Proceedings of the 22nd Python in Science Conference

PROCEEDINGS OF THE 22ND PYTHON IN SCIENCE CONFERENCE

Edited by Meghann Agarwal, Chris Calloway, and Dillon Niederhut.

SciPy 2023
Austin, Texas
July 10 - July 16, 2023

ISSN:2575-9752
https://doi.org/10.25080/gerudo-f2bc6f59-039

ORGANIZATION

Conference Chairs

ALEXANDRE CHABOT-LECLERC, Enthought, Inc.
JULIE HOLLEK, Mozilla

Program Chairs

PAUL IVANOV, Citadel
MADICKEN MUNK, University of Illinois
GUEN PRAWIROATMODJO, Microsoft Corp

Communications

ARLISS COLLINS, NumFOCUS
JUANITA GOMEZ, Scientific Python
SAMINA TRACHIER, NumFOCUS

Birds of a Feather

ANDREW REID, NIST
MIKE HEARNE, USGS

Proceedings

MEGHANN AGARWAL,
CHRIS CALLOWAY, University of North Carolina
DILLON NIEDERHUT, Novi Labs

Financial Aid

SCOTT COLLIS, Argonne National Laboratory
ERIC MA, Moderna
NADIA TAHIRI, Université de Montréal

Tutorials

TETSUO KOYAMA, ARK Information Systems
LOGAN THOMAS, Enthought, Inc.
SOPHIA YANG, Anaconda

Sprints

TANIA ALLARD, Quansight Labs
ALAN BRAZ, IBM
BRIGITTA SIPŐCZ, Caltech/IPAC

Diversity

SARAH KAISER, Microsoft
BONNY P MCCLAIN, O'Reilly Media

Activities

PAUL ANZEL, Sentry
ED ROGERS, Majesco

Sponsors/Financial/Logistics

JIM WEISS, NumFOCUS

Hybrid

DAVID NICHOLSON, Embedded Intelligence
NEELIMA PULAGAM, Ford Motor Company

Proceedings Reviewers

David Nicholson
Pushkar Sathe
Shubhanshu Mishra
Jonny Saunders
Fernando Julian Chaure
Talha Irfan
Chitaranjan Mahapatra
Shaswat Shah
Ankit Shrivastava
Yuanqing Wang
Sanhita Joshi
Matthew Feickert
Nadia Tahiri
Hongsup Shin
Jim Pivarski
Tetsuo Koyama
Luis Medina Uzcategui
Jyotika Singh
Tolulade Ademisoye
Tanya Akumu
Aditya Singh
Olivia Dizon-Paradis
Saul Shanabrook
Brian McDermott
Kuntao Zhao
Maximilien Colange
John Derek Morgan
Kalyan Prasad
Amey Ambade

Accepted Talk Slides

Fast Exploration of the Milky Way (or any other n-dimensional dataset), Francesc Alted
doi.org/10.25080/gerudo-f2bc6f59-025
Better (Open Source) Homes and Gardens with Project Pythia, Drew Camron, and Kevin Tyle
doi.org/10.25080/gerudo-f2bc6f59-026
Accessibility best practices for authoring Jupyter notebooks, Isabela Presedo-Floyd, and Stephannie Jimenez Gacha
doi.org/10.25080/gerudo-f2bc6f59-027
Gammapy: a Python package for gamma-ray astronomy, Axel Donath, and The Gammapy Developer Team (https://gammapy.org/team.html)
doi.org/10.25080/gerudo-f2bc6f59-028
Python Array API Standard: Toward Array Interoperability in the Scientific Python Ecosystem, Aaron Meurer
doi.org/10.25080/gerudo-f2bc6f59-029
bayes_mapvar: Bayesian Statistics with Python, No Resampling Necessary, Charles Lindsey
doi.org/10.25080/gerudo-f2bc6f59-02a
New CUDA Toolkit packages for Conda, Rick Ratzel, and Thomson Comer, and John Kirkham
doi.org/10.25080/gerudo-f2bc6f59-02b
Taming Black Swans: Long-tailed distributions in the natural and engineered world, Allen B. Downey
doi.org/10.25080/gerudo-f2bc6f59-02c
In-Process Analytical Data Management with DuckDB, Alexander Monahan, and Hannes Mülheisen, and Mark Raasveldt, and Pedro Holanda
doi.org/10.25080/gerudo-f2bc6f59-02d
DataJoint: Bringing databases back into data science, Raphael Guzman, and Dimitri Yatsenko
doi.org/10.25080/gerudo-f2bc6f59-02e
Accelerating the Use of Public Geohysical Data for Recharging California's Groundwater, Seogi Kang, and Steve Purves
doi.org/10.25080/gerudo-f2bc6f59-02f
Interactive Exploration of Large-Scale Datasets with Jupyter-Scatter, Fritz Lekschas
doi.org/10.25080/gerudo-f2bc6f59-030
vak: a neural network framework for researchers studying animal acoustic communication , David Nicholson, and Yarden Cohen
doi.org/10.25080/gerudo-f2bc6f59-031
Open Force Field: next-generation force fields with open data, open software, and open science, Jeffrey Wagner
doi.org/10.25080/gerudo-f2bc6f59-032
Pandera: Going Beyond Pandas DataFrame Validation, Niels Bantilan
doi.org/10.25080/gerudo-f2bc6f59-033
Tidy geospatial data cubes, Emma Marshall, and Deepak Cherian, and Scott Henderson
doi.org/10.25080/gerudo-f2bc6f59-034
Zarr: Community specification of large, cloud-optimised, N-dimensional, typed array storage, Sanket Verma, and Josh Moore, and John Kirkham
doi.org/10.25080/gerudo-f2bc6f59-035

