multiple pairs of operands at once.

rows	cols	IDP,s P=1	IDP,s P=32	System,s P=1	System,s P=32	Vs System,P=1	Vs System,P=32
10000	2	0.01	0.01	0.38	0.27	28.55	36.52
10000	25	0.05	0.01	1.46	0.57	27.59	48.22
10000	50	0.09	0.02	2.21	0.87	23.83	40.76
50000	2	0.08	0.01	1.62	0.57	20.57	47.43
50000	25	0.67	0.07	14.43	2.79	21.47	38.69
50000	50	1.05	0.10	24.04	4.00	22.89	38.52
100000	2	0.15	0.02	3.33	0.87	22.30	56.72
100000	25	1.34	0.11	33.27	5.53	24.75	49.07
100000	50	2.21	0.17	63.30	8.36	28.65	47.95

TABLE 3

According Numba's project page Numba is an Open Source NumPy-aware optimizing compiler for Python. It uses the remarkable LLVM compiler infrastructure to compile Python syntax to machine code. And it is quite expected that Numba tries to use all these features to improve performance especially for scientific applications.

LLVM implemented auto-vectorization for simple cases several years ago but there remain sigificant problems with vectorization of elementary transcendental math functions. To enable proper vectorization support a special vectorized implementation of math functions such as `sin`, `cos`, `exp` is needed.

The Intel® C++ Compiler provides short vector math library (SVML) intrinsics implementing vectorized mathematical functions. These intrinsics are available for IA-32 and Intel® 64 architectures running on supported operating systems.

The SVML intrinsics are vector variants of corresponding scalar math operations using `__m128`, `__m128d`, `__m256`, `__m256d`, and `__m256i` data types. They take packed vector arguments, simultaneously perform the operation on each element of the packed vector argument, and return a packed vector result. Due to low overhead of the packing for aligned contiguously laid out data, vector operations may offer speed-ups over scalar operations which are proportional to the width of the vector register.

For example, the argument to the `_mm_sin_ps` intrinsic is a packed 128-bit vector of four 32-bit precision floating point numbers. The intrinsic simultaneously computes values of the sine function for each of these four numbers and returns the four results in a packed 128-bit vector, all within about the time of scalar evaluation of only one argument.

Using SVML intrinsics is faster than repeatedly calling the scalar math functions. However, the intrinsics may differ from the corresponding scalar functions in accuracy of their results.

Besides intrinsics available with Intel® compiler there is opportunity to call vectorized implementations directly from svml library by their names.

Beginning with version 4.0 LLVM features (experimental) model of autovectorization using SVML library, so a full stack of technologies is now available to exploit in-core parallelization of python code. To enable the autovectorization feature in Numba, included in the Intel® Distribution for Python*, user needs to set `NUMBA_INTEL_SVML` environmental variable to a non-zero value, prompting Numba to load SVML library and to pass an appropriate option to LLVM.

Let's see how it works with a small example:

```
import math
import numpy as np
from numba import njit

def foo(x,y):
    for i in range(x.size):
        y[i] = math.sin(x[i])
foo_compiled = njit(foo)
```

Inspite of the fact that numba generates call for usual `sin` function, as seen in the following excerpt from the generated LLVM code:

```
label 16:
    $16.2 = iternext(value=$phi16.1)     ['$16.2',
                                          '$phi16.1']
    $16.3 = pair_first(value=$16.2)      ['$16.2',
                                          '$16.3']
    $16.4 = pair_second(value=$16.2)     ['$16.2',
                                          '$16.4']
    del $16.2                            []
    $phi19.1 = $16.3                     ['$16.3',
                                          '$phi19.1']
    del $16.3                            []
    branch $16.4, 19, 48                 ['$16.4']
label 19:
    del $16.4                            []
    i = $phi19.1                         ['$phi19.1',
                                          'i']
    del $phi19.1                         []
    $19.2 = global(math: <module 'math'\
     from '/path_stripped/lib-dynload/\
     math.cpython-35m-x86_64-...,.so'>) ['$ 19.2']
    $19.3 = getattr(attr=sin,
                    value=$19.2)         ['$19.2',
                                          '$19.3']
    del $19.2                            []
    $19.6 = getitem(index=i, value=x)    ['$19.6',
                                          'i', 'x']
    $19.7 = call $19.3($19.6)            ['$19.3',
                                          '$19.6',
                                          '$19.7']
    del $19.6                            []
    del $19.3                            []
    y[i] = $19.7                         ['$19.7',
                                          'i', 'y']
    del i                                []
    del $19.7                            []
    jump 16                              []
```

We can see direct use of the SVML-provided vector implementation of sine function:

```
leaq    96(%rdx), %r14
leaq    96(%rsi), %r15
movabsq $__svml_sin4_ha, %rbp
movq    %rbx, %r13
.p2align        4, 0x90
```

```
.LBB0_13:
        vmovups -96(%r14), %ymm0
        vmovups -64(%r14), %ymm1
        vmovups %ymm1, 32(%rsp)
        vmovups -32(%r14), %ymm1
        vmovups %ymm1, 64(%rsp)
        vmovups (%r14), %ymm1
        vmovups %ymm1, 128(%rsp)
        callq   *%rbp
        vmovups %ymm0, 96(%rsp)
        vmovups 32(%rsp), %ymm0
        callq   *%rbp
        vmovups %ymm0, 32(%rsp)
        vmovups 64(%rsp), %ymm0
        callq   *%rbp
        vmovups %ymm0, 64(%rsp)
        vmovupd 128(%rsp), %ymm0
        callq   *%rbp
        vmovups 96(%rsp), %ymm1
        vmovups %ymm1, -96(%r15)
        vmovups 32(%rsp), %ymm1
        vmovups %ymm1, -64(%r15)
        vmovups 64(%rsp), %ymm1
        vmovups %ymm1, -32(%r15)
        vmovupd %ymm0, (%r15)
        subq    $-128, %r14
        subq    $-128, %r15
        addq    $-16, %r13
        jne     .LBB0_13
```

Thanks to enabled support of high accuracy SVML functions in LLVM this jitted code sees more than 4x increase in performance.

svml enabled:

```
%timeit foo_compiled(x,y)
1000 loops, best of 3: 403 us per loop
```

svml disabled:

```
%timeit foo_compiled(x,y)
1000 loops, best of 3: 1.72 ms per loop
```

Auto-parallelization for Numba

In this section, we introduce a new feature in Numba that automatically parallelizes NumPy programs. Achieving high performance with Python on modern multi-core CPUs is challenging since Python implementations are generally interpreted and prohibit parallelism. To speed up sequential execution, Python functions can be compiled to native code using Numba, implemented with the LLVM just-in-time (JIT) compiler. All a programmer has to do to use Numba is to annotate their functions with Numba's `@jit` decorator. However, the Numba JIT will not parallelize NumPy functions, even though the majority of them are known to have parallel semantics, and thus cannot make use of multiple cores. Furthermore, even if individual NumPy functions were parallelized, a program containing many such functions would likely have lackluster performance due to poor cache behavior. Numba's existing solution is to allow users to write scalar kernels in OpenCL style, which can be executed in parallel. However, this approach requires significant programming effort to rewrite existing array code into explicit parallelizable scalar kernels and therefore hurts productivity and may be beyond the capabilities of some programmers. To achieve both high performance and high programmer productivity, we have implemented an automatic parallelization feature as part of the Numba JIT compiler. With auto-parallelization turned on, Numba attempts to identify operations with parallel semantics and to fuse adjacent ones together to form kernels that are automatically run in parallel, all fully automated without manual effort from the user.

Our implementation supports the following parallel operations:

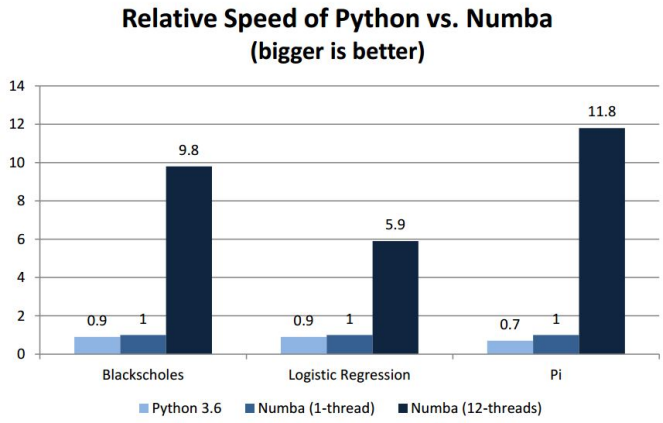
1) Common arithmetic functions between NumPy arrays, and between arrays and scalars, as well as NumPy ufuncs. They are often called *element-wise* or *point-wise* array operations:
   - unary operators: `+ - ~`
   - binary operators: `+ - * / /? % | >> ^ << & ** //`
   - comparison operators: `== != < <= > >=`
   - NumPy ufuncs that are supported in Numba's nopython mode.
   - User defined *DUFunc* through `@vectorize`.
2) NumPy reduction functions `sum` and `prod`, although they have to be written as `numpy.sum(a)` instead of `a.sum()`.
3) NumPy `dot` function between a matrix and a vector, or two vectors. In all other cases, Numba's default implementation is used.
4) Multi-dimensional arrays are also supported for the above operations when operands have matching dimension and size. The full semantics of NumPy broadcast between arrays with mixed dimensionality or size is not supported, nor is the reduction across a selected dimension.
5) NumPy `array` created from list comprehension is turned into direct array allocation and initialization without intermediate list.
6) Explicit parallelization via `prange` that turns a for-loop into a parallel loop.

As an example, consider the following Logistic Regression function:

```
@jit(parallel=True)
def logistic_regression(Y, X, w, iters):
    for i in range(iters):
        w += np.dot(
                Y / (1.0 + np.exp(Y * np.dot(X, w))),
                X )
    return w
```

We will not discuss details of the algorithm, but instead focus on how this program behaves with auto-parallelization:

1) Input `Y` is a vector of size `N`, `X` is an `N x D` matrix, and `w` is a vector of size `D`.
2) The function body is an iterative loop that updates variable `w`. The loop body consists of a sequence of vector and matrix operations.
3) The inner `dot` operation produces a vector of size `N`, followed by a sequence of arithmetic operations either between a scalar and vector of size `N`, or two vectors both of size `N`.
4) The outer `dot` produces a vector of size `D`, followed by an inplace array addition on variable `w`.
5) With auto-parallelization, all operations that produce array of size `N` are fused together to become a single parallel kernel. This includes the inner `dot` operation and all point-wise array operations following it.
6) The outer `dot` operation produces a result array of different dimension, and is not fused with the above kernel.



Here, the only thing required to take advantage of parallel hardware is to set the `parallel=True` option for `@jit`, with no modifications to the `logistic_regression` function itself. If we were to give an equivalent parallel implementation using Numba's `@guvectorize` decorator, it would require a pervasive change that rewrites the code to extract kernel computation that can be parallelized, which is both tedious and challenging.

We measure the performance of automatic parallelization over three workloads, comparing auto-parallelization with Numba's sequential JIT and Python 3.6, normalized to the sequential (1-thread) speed of Numba.

Auto-parallelization proves to be an effective optimization for these benchmarks, achieving speedups from 5.9x to 11.8x over sequential Numba on 12-core Intel® Xeon® X5680 @3.33GHz with 64GB RAM. The benchmarks are available as part of Numba's source distribution [numba].

Our future plan is to support array range selection, enable auto-parallelization of more NumPy functions, as well as to add new features such as iterative stencils. We also plan to implement more optimizations that help make parallel programs run fast, improving both performance and productivity for Python programmers in the scientific domain.