Accepted Posters

Unleashing the Power of Modern Portfolio Theory: Maximizing Returns while Managing Risk, Kalyan Prasad
doi.org/10.25080/gerudo-f2bc6f59-015
Data engineering and analytics for photolithography manufacturing process at DuPont - A practical approach from lab to fab, Avishek Panigrahi, and Stefan J Caporale, and Abhishek Shrivastava, and Sumanth Sekar
doi.org/10.25080/gerudo-f2bc6f59-016
EEG-to-fMRI Neuroimaging Cross Modal Synthesis in Python, David Calhas
doi.org/10.25080/gerudo-f2bc6f59-017
Hamilton: Scalable, Portable, and Self-Documenting Dataflows in Python, Stefan Krawczyk, and Elijah ben Izzy, and Levi Sweet-Breu, and Emily Rexer, and Chris Vernon, and Melissa Allen-Dumas
doi.org/10.25080/gerudo-f2bc6f59-018
itk-elastix: Medical image registration in Python, Konstantinos Ntatsis, and Niels Dekker, and Viktor van der Valk, and Tom Birdsong, and Dženan Zukić, and Stefan Klein, and Marius Staring, and Matthew McCormick
doi.org/10.25080/gerudo-f2bc6f59-019
Spatial Microsimulation and Activity Allocation in Python: An Update on the Likeness Toolkit, Joseph V. Tuccillo, and James D. Gaboardi
doi.org/10.25080/gerudo-f2bc6f59-01a
Matchmaker: A Toolkit for Combining Satellite Observations from Multiple Sensors, Greg Quinn
doi.org/10.25080/gerudo-f2bc6f59-01b
Patterns and Anti-Patterns when Measuring Diversity in Open Source, amanda casari
doi.org/10.25080/gerudo-f2bc6f59-01c

PyQtGraph - High Performance Visualization for All Platforms, Ognyan Moore, and Nathan Jessurun, and Nils Nemitz, and Martin Chase, and Luke Campagnola
doi.org/10.25080/gerudo-f2bc6f59-01d
PyVista, Tetsuo Koyama
doi.org/10.25080/gerudo-f2bc6f59-01e
OpenCRUMS: Open Classification of Regimes in the Southeast USA, Robert Jackson, and Maria Zawadowicz, and Die Wang, and Chongai Kuang, and Minnie Park, and Michael Jensen, and Scott Collis
doi.org/10.25080/gerudo-f2bc6f59-01f
Analyse the uncertainty of your system: Sensitivity Analysis in Python with scipy.stats.sobol_indices, Pamphile T. Roy
doi.org/10.25080/gerudo-f2bc6f59-020
aPhyloGeo-Covid: A Web Interface for Reproducible Phylogeographic Analysis of SARS-CoV-2 Variation using Neo4j and Snakemake, Wanlin Li, and Nadia Tahiri
doi.org/10.25080/gerudo-f2bc6f59-021
Moving the Earth with thermodynamics and python, Cian Wilson, and Marc Spiegelman, and Owen Evans, and Mark Ghiorso, and Lucy Tweed
doi.org/10.25080/gerudo-f2bc6f59-022
TUG-RSE: Pulling Students into Research Software Engineering, Aman Goel
doi.org/10.25080/gerudo-f2bc6f59-023
Yori: a new, highly customizable tool for Level-3 data production, Paolo Veglio, and Robert Holz, and Liam Gumley, and Steve Dutcher, and Greg Quinn, and Bruce Flynn
doi.org/10.25080/gerudo-f2bc6f59-024

SciPy Tools Plenaries

SciPy Tools Plenary on Matplotlib, Elliott Sales de Andrade
doi.org/10.25080/gerudo-f2bc6f59-036
SciPy Tools Plenary on SciPy, Pamphile T. Roy
doi.org/10.25080/gerudo-f2bc6f59-037
Zarr Updates for SciPy 2023, Josh Moore
doi.org/10.25080/gerudo-f2bc6f59-038

Lightning Talks

NumFOCUS Academic Consortium and Open Source Pledge, Arliss Collins
doi.org/10.25080/gerudo-f2bc6f59-013
Hamilton: drop procedural scripts in favor of declarative functions, Stefan Krawczyk
doi.org/10.25080/gerudo-f2bc6f59-014

Scholarship Recipients

Arnaud Kayonga,
Caitlin Lewis,
Emanuel Lima,
Gajendra Deshpande,
Micaela Matta,
Noa Tamir,
Tetsuo Koyama,
Wanlin Li,

CONTENTS

Using Blosc2 NDim As A Fast Explorer Of The Milky Way (Or Any Other NDim Dataset)

Project Blosc[‡†], Francesc Alted[‡†*], Marta Iborra[‡†], Oscar Guiñón[‡†], David Ibáñez[‡], Sergio Barrachina[§]

Abstract—Large multidimensional datasets are widely used in various engineering and scientific applications. Prompt access to the subsets of these datasets is crucial for an efficient exploration experience. To facilitate this, we have added support for large dimensional datasets to Blosc2, a compression and format library. The extension enables effective support for large multidimensional datasets, with a special encoding of zeros that allows for efficient handling of sparse datasets. Additionally, the new two-level data partition used in Blosc2 reduces the need for decompressing unnecessary data, further accelerating slicing speed.

The Blosc2 NDim layer enables the creation and reading of n-dimensional datasets in an extremely efficient manner. This is due to a completely general n-dim 2-level partitioning, which allows for slicing and dicing of arbitrary large (and compressed) data in a more fine-grained way. Having a second partition provides a better flexibility to fit the different partitions at the different CPU cache levels, making compression even more efficient.

Additionally, Blosc2 can make use of Btune, a library that automatically finds the optimal combination of compression parameters to suit user needs. Btune employs various techniques, such as a genetic algorithm and a neural network model, to discover the best parameters for a given dataset much more quickly. This approach is a significant improvement over the traditional trial-and-error method, which can take hours or even days to find the best parameters.

As an example, we will demonstrate how Blosc2 NDim enables fast exploration of the Milky Way using the Gaia DR3 dataset.

Index Terms—explore datasets, n-dimensional datasets, Gaia DR3, Milky Way, Blosc2, compression

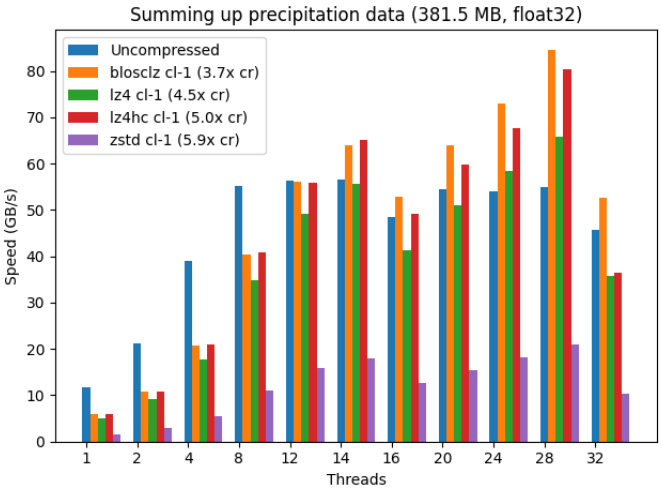

Fig. 1: *Speed for summing up a vector of real float32 data (meteorological precipitation) using a variety of codecs provided by Blosc2. Note that the maximum speed is achieved when utilizing the maximum number of (logical) threads available on the computer (28), where different codecs are allowing faster computation than using uncompressed data. Benchmark performed on a Intel i9-10940X CPU, with 14 physical cores. More info at [2].*

Introduction

The exploration of datasets that are high dimensional is a common practice in various fields of science. However, exploring such n-dimensional datasets is challenging when the memory size of the dataset is extremely large. This can slow down the data exploration process significantly. In this paper, we demonstrate how Blosc2 NDim can be used to accelerate the exploration of huge n-dimensional datasets.

Blosc is a high-performance compressor optimized for binary data. Its design enables faster transmission of data to the processor cache than the traditional, non-compressed, direct memory fetch approach using an OS call to `memcpy()`. This can be helpful not only in reducing the size of large datasets on-disk and in-memory, but also in accelerating memory-bound computations, which are typical in big data processing.

Blosc uses the blocking technique [1] to minimize activity on the memory bus. The technique divides datasets into blocks small enough to fit in the caches of modern processors, where compression/decompression is performed. Blosc also takes advantage of single-instruction multiple-data streams (SIMD), like SSE2, AVX2, NEON... and multi-threading capabilities in modern multi-core processors to maximize the compression/decompression speed.

In addition, using the Blosc compressed data can accelerate memory-bound computations when enough cores are dedicated to the task. Figure 1 provides a real example of this.

Blosc2 is the latest version of the Blosc 1.x series, which is used in many important libraries, such as HDF5 [3], Zarr [4], and PyTables [5]. Its NDim feature excels at reading multi-dimensional slices, thanks to an innovative pineapple-style partitioning technique [6]. This enables fast exploration of general n-dimensional datasets, including the 3D Gaia array.

† *These authors contributed equally.*
‡ *Project Blosc*
* *Corresponding author: francesc@blosc.org*
§ *Universitat Jaume I*

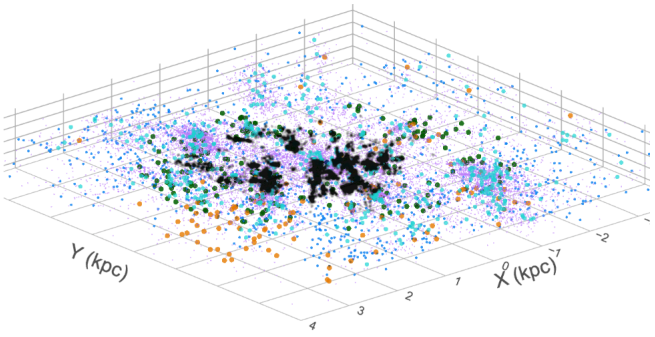

Fig. 2: *Gaia DR3 dataset as a 3D array (Gaia collaboration).*

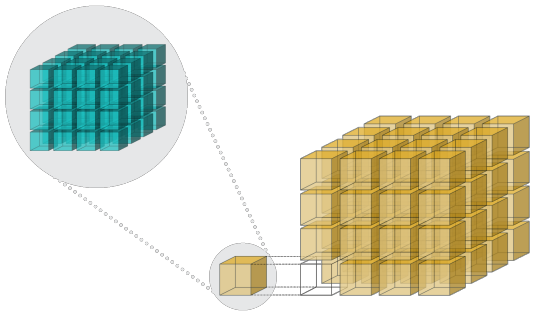

Fig. 3: *Blosc2 NDim 2-level partitioning.*

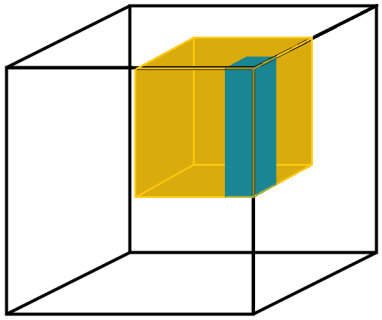

Fig. 4: *Blosc2 NDim 2-level partitioning is flexible. The dimensions of both partitions can be specified in any arbitrary way that fits the expected read access patterns.*

The Milky Way dataset

Figure 2 shows a 3D view of the Milky Way different type of stars. Each point is a star, and the color of each point represents the star's magnitude, with the brightest stars appearing as the reddest points. Although this view provides a unique perspective, the dimensions of the cube are not enough to fully capture the spiral arms of the Milky Way.

One advantage of using a 3D array is the ability to utilize Blosc2 NDim's powerful slicing capabilities for quickly exploring parts of the dataset. For example, we could search for star clusters by extracting small cubes as NumPy arrays, and counting the number of stars in each one. A cube containing an abnormally high number of stars would be a candidate for a cluster. We could also extract a thin 3D slice of the cube and project it as a 2D image, where the pixels colors represent the magnitude of the shown stars. This could be used to generate a cinematic view of a journey over different trajectories in the Milky Way.