Summary

The Intel® Distribution for Python is powered by Anaconda* and conda build infrastructures that give all Python users the benefit of interoperability within these two environments and access to the optimized packages through a simple `conda install` command. Intel® Distribution for Python* delivers significant performance optimizations for many core algorithms and Python packages, while maintaining the ease of downloading and installation.

REFERENCES

[fft_bench] http://github.com/intelpython/fft_benchmark
[sklearn_benches] https://github.com/dvnagorny/sklearn_benchs
[numba] https://github.com/numba/numba
[IDP] Intel |R| Distribution for Python*

NEXT: A system to easily connect crowdsourcing and adaptive data collection

Scott Sievert[‡†*], Daniel Ross[‡†], Lalit Jain[§†], Kevin Jamieson[¶], Rob Nowak[‡], Robert Mankoff[‖]

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

Abstract—Obtaining useful crowdsourcing results often requires more responses than can be easily collected. Reducing the number of responses required can be done by *adapting* to previous responses with "adaptive" sampling algorithms, but these algorithms present a fundamental challenge when paired with crowdsourcing. At UW–Madison, we have built a powerful crowdsourcing data collection tool called NEXT (http://nextml.org) that can be used with arbitrary adaptive algorithms. Each week, our system is used by The New Yorker to run their Cartoon Caption contest (http://www.newyorker.com/cartoons/vote). In this paper, we will explain what NEXT is and it's applications, architecture and experimentalist use.

Index Terms—crowdsourcing, adaptive sampling, system

Introduction

The ubiquitousness of the Internet has enabled crowdsourcing, which gives fast access to unprecedented amounts of human judgment data. For example, millions of crowdsourcing participants have been asked to determine the locations in an image that contain a certain object (e.g., "select all image locations that contain buildings") on many different images [DDS+09].

The cost of collecting crowdsourcing responses can be significant – especially in problem domains where expert input is required. Minimizing the number of queries required has large practical benefits: higher accuracy with fewer responses, and ultimately a shorter time to the result. To obtain these benefits, a fundamental change in the method of data collection is required.

At UW–Madison, we have developed a crowdsourcing data collection tool that efficiently collects crowdsourced data via "adaptive" sampling algorithms [JJF+15]. In this paper, we will focus on the use of NEXT rather than the applications of NEXT and their results. We will mention the fundamental problem NEXT addresses, its applications, and the interfaces NEXT presents to the experimentalist and algorithm designer.

Problem statement

Supervised machine learning relies humans to label examples in order to build a model to predict the response a human would give [KZP07]. One example of this workflow is with the popular ImageNet dataset [DDS+09]: humans have provided millions of image labels, and there have been dozens of models to predict labels for unseen images [SLJ+15], [HZRS15], [SZ14].

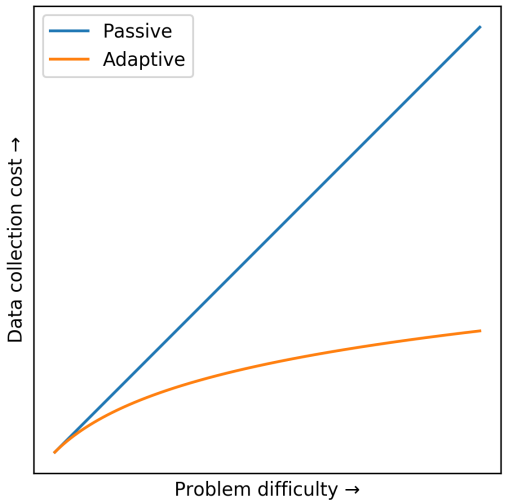


Fig. 1: *As problem difficulty increases, fewer samples (e.g., labeled images) are needed with adaptive sampling to reach a particular quality (e.g., classification accuracy).*

The collection of these data is *passive* and does not *adapt* to previous responses: previous responses do not effect which queries are presented. Adaptive data collection is a process which selects the most useful data as quickly as possible to help achieve some goal (e.g., classification accuracy) [Hol92]. Adaptive data collection is done by an adaptive sampling algorithm that chooses the next query to be labeled.

Adaptive data collection naturally requires fewer responses to produce the same model as passive data collection: it's adapting to previous responses by choosing which query to present next. This is most useful when many labels are needed unlabeled examples. Adaptive algorithms do not require more responses than passive algorithms [CWN05]. A representative depiction of gains obtained by adaptive data collection is shown in Figure 1 [DHM08].

Applying adaptive data collection to crowdsourcing has the potential to reduce the number of samples required. An simple example that requires many human judgments is sorting n items with pairwise comparisons (e.g., $x < y$). In the ideal case, an

† *These authors contributed equally.*
∗ *Corresponding author: stsievert@wisc.edu*
‡ *University of Wisconsin–Madison*
§ *University of Michigan, Ann Arbor*
¶ *University of California, Berkeley*
‖ *The New Yorker*

adaptive algorithm requires $O(n \log n)$ comparisons on average while passive algorithms requires $O(n^2)$ comparisons [Hoa62].

Adaptively collecting large-scale datasets is challenging and time-consuming, as mentioned below. As such, the evaluation of novel adaptive sampling algorithms resort to simulations that use large passively collected datasets. These simulations do not address the practical issues faced in crowdsourcing: adaptive algorithm response time, human fatigue and differing label quality among humans.

The problem that needs to be solved is to allow arbitrary adaptive algorithms to collect crowdsourced data in real time by experimentalists. Arguably, some of the deepest insights and greatest innovations have come through experimentation. This is only possible if adaptive data collection is easily accessible by both

1) Machine learning researchers, to test and deploy adaptive algorithms
2) Experimentalists, to use and test adaptive algorithms in real-world applications

Easy use by both groups will enable feedback between experimentalists and machine learning researchers to improve adaptive data collection through crowdsourcing.

Challenges

Adaptive data collection is not possible without access to previous responses, a fundamental change to data collection. This introduces human feedback: the most useful queries are selected using previously recorded human labels by some adaptive algorithm. If a particular query has shown to be of little use, it doesn't make much sense to label the same query again.

Adaptive algorithms use previous responses to ask questions, which means that they require

- receiving, storing and accessing responses
- delivering and selecting queries to be labeled
- updating some internal model which selects queries to be presented.
- scaling to tens or hundreds of simultaneous users in an online environment when applied to crowdsourcing

General crowdsourcing systems (e.g., Mechanical Turk, PsiTurk, Crowd Flower) were not designed with these requirements in mind. Adaptive data collection requires a fundamentally different interaction flow as show in Figure 2, which requires the data flow in Figure 3 when applied to crowdsourcing.

Crowdsourcing adaptive data collection presents a variety of challenges in mathematics, systems and software development. These challenges stem from the storage and connection of responses to the adaptive sampling algorithm. Any such system needs to process, store and receive crowdsourcing responses and work crowdsourcing scale, meaning the development and maintenance of such a system is involved. This has served as a barrier to developing such a system for mathematicians, and lack of knowledge on adaptive methods have hindered experimentalists.

One other system that addresses this challenge is the Microsoft Decision Service [ABC+16], which can effectively evaluate the collection of crowdsourced data with different adaptive algorithms. However, design of this system involved different goals, including working with exactly one problem formulation and working well at very large scales.

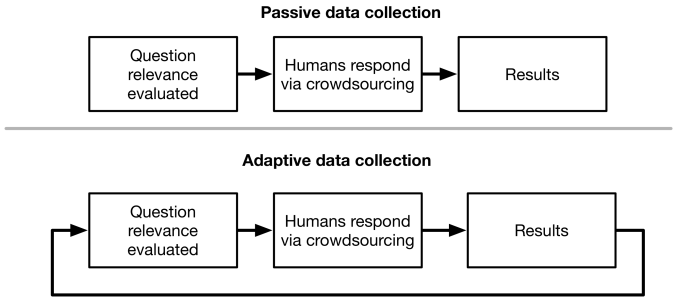


Fig. 2: *The data flows required to collect crowdsourcing data both passively and adaptively. The primary difference is adaptive data collection requires using previous responses in some way.*

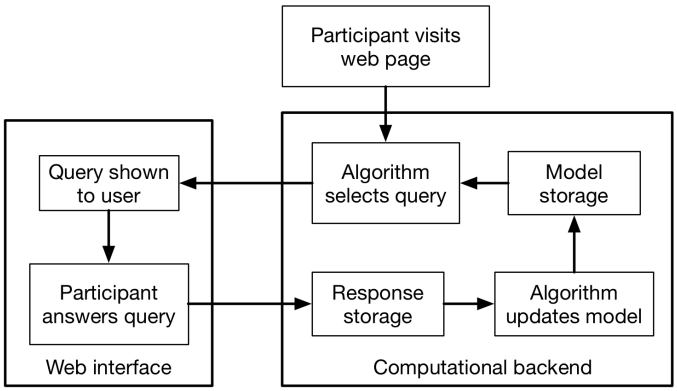


Fig. 3: *The system required to use adaptive algorithm with crowdsourcing. The results are stored in the model, which may contain additional information.*

Our system

The system we have developed at the UW–Madison is called NEXT[1][2]. It provides adaptive, crowdsourced data collection by selecting which query to present *next*. NEXT provides

- easy implementation, selection, and evaluation of different adaptive algorithms
- a web interface for crowdsourced experiment participation
- an HTTP-based API for experiment access (and for use in other contexts)
- live experiment monitoring dashboards that update as responses are received
- easy use and configuration by experimentalists in a wide variety of fields and disciplines

Our design goals necessitate that NEXT be an end-to-end system that is easily accessible. It is a web interface that can be accessed by both experimentalists and crowdsourcing participants, and a Python interface for the algorithm developer. We explain use by experimentalists and algorithm developers in the following sections. A block diagram representation of our system is in Figure 4.

In use of NEXT, mathematicians have implemented new algorithms [Jun16] and UW–Madison psychologists have independently used our system[3]. NEXT has been used by the New Yorker and in the insurance industry. In at least one case, two

1. Homepage at http://nextml.org
2. Source available at https://github.com/nextml/NEXT

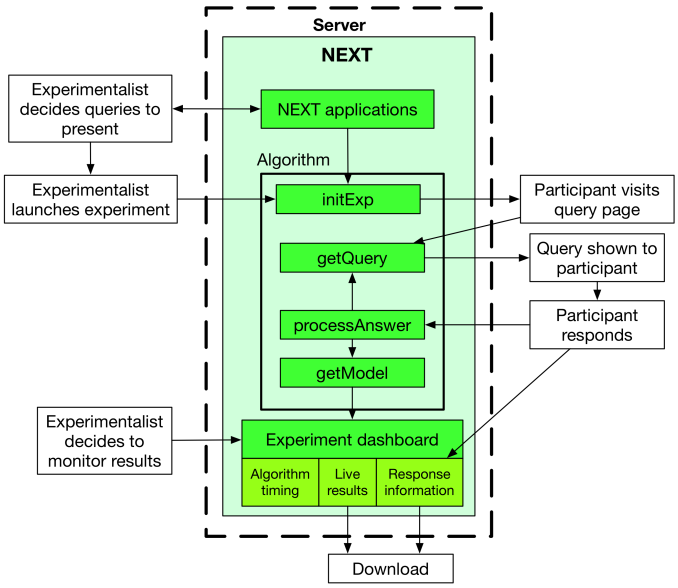


Fig. 4: *When and how different users interact with NEXT. Arrows represent some form of communication between different system components.*

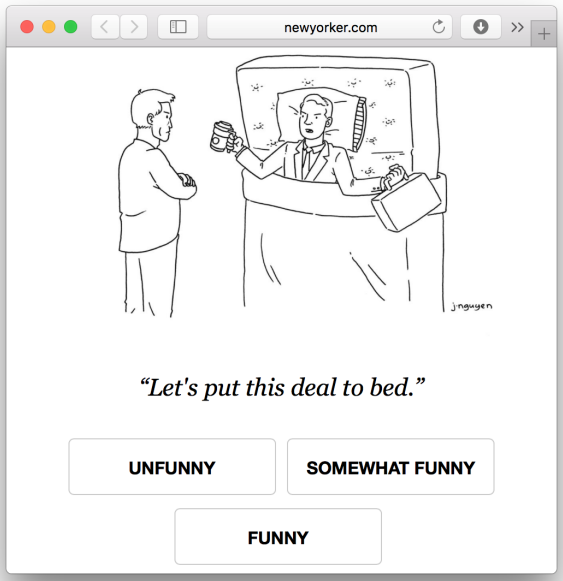


Fig. 5: *An example query shown in The New Yorker Caption Contest (cartoon drawn by P. C. Vey)*

adaptive algorithms have been evaluated in the real world and one required fewer samples as expected[4].