For getting the coordinates of the stars in the Milky Way, we will be using the Gaia DR3 dataset [7], a catalog containing information on 1.7 billion stars in our galaxy. For this work, we extracted the 3D coordinates of 1.4 billion stars (those with non-null parallax values). When stored as a binary table, the dataset is 22 GB in size (uncompressed).

We converted the tabular dataset into a sphere with a radius of 10,000 light years and framed it into a 3D array of shape (20,000, 20,000, 20,000). Each cell in the array represents a cube of 1 light year per side and contains the number of stars within it. Given that the average distance between stars in the Milky Way is about 5 light years, very few cells will contain more than one star (e.g. the maximum of stars in a single cell in our sphere is 6). This 3D array contains 0.5 billion stars, which is a significant portion of the Gaia catalog.

The number of stars is stored as a uint8, resulting in a total dataset size of 7.3 TB. However, compression can greatly reduce its size to 2.2 GB since the 3D array is very sparse, and the Zstandard codec [8] is used. Blosc2 can compress the zeroed parts almost entirely thanks to a specific algorithm to detect zeros early in the compression pipeline and encoding them efficiently.

In addition, we store other data about the stars in a separate table indexed with the position of each star (using PyTables). For demonstration purposes, we store the distance from Sun, radial velocity, effective temperature, and G-band magnitude using a float32 for each field. The size of the table is 10 GB uncompressed, but it can be compressed to 4.8 GB. Adding another 1.0 GB for the index brings the total size to 5.8 GB. Therefore, the 3D array is 2.2 GB, and the table with the additional information and its index are 5.8 GB, making a total of 8.0 GB. This comfortably fits within the storage capacity of any modern laptop.

Blosc2 NDim

In the plain Blosc and Blosc2 libraries, there are two levels of partitioning: the block and the chunk. The block is the smallest unit of data that can be compressed and decompressed independently. The chunk is a group of blocks that are compressed together. The chunk and block sizes are parameters that can be tuned to fit the different cache levels in modern CPUs. For optimal performance, it is recommended that the block size should fit in the L1 or L2 CPU cache, minimizing contention between worker threads during compression/decompression. The chunk size, on the other hand, should fit in the L3 CPU cache, in order to minimize data movement to RAM and speed up decompression.

With Blosc2 NDim, we are taking this feature a step further and both partitions, known as chunks and blocks, are gaining multidimensional capabilities. This means that one can split a dataset (called a "super-chunk" in Blosc2 terminology) into n-dimensional cubes and sub-cubes. Refer to Figures 3 and 4 to learn more about how this works and how to set it up.

With these finer-grained cubes, arbitrary n-dimensional slices can be retrieved faster. This is because not all the data necessary for the coarser-grained partition has to be decompressed, as is typically required in other libraries (see Figure 5).

For example, for a 4-d array with a shape of (50, 100, 300, 250) with float64 items, we can choose a chunk with shape (10, 25, 50, 50) and a block with shape (3, 5, 10, 20) which makes for about 5 MB and 23 KB respectively. This way, a chunk fits

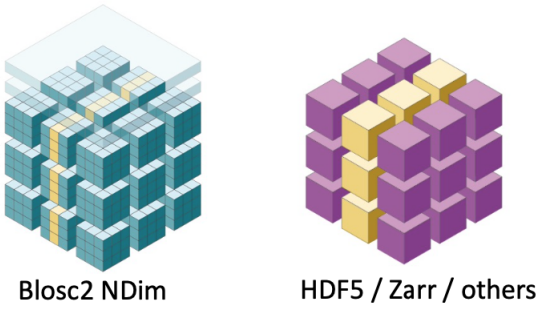

Fig. 5: *Blosc2 NDim can decompress data faster by using double partitioning, which allows for higher data selectivity. This means that less data compression/decompression is required in general.*

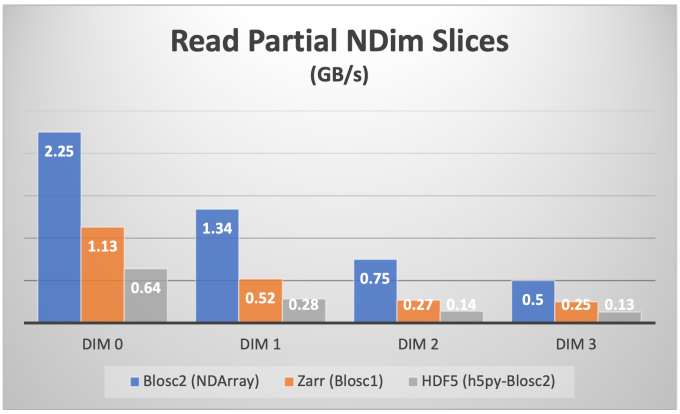

Fig. 6: *Speed comparison for reading partial n-dimensional slices of a 4D dataset. The legends labeled "DIM N" refer to slices taken orthogonally to each dimension. The sizes for the two partitions have been chosen such that the first partition fits comfortably in the L3 cache of the CPU (Intel i9 13900K), and the second partition fits in the L1 cache of the CPU. [6].*

comfortably on a L3 cache in most of modern CPUs, and a block in a L1 cache (we are tuning for speed here). See Figure 6 for a speed comparison with other libraries supporting just one single n-dimensional partition.

Finally, Blosc2 NDim supports all data types in NumPy. This means that, in addition to the typical data types like signed/unsigned int, single and double-precision floats, bools or strings, it can also store datetimes (including units), and arbitrarily nested heterogeneous types. This allows to create multidimensional tables and more.

Support for multiple codecs, filters, and other compression features

Blosc2 is not only a compression library, but also a framework for creating efficient compression pipelines. A compression pipeline is composed of a sequence of filters, followed by a compression codec. A filter is a transformation that is applied to the data before compression, and a codec is a compression algorithm that is applied to the filtered data. Filters can lead to better compression ratios and improved compression/decompression speeds.