In our usage, the system remains responsive to participants even after receiving millions of responses from thousands of participants. This is illustrated by the problem below, though it also illustrates other features.

Applications of NEXT

NEXT *applications* control the presentation of queries for users to consider.

There are three "built-in" applications shipped with NEXT, geared to three different types of judgments a user can make. These applications are

- Cardinal bandits, which asks participants to rate one object [GGL12] as shown in Figure 5.
- Dueling bandits, which asks participants to select one of two objects [YBKJ12] as shown in Figure 6.
- Triplets, which displays three objects and asks for *triplet responses* of the form "object i is more similar to object j than object k." [JJN16], as shown in Figure 7.

We will now describe each application in more detail.

Cardinal bandits

Each week, The New Yorker draws a cartoon and asks readers for funny captions. They receive about 5,000 captions, of which they have to find the funniest. NEXT runs this contest each week. The interface NEXT provides is visible at http://www.newyorker.com/cartoons/vote and in Figure 5.

The interface is presented every time a query is generated. One caption is presented below the comic with buttons to rate the caption as "unfunny", "somewhat funny" or "funny". Every time one of these buttons is pressed, the adaptive algorithm processes the response and generates a new query.

Each week, we collect and record up to a million ratings from over 10,000 users. All told, this dataset[5] includes over 20 million ratings on over 363,000 different captions. This dataset has been of practical use in improving adaptive sampling algorithms [Jun16].

The New Yorker's goal is to find the funniest caption from this set of 5,000 captions[6]. To achieve this, the algorithms of choice only sample captions that can possibly be the funniest. If a caption has received only "unfunny" ratings, it is probably not the funniest caption and should not be further sampled.

This system has enabled evaluation and improvement in algorithm implementation. In initial contests, we verified that one adaptive algorithm [JMNB14] saw gains over a random algorithm. Later, we implemented an improved adaptive algorithm (KL-UCB at [KK13]) and saw adaptive gains as expected.

This was one of the motivations for NEXT: enabling easy evaluation of adaptive algorithms.

Dueling bandits

We also support asking the crowdsourcing participants to chose the "best" of two items. We tried this method during the first several caption contests we launched for The New Yorker. This interface asks participants to select the funnier of two captions, and is shown in Figure 6. This problem formulation has theoretic guarantees on finding the best item in a set [AB10], but can also be applied to ranking different objects [CBCTH13].

The early evaluation of dueling bandits in the Caption Contest is again part of why we developed NEXT. After trying dueling bandits for several contests, we decided using cardinal bandits is preferable. Cardinal bandits works better at scale, and requires less work by The New Yorker.

3. See http://concepts.psych.wisc.edu/index.php/next-tutorial/
4. With contest 559 of The New Yorker Cartoon Caption contest
5. https://github.com/nextml/caption-contest-data
6. The top caption for the comic in Figure 5 was "Like you've never taken anything from a hotel room"

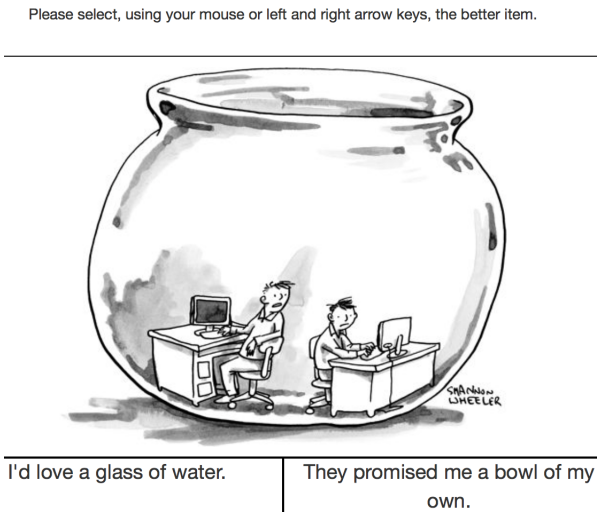


Fig. 6: *The dueling bandits interface, where two items are compared and the "better" item is selected (cartoon drawn for The New Yorker Caption Contest by Shannon Wheeler)*

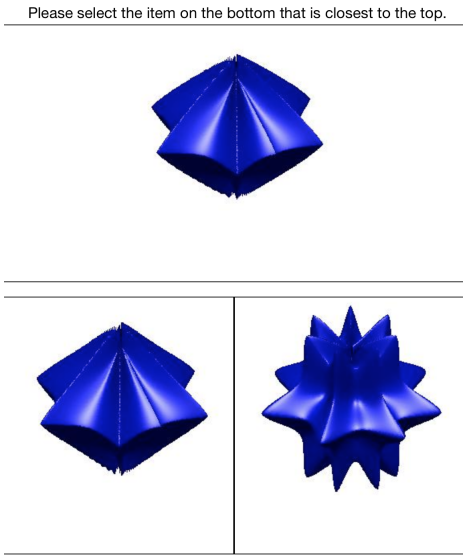


Fig. 7: *An interface that asks the user to select the most similar bottom object in relation to the top object.*

Triplets

Finding a similarity measure between different objects is the goal of this problem formulation. For example, it may be desired to find the similarity between different facial expressions. Happy and excited faces may be similar but are probably different from sad faces.

Human attention span cannot handle the naive number of comparisons (which is proportional to n^2 with n items). Instead, we ask the crowdsourcing participant to make a pairwise similarity judgement, or a triplet response as shown in Figure 7. There are theoretic guarantees on finding some similarity measure given these responses [JJN16] and have been used in practice with NEXT to compare visual representations of different molecules [RMN16].

NEXT Architecture

The design goals of NEXT are to provide

- convenient default *applications* (which handle different problem formulations by serving different types of queries; e.g., one application involves the rating of exactly one object)
- straightforward and modular algorithm implementation
- live experiment monitoring tools via a dashboard, which must update as responses are received and provide some sort of offline access
- easy experimentalist use, both in system launch and in experiment launch

These different system components and their data flow is shown in Figure 4. Complete system documentation is available and addresses use cases seen by both algorithm developers and experimentalists[7].

Algorithm implementation

Required functions: To implement Figure 4, we must implement four functions for each algorithm:

1) `initExp`, which initializes the algorithm when the experiment is launched
2) `getQuery`, which generates a query to show one participant
3) `processAnswer`, which processes the human's answer
4) `getModel`, which gets the results and is shown on the dashboard

These function handle various objects to displayed in each query (e.g., the New Yorker displays one text object in every query for a rating). By default, these objects or *target* are abstracted to an integer index (though the other information is still accessible). This means that a particular target is referred to only by index (e.g., the user is seeing target i, not `foo.png`).

All these functions are implemented in Python, and we provide easy access other tasks needed for adaptive algorithms (database access, background jobs).

Arguments and returns: We treat each algorithm as a black box – NEXT only needs each algorithm function to accept and return specific values. These arguments and return values for all algorithm functions are specified exactly in a YAML-based schema. Every algorithm has to create a mapping from the specified inputs to the specified outputs.

NEXT verifies the inputs and output to/from algorithms and can also include a description of each parameter. This means that YAML schema is always up to date and is self-documenting. Changing this schema means different arguments are passed to every algorithm, and we offer flexibility by allowing arguments of any type to be passed.

This schema depends on `Algs.yaml` (e.g., in `apps/[application]/algs/Algs.yaml`) and contains four root level keys for each of `initExp`, `getQuery`, `processAnswer`, and `getModel`. Each one of these sections describes the input arguments and returns values by `args` and `rets` respectively. These sections are filled with type specifications that describe the name and type of the various keyword arguments.

For example, a particular `Algs.yaml` may include

```
getQuery:
  args:
    participant_uid:
```

7. Documentation can be found at https://github.com/nextml/NEXT/wiki

```
      type: string
      description: ID of the participant answering the query
  rets:
    description: The index of the target to ask about
    type: num
```

The keyword argument `participant_uid` is specified in the `args` key, and the return value must be a number. The corresponding `getQuery` implementation would be

```
def getQuery(butler, participant_uid):
    return 0  # for example
```

More complete documentation on these parameter specifications, which can be found at the API endpoint `assistant/doc/[application-name]/pretty`.

Database access: We provide a simple database wrapper, as algorithms need to store different values (e.g., the number of targets, a list of target scores). We provide a variety of atomic database operations through a thin wrappers to PyMongo[8] and Redis[9], though we can support arbitrary databases[10]. Each "collection" in this wrapper mirrors a Python dictionary and has several other atomic database operations. We provide

- `get`, `set` and `{get, set}_many` which provide atomic operations to store values in the database
- `append` and `pop`, which atomically modify list values, and return the result
- `increment`, which atomically increments a stored value by a given amount

All these operations are atomic, and can be accessed through an interface called `butler` which contains multiple collections. The primary collection used by algorithms (`butler.algorithms`) is specific to each algorithm and allows for independent evaluation of different algorithms (though other collections are available). The arguments to an algorithm function are `butler` followed by the values in the schema.

Example: This example illustrates the interface we have created for the algorithm developer and provides an example of algorithm implementation. After implementation, this algorithm can receive crowdsourcing responses through the web interface.

```
import numpy as np

def choose_target(butler):
    # Adaptive sampling hidden for brevity
    n = butler.algorithms.get(key='n')
    return np.random.choice(n)

class MyAlg:
    def initExp(self, butler, n):
        butler.algorithms.set(key='n', value=n)
        scores = {'score'+ str(i): 0 for i in range(n)}
        pulls = {'pulls' + str(i): 0 for i in range(n)}
        butler.algorithms.set_many(
            key_value_dict=scores
        )
        butler.algorithms.set_many(
            key_value_dict=pulls
        )

    def getQuery(self, butler):
        return choose_target(butler)

    def processAnswer(self, butler,
                      target_id, reward):
        butler.algorithms.increment(
            key='score' + str(target_id),
            value=reward
        )
        butler.algorithms.increment(
            key='pulls' + str(target_id),
        )

    def getModel(self, butler):
        n = butler.algorithms.get(key='n')
        scores = [butler.algorithms.get(
                    'score' + str(i))
                  for i in range(n)]
        pulls = [butler.algorithms.get(
                    'pulls' + str(i))
                 for i in range(n)]
        mean_scores = [s/p if p != 0 else float('nan')
                       for s, p in zip(scores, pulls)]
        return mean_scores
```

The `Algs.yaml` file for this algorithm would be

```
initExp:
  args:
    n:
      description: Number of targets
      type: num
getQuery:
  rets:
    type: num
    description: The target to show
                 the user
processAnswer:
  args:
    target_id:
      description: The target_id that was shown
                   to the user
      type: num
    reward:
      description: The reward the user gave
                   the target
      values: [1, 2, 3]
      type: num
getModel:
  rets:
    type: list
    description: The scores for each target ordered
                 by target_id.
    values:
      description: The mean score for a particular target
      type: num
```

Experiment dashboards

NEXT can be monitored in real-time via dashboards for each experiment, which include:

- experiment logs
- basic information (launch date, number of received responses, etc)
- the results, with current responses received (example in Figure 8)
- client- and server-side timing information
- download links to the responses and the live results (which allows processing of these data offline).

The dashboards include histograms for both human response time and network delay (time taken for NEXT to respond to request), a measure of system responsiveness. An example is shown in Figure 9. These dashboards also include timing information for algorithm functions, a useful debugging tool for the algorithm developer.

From the dashboard, we support the download of both experiment results and participant response information.

8. http://api.mongodb.com/python/current
9. https://redis.io/
10. Which requires implementation of the Collection API found in `next.apps.Butler`

Rankings

BR_LilUCB BR_Random

Rank	Target	Score	Precision
0	I'm drowning in work.	0.91667	0.20412
1	The women's office is the same, except it has glass on the top, too.	0.86957	0.20851
2	If you work hard they give you a rock and plastic seaweed.	0.78571	0.26726
3	Your seat cushion can be used as a flotation device.	0.75000	0.28868

Fig. 8: *The dashboard display of results from different algorithms for the example in Figure 6.*

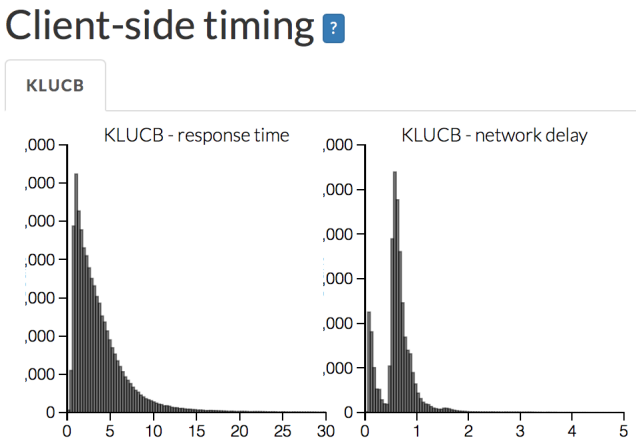


Fig. 9: *Timing histograms measured client-side in seconds for cartoon caption contest 573. Network delay represents the total time NEXT took to respond and response time measures human resposne time.*

Experimentalist use

Below, we will refer to different NEXT features which are available through different API endpoints. After NEXT has launched, these are available via HTTP on port `8000` on the hosting machine. In practice, this means the API endpoint `/home` (for example) is available at `[next-url]:8000/home` when `[next-url]` is one of `ec2-...-amazonaws.com` or `localhost`.

Launching NEXT: The easiest way to launch NEXT is through Amazon EC2 (which can provide the interface required for crowdsourcing) and their AMI service. After launch, the main NEXT interface is available at the API endpoint `/home` which provides links to the list of dashboards, an experiment launching interface and the associated documentation.

Launching can be done by selecting the "Launch instance" button on Amazon EC2 and choosing the AMI "NEXT_AMI", `ami-36a00c56` which is available in the Oregon region. We recommend that production experiments be run on the EC2 instance-type `c4.8xlarge`, a server large enough to provide the necessary memory and compute power. A complete guide can be found in the documentation at https://github.com/nextml/NEXT/wiki.

Experiment launch: Experiments are launched by providing two files to NEXT, either via a web interface or an API endpoint. An experiment description file is required. The other (optional) file enumerate the objects under consideration ("target"). These two files can be uploaded through the interface available at `/assistant/init`.

The experiment description contains the information required to launch and configure the experiment. The following experiment description was used to generate the image in Figure 6:

```
app_id: CardinalBanditsPureExploration
args:
  alg_list:
  - {alg_id: KLUCB, alg_label: KLUCB}
  algorithm_management_settings:
    mode: fixed_proportions
    params:
    - {alg_label: KLUCB, proportion: 1.0}
  context: # image URL, trimmed for brevity
  context_type: image
  failure_probability: 0.05
  participant_to_algorithm_management: one_to_many
  rating_scale:
    labels:
    - {label: unfunny, reward: 1}
    - {label: somewhat funny, reward: 2}
    - {label: funny, reward: 3}
```

These parameters are defined in schemes, and are documented at the API endpoint `/assistant/doc/[application-id]/pretty` in the "initExp" section.

The other file necessary for experiment launch is a ZIP file of targets (e.g., the images involved in each query). We support several different formats for this ZIP file so images, text and arbitrary URLs can be supported. If images are included in this ZIP file, we upload all images to Amazon S3.

Experimentalist use with crowdsourcing: After experiment launch, a link to the experiment dashboard and query page is presented. We recommend distributing this query page link to crowdsourcing participants, which typically happens via Mechanical Turk or email.

Experiment persistence: We support saving and restoring experiments on the experiment list at `/dashboard/experiment_list`. This allows experiment persistence even when Amazon EC2 machines are terminated.

Conclusion

At UW–Madison, we have created a system that is connecting useful adaptive algorithms with crowdsourced data collection. This system has been successfully used by experimentalists in a wide variety of disciplines from the social sciences to engineering to efficiently collect crowdsourced data; in effect, accelerating research by decreasing the time to obtain results.

The development of this system is modular: sampling algorithms are treated as black boxes, and this system is accessible with other interfaces. NEXT provides useful experiment monitoring tools that update as responses are received. This system has shown to be cost effective in bringing decision making tools to new applications in both the private and public sectors.

References

[AB10] Jean-Yves Audibert and Sébastien Bubeck. Best arm identification in multi-armed bandits. In *COLT-23th Conference on Learning Theory-2010*, pages 13–p, 2010.

[ABC+16] Alekh Agarwal, Sarah Bird, Markus Cozowicz, Luong Hoang, John Langford, Stephen Lee, Jiaji Li, Dan Melamed, Gal Oshri, Oswaldo Ribas, et al. A multiworld testing decision service. *arXiv preprint arXiv:1606.03966*, 2016.

[CBCTH13] Xi Chen, Paul N Bennett, Kevyn Collins-Thompson, and Eric Horvitz. Pairwise ranking aggregation in a crowdsourced setting. In *Proceedings of the sixth ACM international conference on Web search and data mining*, pages 193–202. ACM, 2013.

[CWN05] Rui Castro, Rebecca Willett, and Robert Nowak. Faster rates in regression via active learning. In *NIPS*, volume 18, pages 179–186, 2005.

[DDS+09] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, pages 248–255. IEEE, 2009.

[DHM08] Sanjoy Dasgupta, Daniel J Hsu, and Claire Monteleoni. A general agnostic active learning algorithm. In *Advances in neural information processing systems*, pages 353–360, 2008.

[GGL12] Victor Gabillon, Mohammad Ghavamzadeh, and Alessandro Lazaric. Best arm identification: A unified approach to fixed budget and fixed confidence. In *Advances in Neural Information Processing Systems*, pages 3212–3220, 2012.

[Hoa62] Charles AR Hoare. Quicksort. *The Computer Journal*, 5(1):10–16, 1962.

[Hol92] John H Holland. *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*. MIT press, 1992.

[HZRS15] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034, 2015.

[JJF+15] Kevin G Jamieson, Lalit Jain, Chris Fernandez, Nicholas J Glattard, and Rob Nowak. Next: A system for real-world development, evaluation, and application of active learning. In *Advances in Neural Information Processing Systems*, pages 2656–2664, 2015.

[JJN16] Lalit Jain, Kevin G Jamieson, and Rob Nowak. Finite sample prediction and recovery bounds for ordinal embedding. In *Advances in Neural Information Processing Systems*, pages 2711–2719, 2016.

[JMNB14] Kevin Jamieson, Matthew Malloy, Robert Nowak, and Sébastien Bubeck. lil'ucb: An optimal exploration algorithm for multi-armed bandits. In *Conference on Learning Theory*, pages 423–439, 2014.

[Jun16] Kwang-Sung Jun. Anytime exploration for multi-armed bandits using confidence information. In *Proceedings of The 33rd International Conference on Machine Learning*, pages 974–982, 2016.

[KK13] Emilie Kaufmann and Shivaram Kalyanakrishnan. Information complexity in bandit subset selection. In *COLT*, pages 228–251, 2013.

[KZP07] Sotiris B Kotsiantis, I Zaharakis, and P Pintelas. Supervised machine learning: A review of classification techniques, 2007.

[RMN16] Martina A Rau, Blake Mason, and Robert Nowak. How to model implicit knowledge? similarity learning methods to assess perceptions of visual representations. In *Proceedings of the 9th International Conference on Educational Data Mining*, 2016.

[SLJ+15] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–9, 2015.

[SZ14] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[YBKJ12] Yisong Yue, Josef Broder, Robert Kleinberg, and Thorsten Joachims. The k-armed dueling bandits problem. *Journal of Computer and System Sciences*, 78(5):1538–1556, 2012.

ChiantiPy: a Python package for Astrophysical Spectroscopy

Will T. Barnes[§*], Kenneth P. Dere[‡]

https://youtu.be/79ledxbnrPU

Abstract—ChiantiPy is an interface to the CHIANTI atomic database for astrophysical spectroscopy. The highly-cited CHIANTI project, now in its 20th year, is an invaluable resource to the solar physics community. The ChiantiPy project brings the power of the scientific Python stack to the CHIANTI database, allowing solar physicists and astronomers to easily make use of this atomic data and calculate commonly used quantities from it such as radiative loss rates and emissivities for particular atomic transitions. This paper will discuss the capabilities of the CHIANTI database and the ChiantiPy project as well as the current state of the project and its place in the solar physics community. We will demonstrate how the core modules in ChiantiPy can be used to study emission from optically thin transitions and the continuum in the x-ray and EUV wavelengths. Additionally, we will discuss some of the infrastructure around the ChiantiPy project and some of the goals for the near future.

Index Terms—solar physics, atomic physics, astrophysics, spectroscopy

Introduction

Nearly all astrophysical observations are done through *remote sensing*. Light at various wavelengths is collected by instruments, either ground- or space-based, in an attempt to understand physical processes happening in distant astrophysical objects. However, in order to translate these detector measurements to meaningful physical insight, it is necessary to understand what physical conditions give rise to different spectral lines and continuum emission. Started in 1996 by researchers at the Naval Research Laboratory, the University of Cambridge, and Arcetri Astrophysical Observatory in Florence for the purpose of analyzing solar spectra, the CHIANTI atomic database provides a set of up-to-date atomic data for thirty different elements as well as a suite of tools, written in the proprietary Interactive Data Language (IDL), for analyzing this data. Described in a series of 15 papers from 1997 to 2016 that have been cited collectively over 3000 times (see Table 1), the CHIANTI database is an invaluable resource to the solar physics community.

The CHIANTI project is comprised of two main parts: the database containing the actual atomic data and the IDL software libraries for accessing the data and calculating useful quantities from them. The database provides atomic data for optically-thin transitions, primarily in the x-ray and extreme ultraviolet (EUV) wavelengths, for ions of 30 different elements, H ($Z = 1$) through Zn ($Z = 30$). The CHIANTI project stemmed largely from the need for a consolidated database of spectral lines for interpreting data from spectroscopic and narrow-band solar observing instruments.