Blosc2 supports a variety of codecs, filters, and other compression features. In particular, it supports the following codecs out-of-the-box:

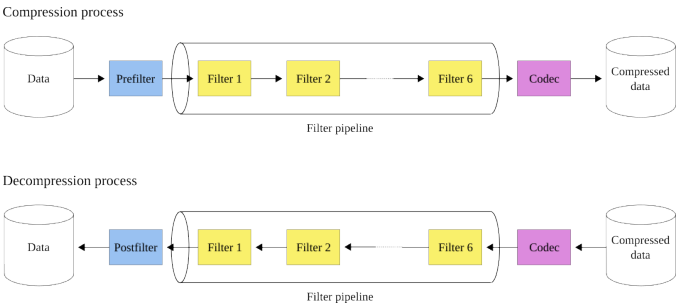

Fig. 7: *The Blosc2 filter pipeline. During compression, the first function applied is the prefilter (if any), followed by the filter pipeline (with a maximum of six filters), and finally, the codec. During decompression, the order is reversed: first the codec, then the filter pipeline, and finally the postfilter (if any).*

- BloscLZ (fast codec, the default),
- LZ4 (a very fast codec),
- LZ4HC (high compression variant of LZ4),
- Zlib (the Zlib-NG variant of Zlib),
- Zstd (high compression), and
- ZFP (lossy compression for n-dimensional datasets of floats).

It also supports the following filters out-of-the-box:

- Shuffle (groups equal significant bytes together, useful for ints/floats),
- Shuffle with bytedelta (same than shuffle, but storing deltas of consecutive same significant bytes),
- Bitshuffle (groups equal significant bits together, useful for ints/floats), and
- Truncation (truncates precision, useful for floats; lossy).

Blosc2 utilizes a pipeline architecture that enables the chaining of different filters [9] followed by a compression codec. Additionally, it allows for pre-filters (user code meant to be executed before the pipeline) and post-filters (user code meant to be executed after the pipeline). This architecture is highly flexible and minimizes data copies between the different steps, making it possible to create highly efficient pipelines for a variety of use cases. Figure 7 illustrates how this works.

Furthermore, Blosc2 supports user-defined codecs and filters, allowing one to create their own compression algorithms and use them within Blosc2 [9]. These user-defined codecs and filters can also be dynamically loaded [10], registered globally within Blosc2, and installed via a Python wheel so that they can be used seamlessly from any Blosc2 application (whether in C, Python, or any other language that provides a Blosc2 wrapper).

Automatic tuning of compression parameters

Finding the right compression parameters for the data is probably the most difficult part of using a compression library. Which combination of code and filters would provide the best compression ratio? Which one would provide the best compression/decompression speed?

Btune is an AI tool for Blosc2 that automatically finds the optimal combination of compression parameters to suit user needs. It uses a neural network trained on representative datasets to be compressed to predict the best compression parameters based on the given tradeoff between compression ratio and compression/decompression speed.

Tradeoff	Most predicted	Cratio	Cspeed	Dspeed
0.0	blosclz-nofilter-5	786.51	106.86	91.04
0.1	blosclz-nofilter-5	786.51	106.86	91.04
0.2	blosclz-nofilter-5	786.51	106.86	91.04
0.3	blosclz-nofilter-5	786.51	106.86	91.04
0.4	blosclz-nofilter-5	786.51	106.86	91.04
0.5	blosclz-nofilter-5	786.51	106.86	91.04
0.6	zstd-nofilter-9	8959.6	8.79	59.13
0.7	zstd-nofilter-9	8959.6	8.79	59.13
0.8	zstd-nofilter-9	8959.6	8.79	59.13
0.9	zstd-bitshuffle-9	10789.6	3.41	12.78
1.0	zstd-bitshuffle-9	10789.6	3.41	12.78

TABLE 1: *Btune prediction of the best compression parameters for decompression speed for the 3D Gaia array, depending on a tradeoff value between compression ratio and decompression speed. It can be seen that BloscLZ with compression level 5 is the most predicted category when decompression speed is preferred, whereas Zstd with compression level 9 + BitShuffle is the most predicted one when the specified tradeoff is towards optimizing for the compression ratio. Speeds are in GB/s.*

Tradeoff	Most predicted	Cratio	Cspeed	Dspeed
0.0	blosclz-shuffle-5	2.09	14.47	48.93
0.1	blosclz-shuffle-5	2.09	14.47	48.93
0.2	blosclz-shuffle-5	2.09	14.47	48.93
0.3	blosclz-shuffle-5	2.09	14.47	48.93
0.4	zstd-bytedelta-1	3.30	17.04	21.65
0.5	zstd-bytedelta-1	3.30	17.04	21.65
0.6	zstd-bytedelta-1	3.30	17.04	21.65
0.7	zstd-bytedelta-1	3.30	17.04	21.65
0.8	zstd-bytedelta-1	3.30	17.04	21.65
0.9	zstd-bytedelta-1	3.30	17.04	21.65
1.0	zstd-bytedelta-9	3.31	0.07	11.40

TABLE 2: *Btune prediction of the best compression parameters for decompression speed for another dataset (cancer imaging). It can be seen that BloscLZ with compression level 5 + Shuffle is the most predicted category when decompression speed is preferred, whereas Zstd (either compression level 1 or 9) + Shuffle + ByteDelta is the most predicted one when the specified tradeoff is towards optimizing for the compression ratio. Speeds are in GB/s.*

For example, Table 1 displays the results for the predicted compression parameters tuned for decompression speed of the 3D Gaia array. This table can be provided to the Btune plugin so that it can choose the best tradeoff value for user's needs (0 means favoring speed only, and 1 means favoring compression ratio only).

Of course, results will be different for another dataset. For example, Table 2 displays the results for the predicted compression parameters tuned for decompression speed for a dataset coming from cancer imaging. Curiously, in this case fast decompression does not necessarily imply fast compression.

On the other hand, there are also situations where data have to be compressed at a high speed (e.g. consolidating data from high bandwidth detectors). Table 3 shows an example of predicted compression parameter tuned this time for compression speed and ratio on yet another dataset for this scenario (in this case, images coming from synchrotron facilities).