Paper	Version	Citations
[DLM+97]	1	1167
[YLT98]	1	105
[LLD+99]	2	94
[DLYDZ01]	3	156
[LFD02]	3	86
[YDZL+03]	4	250
[LDZY+06]	5	373
[LP06]	5	25
[DLY+09]	6	301
[LY09]	6	25
[YL09]	6	22
[LDZY+12]	7	174
[LYD+13]	7.1	227
[DZDY+15]	8	60
[YDL+16]	8	1
		Total
		3066

TABLE 1: *All publications describing the data and capabilities of the CHIANTI atomic database, the associated version of the database, and the number of citations as reported by the NASA Astrophysics Data System at the time of writing.*

While IDL has been the lingua franca of solar physics for over twenty years, Python is gaining momentum in the community and is the language of choice for many younger researchers. This is largely due to the success of Python in general astronomy (e.g. Astropy), the advent of SunPy, a stable and well-supported Python package for solar data analysis [SMC+15], and the adoption of Python as the language of choice by the Daniel K. Inouye Solar Telescope (DKIST), an instrument expected to be the world's largest solar telescope with an estimated data output of 11 TB per day [WBH+16].

Given the growing popularity of Python in the solar community and the importance of CHIANTI to solar observers and modelers alike, a well-supported Python interface to this database is critical. The ChiantiPy project, started in 2003 by Ken Dere, provides a Python package for interacting with the CHIANTI

* *Corresponding author: will.t.barnes@rice.edu*
§ *Department of Physics and Astronomy, Rice University, Houston, TX, USA*
‡ *Department of Physics and Astronomy, George Mason University, Fairfax, VA, USA*

Filetype	Description
ELVLC	Index and energy for each level
WGFA	Wavelength, radiative decay rates, and oscillator strengths for each transiton
SCUPS	Scaled effective collision strengths for each transition
FBLVL	Energy levels for free-bound continuum calculation

TABLE 2: *Some of the filetypes available for each ion in the CHIANTI database. Adapted from Table 1 of [YDL+16].*

database and an alternative to the IDL tools. ChiantiPy is not a direct translation of its IDL counterpart, but instead provides an intuitive object oriented interface to the database (compared to the more functional approach in IDL). ChiantiPy provides an easy to use API to the raw atomic data in the CHIANTI database as well as Python versions of all the primary calculations performed by the original IDL software, including the level balance equation and the ionization equilibrium calculation. This paper will give a brief overview of the CHIANTI database and demonstrate the core capabilities of the ChiantiPy package. These include the line emission for transitions and continuum emission of particular ions as well as spectra and radiative loss rates summed over many ions. We will also discuss the infrastructure of the package and plans for the future of the package.

Database

The CHIANTI database is organized as a collection of directories and ASCII files that can be downloaded as a tarball from the CHIANTI database website or as part of the SolarSoftware (or SolarSoft) IDL package [FH98]. The solar physics community has typically relied on the latter as SolarSoft has served as the main hub for solar data analysis software for the last several decades. SolarSoft provides routines for updating software packages automatically and so traditionally CHIANTI users have updated their distributions, including both the software and the database, in this manner[1].

The structure of the CHIANTI database is such that each top level directory represents an element and each subdirectory is an ion of that element. Files in each of the subdirectories contain pieces of information attached to each ion. The database generally follows the structure `{el}/{el}_{ion}/{el}_{ion}.{filetype}`, where `el` is the element, `ion` is the ion number, and `filetype` is the file extension. For example, the energy level information for Fe V is in the file `fe/fe_5/fe_5.elvlc`. A few of these filetypes are summarized in Table 2. For a complete description of all of the different filetypes available, see Table 1 of [YDL+16] and the CHIANTI user guide. Fig. 1 shows all of the available ions in the CHIANTI database as well as the number of levels available for each ion.

ChiantiPy provides several low-level functions for reading raw data directly from the CHIANTI database. For example, to find the energy of the emitted photon for each transition for Fe V (i.e. the fifth ionization state of iron), you would first read in level information for each transition for a given ion,

```
import ChiantiPy.tools.util as ch_util
fe5_wgfa = ch_util.wgfaRead('fe_5')
ilvl1 = np.array(fe5_wgfa['lvl1']) - 1
ilvl2 = np.array(fe5_wgfa['lvl2']) - 1
```

and then use the indices of the level to find the associated level energies in the ELVLC data,

```
fe5_elvlc = ch_util.elvlcRead('fe_5')
delta_energy = (np.array(fe5_elvlc['ecm'])[ilvl2]
                - np.array(fe5_elvlc['ecm'])[ilvl1])
```

where the associated energy levels are given in cm^{-1}. In general, these functions are only used internally by the core ChiantiPy objects. However, users who need access to the raw data may find them useful.

In addition to each of the files associated with each ion, CHIANTI also provides abundance and ionization equilibrium data for each *element* in the database. The elemental abundance, $N(X)/N(H)$ (i.e. the number of atoms of element X relative to the number of hydrogen atoms), in the corona and photosphere has been measured by many workers and these various measurements have been collected in the CHIANTI atomic database. For example, to read the abundance of Fe as measured by [FMS+92],

```
import ChiantiPy.tools.io as ch_io
ab = ch_io.abundanceRead('sun_coronal_1992_feldman')
fe_ab = abundance['abundance'][ch_util.el2z('Fe')-1]
```

As with the other CHIANTI data files, the abundance values are typically read internally and then exposed to the user through more abstract objects like the `ion` class so reading them in this way is not necessary. Similarly, the ionization equilibrium of each ion of each element is available as a function of temperature and various sets of ionization equilibria data can be used. More details about the ionization equilibrium can be found in later sections.

Default values for the abundance and ionization equilibrium files as well as the units for wavelength (nm, Å, or eV) and energy (ergs or photons) can be set in the users `chiantirc` file, located in `~/.chianti/chiantirc`. These settings are stored in `ChiantiPy.tools.data.Defaults` and can be changed at anytime.

Unless otherwise noted, all quantities are expressed in the cgs unit system, with the exception of wavelengths which are recorded in angstroms (Å). As discussed above, some energies in the CHIANTI atomic database, particularly those pertaining to levels in an atom, are stored in cm^{-1} for convenience (i.e. with $h = c = 1$, a common convention in atomic physics). Results of any calculation in ChiantiPy will always be returned in cgs units (unless explicitly stated in the `chiantirc` file, e.g. photons instead of ergs).

Common Calculations and API

The majority of the ChiantiPy codebase is divided into two modules: `tools` and `core`. The former contains utility and helper functions that are mostly for internal use. The latter contains the primary objects for interacting with the data in the CHIANTI atomic database and performing many common calculations with these data. A summary of the objects in `core` can be found in Table 3. These objects can be roughly divided into two categories: those that deal with information and calculations about individual ions and those that aggregate information over a range of ions in order to perform some calculation. The `ion` and `Continuum`

1. The easiest way to acquire the CHIANTI database is to download and unpack the tarball at http://www.chiantidatabase.org/chianti_download.html. In order for ChiantiPy to find the database, it is necessary to point the `XUVTOP` environment variable to the top of the CHIANTI directory tree. For example, if the database is downloaded to `$HOME/chianti`, `export XUVTOP=$HOME/chianti/dbase` should be placed in the Bash shell configuration file.

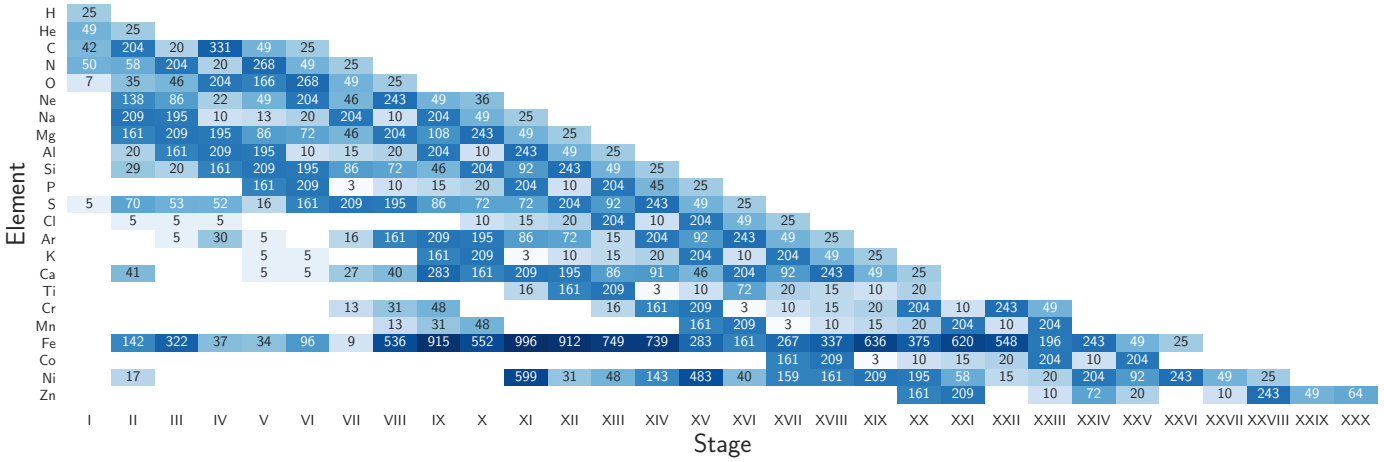


Fig. 1: *All ions available in the latest version (v8.0.6) of the CHIANTI atomic database. The color and number in each square indicate the number of available levels in the database. Adapted from Fig. 1 of [YDL+16].*

Object Name	Description
`ion`	Holds ion properties and calculates level populations and emissivity
`Continuum`	Free-free and free-bound continuum for individual ions
`ioneq`	Ionization equilibrium for individual elements
`spectrum`	Calculate synthetic spectra for a range of ions
`radLoss`	Total radiative losses from multiple ions, including continuum

TABLE 3: *The primary objects in the public API of ChiantiPy.*

objects calculate emissivity information related to specific ions while `ioneq`, `spectrum`, and `radLoss` require information from multiple ions and/or elements.

Line Emission

The most essential and actively developed portion of the ChiantiPy package is the `ion` object which provides an interface to the data and associated calculations for each ion in the database. The `ion` object is initialized with an ion name, a temperature range, and a density[2],

```
import ChiantiPy.core as ch
import numpy as np
temperature = np.logspace(4,6,100)
density = 1e9
fe_5 = ch.ion('fe_5',temperature,density)
```

In this example, we've initialized an `ion` object for Fe V over a temperature range of $T = 10^4 - 10^6$ K at a constant electron density of $n_e = 10^9$ cm^{-3}. All of the data discussed in the previous section are available as attributes of the `ion` object (e.g. `.Elvlc` and `.Wgfa` are dictionaries holding the various fields available in the corresponding filetypes listed in Table 3). In general, ChiantiPy objects follow the convention that methods are lowercase and return their value(s) to attributes with corresponding uppercase names[3]. For example, the abundance value of Fe is stored in `fe_5.Abundance` and the ionization equilibrium is calculated using the method `fe_5.ioneqOne()` with the value being returned to the attribute `fe_5.IoneqOne`.