Tradeoff	Most predicted	Cratio	Cspeed	Dspeed
0.0	lz4-bitshuffle-5	3.41	21.78	32.0
0.1	lz4-bitshuffle-5	3.41	21.78	32.0
0.2	lz4-bitshuffle-5	3.41	21.78	32.0
0.3	lz4-bitshuffle-5	3.41	21.78	32.0
0.4	lz4-bitshuffle-5	3.41	21.78	32.0
0.5	lz4-bitshuffle-5	3.41	21.78	32.0
0.6	lz4-bitshuffle-5	3.41	21.78	32.0
0.7	lz4-bitshuffle-5	3.41	21.78	32.0
0.8	zstd-bytedelta-1	3.98	9.41	18.8
0.9	zstd-bytedelta-1	3.98	9.41	18.8
1.0	zstd-bytedelta-9	4.06	0.15	14.1

TABLE 3: *Btune prediction of the best compression parameters for compression speed (synchrotron imaging). It can be seen that LZ4 with compression level 5 + Bitshuffle is the most predicted category when compression speed is preferred, whereas Zstd (either compression level 1 or 9) + Shuffle + ByteDelta is the most predicted one when the specified tradeoff is leveraged towards the compression ratio. Speeds are in GB/s.*

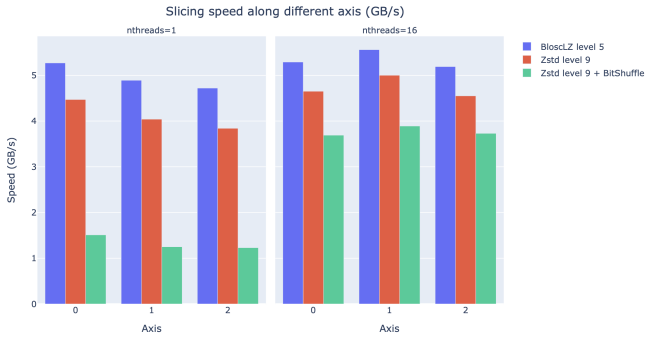

Fig. 8: *Speed of obtaining multiple multidimensional slices of the Gaia dataset along different axes, for different codecs, filters and different number of threads. The speed is measured in GB/s, so a higher value is better.*

After training the neural network, the Btune plugin can automatically tune the compression parameters for a given dataset. During inference, the user can set the preferred tradeoff by setting the `BTUNE_TRADEOFF` environment variable to a floating point value between 0 and 1. A value of 0 favors speed only, while a value of 1 favors compression ratio only. This setting automatically selects the compression parameters most suitable to the current data chunk whenever a new Blosc2 data container is being created.

Results on the Gaia dataset

We will use the training results above to compress the big 3D Gaia array so that it can be explored more quickly. Figure 8 displays the speed that can be achieved when getting multiple multidimensional slices of the dataset along different axes, using the most efficient codecs and filters for various tradeoffs.

These results indicate that the fastest compression is achieved with BloscLZ (compression level 5, no filters), closely followed by Zstd (compression level 9, no filters), exactly as the neural network model predicted. Also, note how the fastest decompression codecs, BloscLZ and also Zstd, are not affected very much by the number of threads used, which means that they are not CPU-bound, so

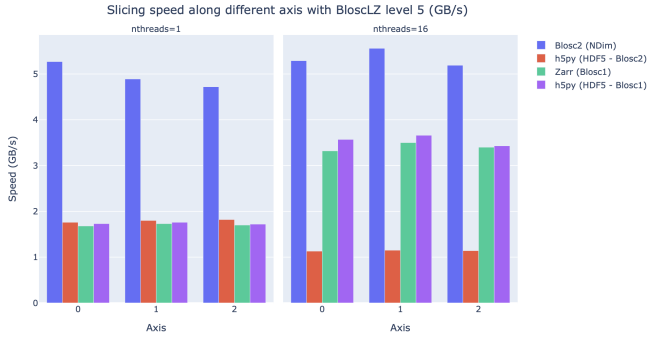

Fig. 9: *Slicing a section of the Gaia dataset with BloscLZ using different libraries. Note how using one single thread is still quite effective for Blosc2 NDim and BloscLZ.*

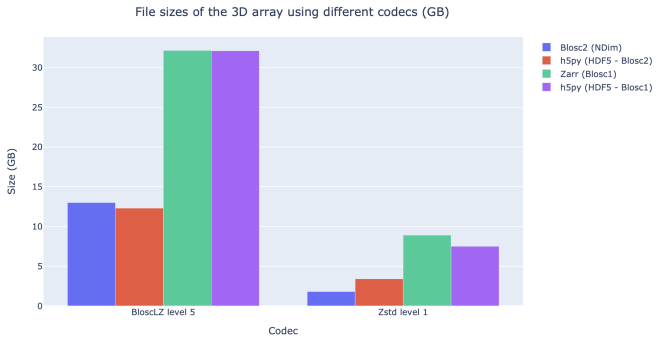

Fig. 10: *Compressing the Gaia dataset with BloscLZ and Zstd using different libraries. Blosc2 provides significantly better compression ratios than using Blosc1 . Also, note how Zstd compresses much better than BloscLZ.*

small computers or laptops with low core counts will be able to reach good speeds.

Now, let's compare the figures above with other libraries that can handle multidimensional data. Figure 9 shows the slicing speed of the 3D array when applying BloscLZ, the best predicted codec for speed, and we compare that speed against other libraries using the same codec but with the previous Blosc1 generation (Zarr and h5py), and also against Blosc2 via the hdf5plugin [11] and h5py. Results show that the data can be explored significantly faster using Blosc2 NDim with the BloscLZ codec. It is also interesting to note that the speed of Blosc2 NDim with BloscLZ is not much affected by the number of threads used, which is a welcome surprise, and probably an indication that the internal zero-suppression mechanism inside Blosc2 works efficiently without the need of multi-threading.

Regarding compression ratio, Figure 10 shows the results of compressing the Gaia dataset with Blosc2 NDim with BloscLZ and Zstd, and we compare that ratio against other libraries using the same codec but with the previous Blosc1 generation (Zarr and h5py), and also against Blosc2 via the hdf5plugin and h5py. Results show that the data can be compressed significantly better using Blosc2. This is because Blosc2 comes with a new and powerful zero-detection mechanism that is able to efficiently handle and compress the many zeros that are present in the Gaia dataset.

Ingesting and processing data of Gaia

The raw data of Gaia is stored in CSV files. The coordinates are stored in the gaia_source directory (http://cdn.gea.esac.esa.int/Gaia/gdr3/gaia_source/). These can be easily parsed and ingested as Blosc2 files with the following code:

```
def load_rawdata(out="gaia.b2nd"):
    dtype = {"ra": np.float32,
             "dec": np.float32,
             "parallax": np.float32}
    barr = None
    for file in glob.glob("gaia-source/*.csv*"):
        # Load raw data
        df = pd.read_csv(
            file,
            usecols=["ra", "dec", "parallax"],
            dtype=dtype, comment='#')
        # Convert to numpy array and remove NaNs
        arr = df.to_numpy()
        arr = arr[~np.isnan(arr[:, 2])]
        if barr is None:
            # Create a new Blosc2 file
            barr = blosc2.asarray(
                arr,
                chunks=(2**20, 3),
                urlpath=out,
                mode="w")
        else:
            # Append to existing Blosc2 file
            barr.resize(
                (barr.shape[0] + arr.shape[0], 3))
            barr[-arr.shape[0]:] = arr
    return barr
```

Once we have the raw data in a Blosc2 container, we can select the stars in a radius of 10 thousand light years using this function:

```
def convert_select_data(fin="gaia.b2nd",
                        fout="gaia-ly.b2nd"):
    barr = blosc2.open(fin)
    ra = barr[:, 0]
    dec = barr[:, 1]
    parallax = barr[:, 2]
    # 1 parsec = 3.26 light years
    ly = ne.evaluate("3260 / parallax")
    # Remove ly < 0 and > 10_000
    valid_ly = ne.evaluate(
        "(ly > 0) & (ly < 10_000)")
    ra = ra[valid_ly]
    dec = dec[valid_ly]
    ly = ly[valid_ly]
    # Cartesian x, y, z from spherical ra, dec, ly
    x = ne.evaluate("ly * cos(ra) * cos(dec)")
    y = ne.evaluate("ly * sin(ra) * cos(dec)")
    z = ne.evaluate("ly * sin(dec)")
    # Save to a new Blosc2 file
    out = blosc2.zeros(mode="w", shape=(3, len(x)),
                       dtype=x.dtype, urlpath=fout)
    out[0, :] = x
    out[1, :] = y
    out[2, :] = z
    return out
```

Finally, we can compute the density of stars in a 3D grid with this script:

```
R = 1  # resolution of the 3D cells in ly
LY_RADIUS = 10_000  # radius of the sphere in ly
CUBE_SIDE = (2 * LY_RADIUS) // R
MAX_STARS = 1000_000_000  # max number of stars to load

b = blosc2.open("gaia-ly.b2nd")
x = b[0, :MAX_STARS]
y = b[1, :MAX_STARS]
z = b[2, :MAX_STARS]

# Create 3d array.
# Be sure to have enough swap memory (around 8 TB!)
```

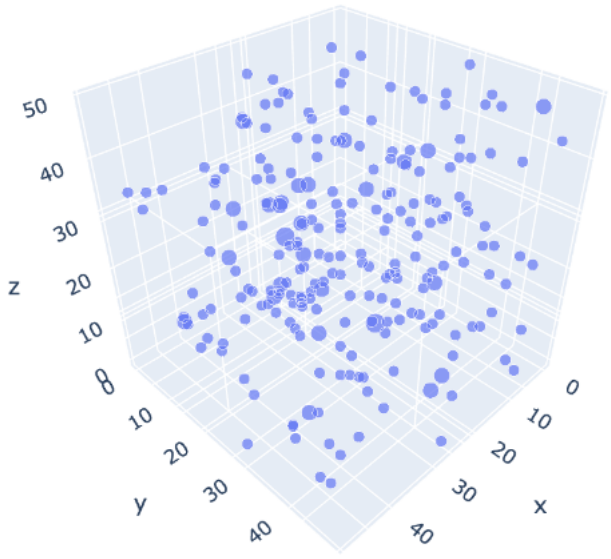

Fig. 11: *Stars in the vicinity of our Sun (cube of 50 light years). Each point represents a star, and its size represents the number of stars in that location (a cube of 1 x 1 x 1 light year). The maximum amount of stars in a single location for this view is 3 (triple star systems are common).*

```
a3d = np.zeros((CUBE_SIDE, CUBE_SIDE, CUBE_SIDE),
               dtype=np.float32)
for i, coords in enumerate(zip(x, y, z)):
    x_, y_, z_ = coords
    a3d[(np.floor(x_) + LY_RADIUS) // R,
        (np.floor(y_) + LY_RADIUS) // R,
        (np.floor(z_) + LY_RADIUS) // R] += 1

# Save 3d array as Blosc2 NDim file
blosc2.asarray(a3d,
               urlpath="gaia-3d.b2nd", mode="w",
               chunks=(250, 250, 250),
               blocks=None,
               )
```

With that, we have a 3D array of shape 20,000 x 20,000 x 20,000 with the number of stars with a 1 light year resolution. We can visualize the vicinity of our Sun with Plotly [12] making use of the following code:

```
import blosc2
import numpy as np
import plotly.express as px

nstars_path = '$HOME/Gaia/gaia-3d-windows-int8.b2nd'
b3d = blosc2.open(nstars_path)
data = b3d[9_975:10_025, 9_975:10_025, 9_975:10_025]
idx = np.indices(data.shape)
fig = px.scatter_3d(x=idx[0, :, :, :].flatten(),
                    y=idx[1, :, :, :].flatten(),
                    z=idx[2, :, :, :].flatten(),
                    size=data[...].flatten())
fig.show()
```

Figure 11 displays an interactive 3D view of the stars within a 50 x 50 x 50 light-year cube centered around our Sun. This visualization was generated using the code above.