One of the most often used calculations in CHIANTI and ChiantiPy is the energy level populations as a function of temperature. When calculating the energy level populations in a low density, high temperature, optically-thin plasma, collisional excitation and subsequent decay often occur much more quickly than ionization and recombination, allowing these two processes to be decoupled. Furthermore, it is assumed that all transitions occur between the excited state and the ground state. These two assumptions make up what is commonly known as the *coronal model approximation*. Thus, the level balance equation can be written as,

$$\sum_{k>j} N_k A_{kj} + n_e \sum_{i=j} N_j C_{ij} - \left(\sum_{i<j} N_j A ji + n_e \sum_{k=j} N_j C_{jk} \right) = 0,$$

where A_{kj} is the radiative decay rate, C_{jk} is the collisional excitation coefficient, and N_j is the number of electrons in excited state j [YDL+16]. Since A and C are given by the CHIANTI database, this expression can be solved iteratively to find $n_j = N_j / \sum_j N_j$, the fraction of electrons in excited state j or the level population fraction. Proton excitation rates, primarily between fine structure levels, can also be included in the calculation of n_j. See Eq. 4 of [YDZL+03].

The method `fe_5.populate()` can then be used to calculate the level populations for Fe V. This method populates the `fe_5.Population` attribute and a 100×34 array (i.e. number of temperatures by number of energy levels) is stored in `fe_5.Population['population']`. ChiantiPy also provides the convenience method `fe_5.popPlot()` which provides a quick visualization of level population as a function of temperature for several of the most populated levels. Note that this calculation can be quite expensive for large temperature/density arrays and for ions with many transitions. The left panel of Fig. 2 shows the level population as a function of temperature, $n_j(T)$, for all of the energy levels of Fe V in the CHIANTI database.

2. A single temperature and an array of densities is also valid. The only requirement is that if one or the other is not of length 1, both arrays must have the same length. The ion object can also be initialized without any temperature or density information if only the ion data is needed.

3. This convention is likely to change in the near future as the ChiantiPy codebase is brought into compliance with the PEP 8 Style Guide for Python code.

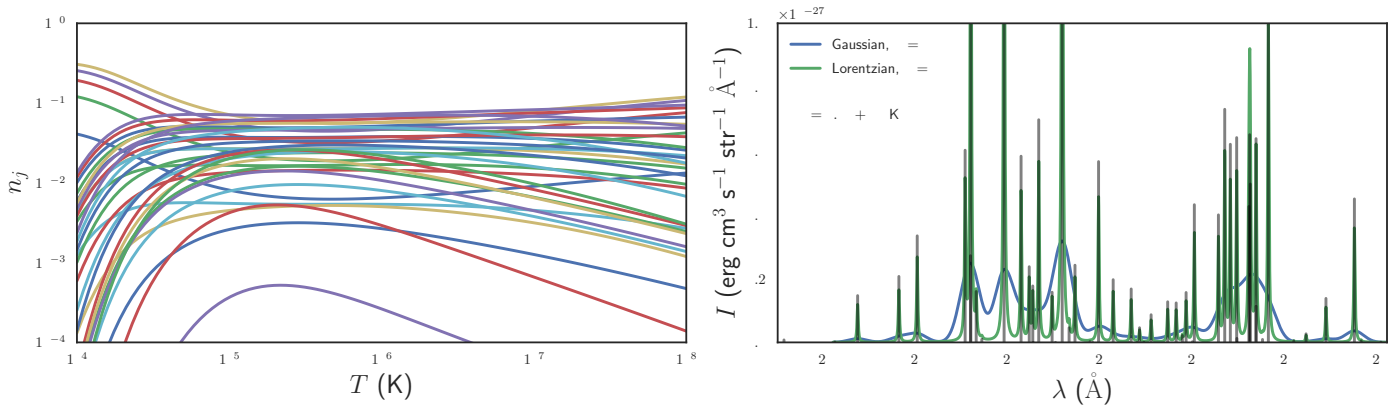


Fig. 2: *Level populations, n_j, as a function of temperature (left) and intensity as a function of wavelength (right) for Fe V. The various curves in the left panel represent the multiple energy levels of the Fe V ion. The right panel shows the intensity at the discrete wavelength values (black) as well as the spectra folded through a Gaussian filter with $\sigma = 5$ Å (blue) and a Lorentzian filter with $\gamma = 5$ Å (green).*

When dealing with spectral line emission, we are often most interested in the line *intensity*, that is, the power per unit volume as a function of temperature (and density). For a particular transition λ_{ij}, the line intensity can be written as,

$$I_{ij} = \frac{1}{4\pi}\frac{hc}{\lambda}\mathrm{Ab}(X)X_k A_{ij} n_j n_e^{-1}, \quad [\mathrm{erg\,cm^3\,s^{-1}\,sr^{-1}}]$$

where $\mathrm{Ab}(X)$ is the abundance and X_k is the ionization equilibrium. To calculate the intensity for each transition in CHIANTI for Fe V, we can use the method `fe_5.intensity()` which returns a 100×219 array (i.e. dimension of temperature by the number of available transitions). The convenience methods `fe_5.intensityPlot()` and `fe_5.intensityList()` can also be used to quickly visualize and enumerate the most intense lines produced by the ion, respectively.

Finally, the intensity can be convolved with a filter to calculate the intensity as a *continuous* function of wavelength to simulate an observed *spectrum*. For a single ion this is done using the `fe_5.spectrum()` method (see later sections for creating multi-ion spectra). To create a spectrum for Fe V between 2600 Å and 2900 Å,

```
wavelength = np.arange(2.6e3,2.9e3,0.1)
fe_5.spectrum(wavelength)
```

This method also accepts an optional keyword argument for specifying a filter with which to convolve the intensity. The default filter is a Gaussian though `ChiantiPy.tools.filters` includes several different filters including Lorentzian and Boxcar filters. The right panel of Fig. 2 shows the Fe V intensity (black) and spectrum folded through a Gaussian (blue) and Lorentzian (green) filter at the temperature at which the ionization fraction is maximized, $T \approx 8.5 \times 10^4$ K. Similar to the `fe_5.populate()` and `fe_5.intensity()`, ChiantiPy also provides the convenience method `fe_5.spectrumPlot()` for quickly visualizing a spectrum.

Continuum Emission

In addition to calculating emissivities for individual spectral lines, ChiantiPy also calculates the free-free, free-bound, and two-photon continua as a function of wavelength and temperature for each ion. In particular, the `Continuum` object is used to calculate the free-free and free-bound emissivities. Free-free emission (or *bremsstrahlung*) is produced by collisions between free electrons and positively charged ions. The free-free emissivity is given by,

$$\frac{dW}{dtdVd\lambda} = \frac{c}{3m_e}\left(\frac{\alpha h}{\pi}\right)^3\left(\frac{2\pi}{3m_e k_B}\right)^{1/2}\frac{Z^2}{\lambda^2 T^{1/2}}\bar{g}_{ff} \times \exp\left(-\frac{hc}{\lambda k_B T}\right), \quad [\mathrm{erg\,cm^3\,s^{-1}\,Å^{-1}\,sr^{-1}}]$$

where α is the fine structure constant, Z is the nuclear charge, T is the electron temperature, and $\bar{g}_{ff}$ is the velocity-averaged Gaunt factor [RL79]. $\bar{g}_{ff}$ is calculated using the methods of [ISK+00] (`Continuum.itoh_gaunt_factor()`) and [Sut98] (`Continuum.sutherland_gaunt_factor()`), depending on the temperature range.

Similarly, free-bound emission is produced when a free electron collides with a positively-charged ion and the previously-free electron is captured into an excited state of the ion. Because this process (unlike free-free emission) involves the details of the energy level structure of the ion, its formulation is necessarily quantum mechanical though a semi-classical treatment is possible (see Section 4.7.2 of [PFL08] and Section 10.5 of [RL79]). From [YDZL+03], the free-bound emission can be calculated as,

$$\frac{dW}{dtdVd\lambda} = \frac{1}{4\pi}\frac{2}{hk_B c^3 m_e\sqrt{2\pi k_B m_e}}\frac{E^5}{T^{3/2}}\sum_i \frac{\omega_i}{\omega_0}\sigma_i^{bf} \times \exp\left(-\frac{E-I_i}{k_B T}\right), \quad [\mathrm{erg\,cm^3\,s^{-1}\,Å^{-1}\,sr^{-1}}]$$

where $E = hc/\lambda$ is the photon energy, ω_i and ω_0 are the statistical weights of the i^{th} level of the recombined ion and the ground level of the recombining ion, respectively, σ_i^{bf} is the photoionization cross-section, and I_i is the ionization potential of level i. The cross-sections are calculated using the methods of [VY95] (for the ground state, i.e. $i = 0$) and [KL61] (for $i \neq 0$). An optional `use_verner` keyword argument (`True` by default) is included in the `Continuum.calclulate_free_bound_emission()` so that users can choose to only use the method of [KL61] in the photoionization cross-section calculation.

To calculate the free-free and free-bound emission with ChiantiPy,

```
temperature = np.logspace(6,8.5,100)
cont_fe18 = ch.Continuum('fe_18',temperature)
wavelength = np.logspace(0,3,100)
```

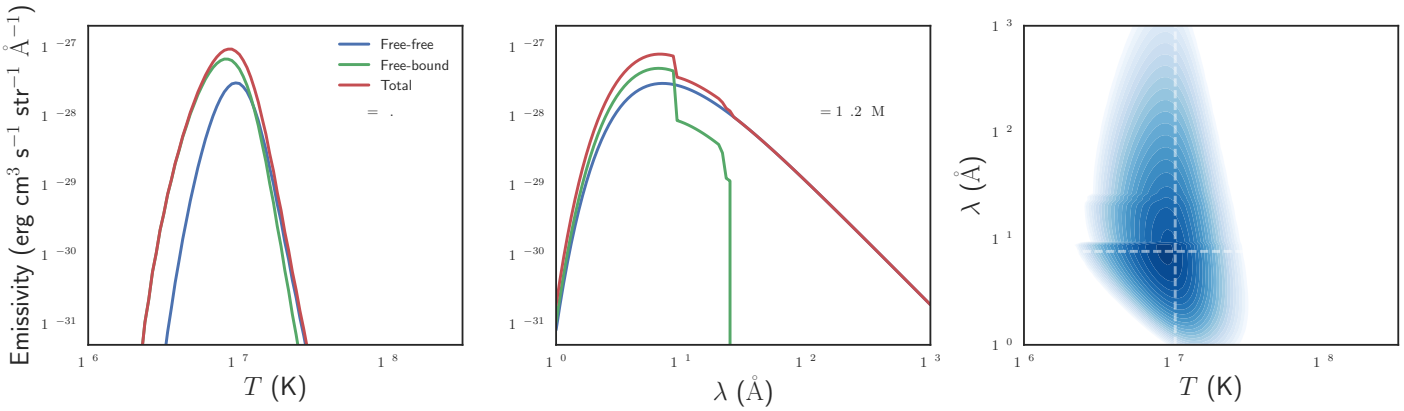


Fig. 3: *Continuum emission for Fe XVIII. The left (middle) panel shows the free-free, free-bound, and total emission as a function of temperature (wavelength) for $\lambda \approx 7.5$ Å ($T \approx 10^7$ K). The contours in the rightmost panel show the total emissivity as a function of both temperature and wavelength on a log scale. The dashed lines indicate the cuts shown in the left and middle panels.*

```
cont_fe18.calculate_free_free_emission(wavelength)
cont_fe18.calculate_free_bound_emission(wavelength)
```

The `Continuum.calculate_free_free_emission()` (`Continuum.calculate_free_bound_emission()`) method stores the N_T by N_λ array (e.g. in the above example, 100×100) in the `Continuum.free_free_emission` (`Continuum.free_bound_emission`) attribute. The left and middle panels of Fig. 3 show the free-free, free-bound, and total continuum emission as a function of temperature and wavelength, respectively, and the rightmost panel shows the total continuum emission as a function of both temperature and wavelength for the Fe XVIII ion.

The `Continuum` object also provides methods for calculating the free-free and free-bound radiative losses (i.e. the wavelength-integrated emission). These methods are primarily used by the `radiativeLoss` module. The `Continuum` module has recently been completely refactored and validated against the corresponding IDL results.

A contribution from the two-photon continuum can also be calculated with ChiantiPy though this is included in the `ion` object through the method `ion.twoPhoton()`. The two-photon continuum calculation is included in the `ion` object and not the `Continuum` object because the level populations are required when calculating the two-photon emissivity. See Eq. 11 of [YDZL+03].

Ionization Equilibrium

The ionization equilibrium of a particular ion describes what fraction of the ions of an element are in a particular ionization state at a given temperature. Specifically, the ionization equilibrium is determined by the balance of ionization and recombination rates. For an element X and an ionization state i, assuming ionization equilibrium, the ionization state $X_i = N(X^{+i})/N(X)$ is given by,

$$I_{i-1}X_{i-1} + R_i X_{i+1} = I_i X_i + R_{i-1} X_i$$

where I_i and R_i are the total ionization and recombination rates for ionization state i, respectively. In CHIANTI, these rates are assumed to be density-independent and only a function of temperature.

In ChiantiPy, the ionization equilibrium for a particular element can be calculated using the `ioneq` module,

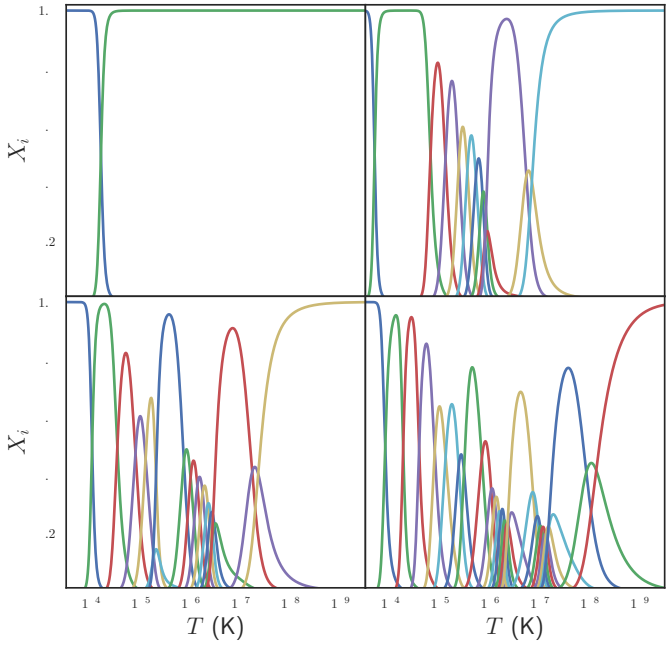


Fig. 4: *Population fractions as a function of temperature for (clockwise from upper left) H, Na, Fe, and S calculated using ionization and recombination data from CHIANTI and assuming ionization equilibrium.*

```
fe_ioneq = ch.ioneq('Fe')
temperature = np.logspace(3.5,9.5,500)
fe_ioneq.calculate(temperature)
```

The `ioneq.calculate()` method sets the `Ioneq` attribute, an array with $Z+1$ columns and N_T rows, where N_T is the length of the temperature array. In the example above, `fe_ioneq.Ioneq` has 27 rows (i.e. $Z = 26$ for Fe) and 500 columns. Fig. 4 shows the ion population fractions for four different elements as a function of temperature, assuming ionization equilibrium.

The `ioneq` module also allows the user to load a predefined set of ionization equilibria via the `ioneq.load()` method. Though CHIANTI includes several ionization equilibrium datasets from other workers, it is recommended to use the most up to date version supplied by CHIANTI (see [DLY+09] for more details).

To load the ionization equilibrium data for Fe,

```
fe_ioneq = ch.ioneq('Fe')
fe_ioneq.load()
```

This will populate the `fe_ioneq.Temperature` and `fe_ioneq.Ioneq` attributes with data from the appropriate ionization equilibrium file. By default, this will be `ioneq/chianti.ioneq` unless otherwise specified in the `chiantirc` file or the `ioneqName` keyword argument.

Spectra

In addition to being able to calculate spectra for single ions, ChiantiPy also provides a wrapper for calculating composite spectra for a range of ions, including continuum contributions. This is handled through the `spectrum` object. To calculate a composite spectrum in ChiantiPy,

```
temperature = np.array([1e+6,4e+6,1e+7])
density = 1e9
wavelength = np.linspace(10,100,1000)
min_abund = 1e-4
spec = ch.spectrum(temperature, density,
                   wavelength, minAbund=min_abund)
```

The spectrum as a continuous function of wavelength can then be accessed in the `spec.Spectrum['intensity']` attribute as a $N_T \times N_\lambda$ array (i.e. 3×1000 in the above example. Most of the keywords that can be passed to `ion.spectrum()` can also be passed to `ChiantiPy.spectrum()` and the attributes that are available following the calculation are largely the same. Fig. 5 shows the integrated spectrum as calculated above with several of the included transitions labeled.

Because of the need to perform calculations and aggregate data over a large range of ions, running `ChiantiPy.spectrum()` can be very time consuming, particularly for large temperature/density ranges. The above code snippet takes approximately five minutes to execute on a modern desktop machine. To help mitigate this difficulty, ChiantiPy provides a parallelized version of the `ChiantiPy.spectrum` module called `ChiantiPy.mspectrum`[4] which takes advantage of the `multiprocessing` package and can help to speed up the calculation, particularly on machines with many cores. The interface to the parallelized code is largely the same as the serial version.

Radiative Losses

The radiative loss rate is an important quantity for calculating the energy loss in coronal plasmas, particularly in hydrodynamic simulations of coronal loops. The total radiative loss rate is given by,

$$\Lambda = \Lambda_{continuum} + \Lambda_{line}, \quad [\mathrm{erg\,cm^3\,s^{-1}}]$$

where

$$\Lambda_{line} = \sum_X \Lambda_X = \sum_{X,k} \Lambda_{X_k} = \sum_{X,k,\lambda_{ij}} \Lambda_{X_{k,\lambda_{ij}}}$$
$$= \sum_{X,k,\lambda_{ij}} \mathrm{Ab}(X) X_k \frac{hc}{\lambda} A_{ij} n_j n_e^{-1},$$

is the contribution to the radiative losses summed over every element (X), ion (X_k) and transition (λ_{ij}), and $\Lambda_{continuum}$ includes the free-free, free-bound, and two-photon continuum contributions to the radiative loss.

4. ChiantiPy provides an additional module `ChiantiPy.ipymspectrum` to support parallelized spectrum calculations inside the Jupyter notebook and qtconsole.

In ChiantiPy, the radiative loss rate can be calculated using the `radLoss` module for a particular temperature and density range. To calculate the total radiative loss rate for all ions with an abundance greater than 10^{-4},

```
temperature = np.logspace(4,8,100)
rl = ch.radLoss(temperature, 1e9, minAbund=1e-4)
```

Instantiating the `radLoss` object automatically calculates the radiative loss rate and stores the total loss rate in `rl.RadLoss['rate']`, in this case an array of length 100. If the continuum contributions are included (`doContinuum` is `True` by default), the free-free, free-bound, and two-photon components are stored in `rl.FreeFreeLoss`, `rl.FreeBoundLoss`, and `rl.TwoPhotonLoss`, respectively. Ions with low abundances can be excluded with the `minAbund` keyword argument which can speed up the calculation. A custom abundance dataset can also be set with the `abundance` keyword. Note that the above calculation takes approximately 11 minutes on modern hardware. Fig. 6 shows the total radiative losses using the coronal abundances of [FMS+92] (solid) and the photospheric abundances of [AGSS09] (dashed). The coronal abundance case is also broken down into the line emission, free-free, free-bound, and two-photon continuum components.

Documentation, Testing, and Infrastructure

The ChiantiPy project has made an effort to embrace modern development practices when it comes to developing, documenting and releasing the ChiantiPy codebase. Like many open source projects started in the late 2000s, ChiantiPy was originally hosted on SourceForge, but has now moved its development entirely to GitHub. The SVN commit history is in the process of being migrated to GitHub as well. The move to GitHub has provided increased development transparency, ease of contribution, and better integration with third-party services.

An integral part of producing quality scientific code, particularly that meant for a large user base, is continually testing said code as improvements are made and features are added. For each merge into master as well as each pull request, a series of tests is run on Travis CI, a continuous integration service that provides free and automated builds configured through GitHub webhooks. This allows each contribution to the codebase to be tested to ensure that these changes do not break the codebase in unexpected ways. Currently, ChiantiPy is tested on Python 2.7, 3.4, and 3.5, with full 3.6 support expected soon. Currently, the ChiantiPy package is installed in each of these environments and minimal set of tests of each core module is run. The actual module tests are currently quite sparse though one of the more pressing goals of the project is to increase test coverage of the core modules.

One of the most important parts of any codebase is the documentation. The ChiantiPy documentation is built using Sphinx and is hosted on Read the Docs. At each merge into the master branch, a new Read the Docs build is kicked off, ensuring that the ChiantiPy API documentation is never out of date with the most recent check in. In addition to the standard API documentation, the ChiantiPy Read the Docs page also provides a tutorial for using the various modules in ChiantiPy as well as a guide for those switching from the IDL version.

ChiantiPy has benefited greatly from the astropy-helpers package template provided by the Astropy collaboration [ART+13].

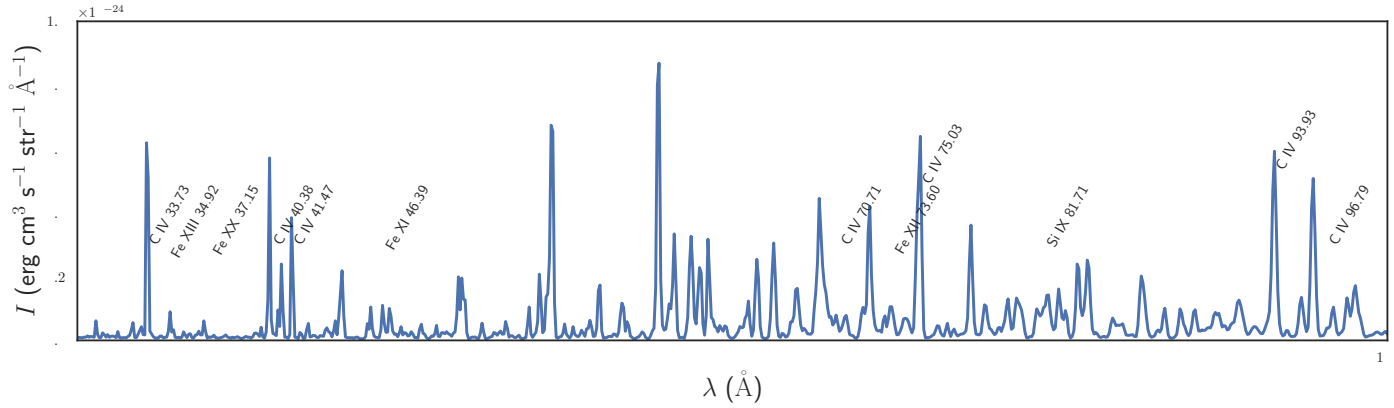


Fig. 5: *Total spectrum for all ions with an abundance greater than 10^{-4}, including the continuum, integrated over three temperatures, $T = 10^6, 4\times10^6, 10^7$ K and at a constant density of $n = 10^9$ cm^{-3}. A few of the transitions included in the spectrum are denoted by their respective spectroscopic labels and wavelengths.*

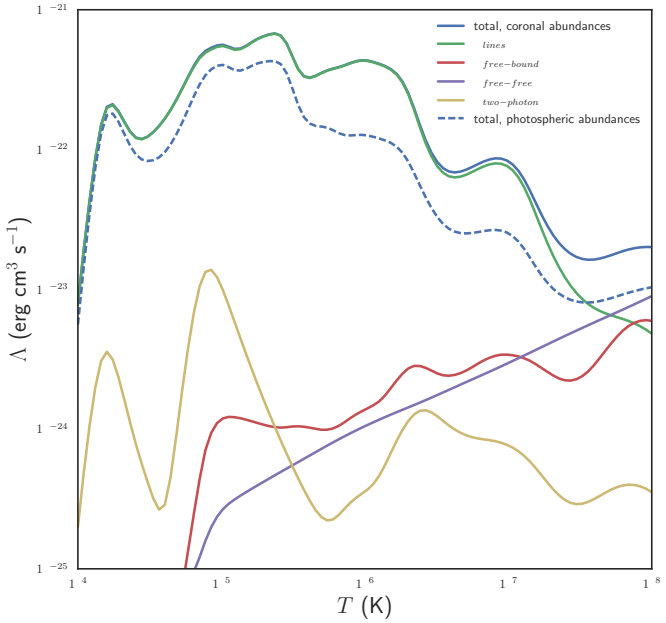


Fig. 6: *Combined radiative losses for all ions in the CHIANTI database for coronal abundances (blue,solid) and photospheric abundances (blue, dashed). The coronal abundance case is also broken down into the line emission (green) and free-free (purple), free-bound (red), and two-photon (yellow) continuum components. In the coronal case, the minimum abundance for elements to be included in the calculation is 10^{-4} and 10^{-6} for the photospheric case.*

asropy-helpers provides boilerplate code for setting up documentation and testing frameworks which has allowed the package to adopt modern testing and documentation practices with little effort.

Conclusion

In this paper, we have described the main capabilities of ChiantiPy, a package for astrophysical spectroscopy and an interface to the widely used and highly cited CHIANTI atomic database. ChiantiPy provides basic functions for reading the raw data as well as higher-level abstractions (e.g. the `ion` class) for exploring the data and performing common calculations with them. ChiantiPy also provides modules for calculating continuum emission, synthesizing spectra, and calculating radiative loss curves. The project has recently made significant infrastructure improvements by moving development to GitHub, adding automatic documentation builds, and implementing a minimal test suite. Future improvements include the addition of unitful quantities throughout the codebase (e.g. the Astropy unit system) and increased test coverage.

References

[AGSS09] Martin Asplund, Nicolas Grevesse, A. Jacques Sauval, and Pat Scott. The Chemical Composition of the Sun. *Annual Review of Astronomy and Astrophysics*, 47:481–522, September 2009. doi:10.1146/annurev.astro.46.060407.145222.

[ART+13] Astropy Collaboration, Thomas P. Robitaille, Erik J. Tollerud, Perry Greenfield, Michael Droettboom, Erik Bray, Tom Aldcroft, Matt Davis, Adam Ginsburg, Adrian M. Price-Whelan, Wolfgang E. Kerzendorf, Alexander Conley, Neil Crighton, Kyle Barbary, Demitri Muna, Henry Ferguson, Frédéric Grollier, Madhura M. Parikh, Prasanth H. Nair, Hans M. Unther, Christoph Deil, Julien Woillez, Simon Conseil, Roban Kramer, James E. H. Turner, Leo Singer, Ryan Fox, Benjamin A. Weaver, Victor Zabalza, Zachary I. Edwards, K. Azalee Bostroem, D. J. Burke, Andrew R. Casey, Steven M. Crawford, Nadia Dencheva, Justin Ely, Tim Jenness, Kathleen Labrie, Pey Lian Lim, Francesco Pierfederici, Andrew Pontzen, Andy Ptak, Brian Refsdal, Mathieu Servillat, and Ole Streicher. Astropy: A community Python package for astronomy. *Astronomy and Astrophysics*, 558:A33, October 2013. doi:10.1051/0004-6361/201322068.

[DLM+97] K. P. Dere, E. Landi, H. E. Mason, B. C. Monsignori Fossi, and P. R. Young. CHIANTI - an atomic database for emission lines - I. Wavelengths greater than 50 Å. *Astronomy and Astrophysics Supplement Series*, 125(1):25, 1997. doi:10.1051/aas:1997368.

[DLY+09] K. P. Dere, E. Landi, P. R. Young, G. Del Zanna, M. Landini, and H. E. Mason. CHIANTI - an atomic database for emission lines. IX. Ionization rates, recombination rates, ionization equilibria for the elements hydrogen through zinc and updated atomic data. *Astronomy and Astrophysics*, 498:915–929, May 2009. doi:10.1051/0004-6361/200911712.

[DLYDZ01] K. P. Dere, E. Landi, P. R. Young, and G. Del Zanna. CHIANTI-An Atomic Database for Emission Lines. IV. Extension to X-Ray Wavelengths. *The Astrophysical Journal Supplement Series*, 134:331–354, June 2001. doi:10.1086/320854.

[DZDY+15] G. Del Zanna, K. P. Dere, P. R. Young, E. Landi, and H. E. Mason. CHIANTI - An atomic database for emission lines. Version 8. *Astronomy and Astrophysics*, 582:A56, October 2015. doi:10.1051/0004-6361/201526827.

[FH98] S. L. Freeland and B. N. Handy. Data Analysis with the SolarSoft System. *Solar Physics*, 182:497–500, October 1998. doi:10.1023/A:1005038224881.

[FMS+92] U. Feldman, P. Mandelbaum, J. F. Seely, G. A. Doschek, and H. Gursky. The potential for plasma diagnostics from stellar extreme-ultraviolet observations. *The Astrophysical Journal Supplement Series*, 81:387–408, July 1992. doi:10.1086/191698.

[ISK+00] Naoki Itoh, Tsuyoshi Sakamoto, Shugo Kusano, Satoshi Nozawa, and Yasuharu Kohyama. Relativistic Thermal Bremsstrahlung Gaunt Factor for the Intracluster Plasma. II. Analytic Fitting Formulae. *The Astrophysical Journal Supplement Series*, 128:125–138, May 2000. doi:10.1086/313375.

[KL61] W. J. Karzas and R. Latter. Electron Radiative Transitions in a Coulomb Field. *The Astrophysical Journal Supplement Series*, 6:167, May 1961. doi:10.1086/190063.

[LDZY+06] E. Landi, G. Del Zanna, P. R. Young, K. P. Dere, H. E. Mason, and M. Landini. CHIANTI-An Atomic Database for Emission Lines. VII. New Data for X-Rays and Other Improvements. *The Astrophysical Journal Supplement Series*, 162:261–280, January 2006. doi:10.1086/498148.

[LDZY+12] E. Landi, G. Del Zanna, P. R. Young, K. P. Dere, and H. E. Mason. CHIANTI—An Atomic Database for Emission Lines. XII. Version 7 of the Database. *The Astrophysical Journal*, 744:99, January 2012. doi:10.1088/0004-637X/744/2/99.

[LFD02] E. Landi, U. Feldman, and K. P. Dere. CHIANTI-An Atomic Database for Emission Lines. V. Comparison with an Isothermal Spectrum Observed with SUMER. *The Astrophysical Journal Supplement Series*, 139:281–296, March 2002. doi:10.1086/337949.

[LLD+99] E. Landi, M. Landini, K. P. Dere, P. R. Young, and H. E. Mason. CHIANTI - an atomic database for emission lines. III. Continuum radiation and extension of the ion database. *Astronomy and Astrophysics Supplement Series*, 135:339–346, March 1999. doi:10.1051/aas:1999449.

[LP06] E. Landi and K. J. H. Phillips. CHIANTI-An Atomic Database for Emission Lines. VIII. Comparison with Solar Flare Spectra from the Solar Maximum Mission Flat Crystal Spectrometer. *The Astrophysical Journal Supplement Series*, 166:421–440, September 2006. doi:10.1086/506180.

[LY09] E. Landi and P. R. Young. CHIANTI—An Atomic Database for Emission Lines. X. Spectral Atlas of a Cold Feature Observed with Hinode/EUV Imaging Spectrometer. *The Astrophysical Journal*, 706:1–20, November 2009. doi:10.1088/0004-637X/706/1/1.

[LYD+13] E. Landi, P. R. Young, K. P. Dere, G. Del Zanna, and H. E. Mason. CHIANTI—An Atomic Database for Emission Lines. XIII. Soft X-Ray Improvements and Other Changes. *The Astrophysical Journal*, 763:86, February 2013. doi:10.1088/0004-637X/763/2/86.

[PFL08] Kenneth J. H. Phillips, Uri Feldman, and Enrico Landi. *Ultraviolet and X-Ray Spectroscopy of the Solar Atmosphere*. June 2008.

[RL79] George B. Rybicki and Alan P. Lightman. *Radiative Processes in Astrophysics*. New York : Wiley, 1979.

[SMC+15] SunPy Community, Stuart J. Mumford, Steven Christe, David Pérez-Suárez, Jack Ireland, Albert Y. Shih, Andrew R. Inglis, Simon Liedtke, Russell J. Hewett, Florian Mayer, Keith Hughitt, Nabil Freij, Tomas Meszaros, Samuel M. Bennett, Michael Malocha, John Evans, Ankit Agrawal, Andrew J. Leonard, Thomas P. Robitaille, Benjamin Mampaey, Jose Iván Campos-Rozo, and Michael S. Kirk. SunPy—Python for solar physics. *Computational Science and Discovery*, 8:014009, January 2015. doi:10.1088/1749-4699/8/1/014009.

[Sut98] Ralph S. Sutherland. Accurate free-free Gaunt factors for astrophysical plasmas. *Monthly Notices of the Royal Astronomical Society*, 300:321–330, October 1998. doi:10.1046/j.1365-8711.1998.01687.x.

[VY95] D. A. Verner and D. G. Yakovlev. Analytic FITS for partial photoionization cross sections. *Astronomy and Astrophysics Supplement Series*, 109, January 1995.

[WBH+16] Fraser T. Watson, Steven J. Berukoff, Tony Hays, Kevin Reardon, Daniel J. Speiss, and Scott Wiant. Calibration development strategies for the Daniel K. Inouye Solar Telescope (DKIST) data center. volume 9910, pages 99101G–99101G–11, 2016. doi:10.1117/12.2233179.

[YDL+16] P. R. Young, K. P. Dere, E. Landi, G. Del Zanna, and H. E. Mason. The CHIANTI atomic database. *Journal of Physics B: Atomic, Molecular and Optical Physics*, 49(7):074009, 2016. doi:10.1088/0953-4075/49/7/074009.

[YDZL+03] P. R. Young, G. Del Zanna, E. Landi, K. P. Dere, H. E. Mason, and M. Landini. CHIANTI-An Atomic Database for Emission Lines. VI. Proton Rates and Other Improvements. *The Astrophysical Journal Supplement Series*, 144:135–152, January 2003. doi:10.1086/344365.

[YL09] P. R. Young and E. Landi. Chianti—An Atomic Database for Emission Lines. XI. Extreme-Ultraviolet Emission Lines of Fe VII, Fe VIII, and Fe IX Observed by Hinode/EIS. *The Astrophysical Journal*, 707:173–192, December 2009. doi:10.1088/0004-637X/707/1/173.

[YLT98] P. R. Young, E. Landi, and R. J. Thomas. CHIANTI: An atomic database for emission lines. II. Comparison with the SERTS-89 active region spectrum. *Astronomy and Astrophysics*, 329:291–314, January 1998.