In [13] you can find the final version of the scripts above, including optimized versions that do not require a machine with more than 32 GB of virtual memory to run.

Conclusions

Working with large, multi-dimensional data cubes can be challenging due to the costly data handling involved. In this document, we demonstrate how the two-partition feature in Blosc2 NDim can help reduce the amount of data movement required when retrieving thin slices of large datasets. Additionally, this feature provides a foundation for leveraging cache hierarchies in modern CPUs.

Blosc2 supports a variety of compression codecs and filters, making it easier to select the most appropriate ones for the dataset being explored. It also supports storage in either memory or on disk, which is crucial for large datasets. Another important feature is the ability to store data in a container format that can be easily shared across different programming languages. Furthermore, Blosc2 has special support for sparse datasets, which greatly improves the compression ratio in this scenario.

We have also shown how the Btune plugin can be used to automatically tune the compression parameters for a given dataset. This is especially useful when we want to compress data efficiently for a tradeoff between compression or decompression speed and compression ratio, but we do not know the best compression parameters beforehand.

In conclusion, we have shown how to utilize the Blosc2 library for storing and processing the Gaia dataset. This dataset serves as a prime example of a large, multi-dimensional dataset that can be efficiently stored and processed using Blosc2 NDim.

Acknowledgments

Jordi Portell, member of the Gaia Collaboration, has been very helpful in answering many questions about the Gaia dataset, and has also proposed possible explorations of it.

NumFOCUS, a non-profit organization with a mission to promote open practices in research, data, and scientific computing. They have provided steady funds to the Blosc Development Team over the past years.

Huawei, a high-tech company that made a significant and selfless donation to the Blosc project.

Sergio Barrachina, associate professor at University Jaume I, has provided many advice and code during the development of the Btune project.

This work has made use of data from the European Space Agency (ESA) mission Gaia (https://www.cosmos.esa.int/gaia), processed by the Gaia Data Processing and Analysis Consortium DPAC (https://www.cosmos.esa.int/web/gaia/dpac/consortium). Funding for the DPAC has been provided by national institutions, in particular the institutions participating in the Gaia Multilateral Agreement.

References

[1] Francesc Alted, "Why Modern CPUs Are Starving and What Can Be Done About It," *Computing in Science and Engineering*, vol. 12, pp. 68–71, 2010, https://doi.org/10.1109/MCSE.2010.51.

[2] ——. (2018) Breaking Down Memory Walls. Https://www.blosc.org/posts/breaking-memory-walls/.

[3] The HDF Group. (1997-2023) Hierarchical Data Format, version 5. Https://www.hdfgroup.org/HDF5/.

[4] Zarr Developers. (2017-2023) An implementation of chunked, compressed, N-dimensional arrays for Python. https://doi.org/10.5281/zenodo.7971911.

[5] PyTables developers. (2002-2023) A Python package to manage extremely large amounts of data. Http://www.pytables.org.

[6] Francesc Alted and Oscar Guiñón. (2023) Introducing Blosc2 NDim. Https://www.blosc.org/posts/blosc2-ndim-intro/.
[7] European Space Agency (ESA) and Gaia Data Processing and Analysis Consortium (DPAC). (2023) Gaia Data Release 3. Documentation release 1.2. Https://gea.esac.esa.int/archive/documentation/GDR3/.
[8] Yann Collet et al. (2023) Zstandard - Fast real-time compression algorithm. Https://github.com/facebook/zstd.
[9] Marta Iborra. (2022) User Defined Pipeline for Python-Blosc2. Https://www.blosc.org/posts/python-blosc2-pipeline/.
[10] Marta Iborra and Francesc Alted. (2023) Dynamic Plugins in C-Blosc2. Https://www.blosc.org/posts/dynamic-plugins/.
[11] Silx maintainers. (2023) Set of compression filters for h5py. Https://github.com/silx-kit/hdf5plugin.
[12] Plotly Technologies Inc. (2015) Collaborative data science. Montreal, QC. Https://plot.ly.
[13] The Blosc Development Team. (2023) Scripts for "A Fast Explorer Of The Milky Way" talk. Https://github.com/Blosc/exploring-milky-way.git.

Python Array API Standard: Toward Array Interoperability in the Scientific Python Ecosystem

Aaron Meurer[‡†*], Athan Reines[‡†], Ralf Gommers[‡†], Yao-Lung L. Fang[§†], John Kirkham[§†], Matthew Barber[‡†], Stephan Hoyer[¶], Andreas Müller[‖], Sheng Zha[**], Saul Shanabrook, Stephannie Jiménez Gacha[‡], Mario Lezcano-Casado[‡], Thomas J. Fan[‡], Tyler Reddy[††], Alexandre Passos, Hyukjin Kwon[‡‡], Travis Oliphant[‡], Consortium for Python Data API Standards

Abstract—The Python array API standard specifies standardized application programming interfaces (APIs) and behaviors for array and tensor objects and operations as commonly found in libraries such as NumPy [1], CuPy [2], PyTorch [3], JAX [4], TensorFlow [5], Dask [6], and MXNet [7]. The establishment and subsequent adoption of the standard aims to reduce ecosystem fragmentation and facilitate array library interoperability in user code and among array-consuming libraries, such as scikit-learn [8] and SciPy [9]. A key benefit of array interoperability for downstream consumers of the standard is device agnosticism, whereby previously CPU-bound implementations can more readily leverage hardware acceleration via graphics processing units (GPUs), tensor processing units (TPUs), and other accelerator devices.

In this paper, we first introduce the Consortium for Python Data API Standards and define the scope of the array API standard. We then discuss the current status of standardization and associated tooling (including a test suite and compatibility layer). We conclude by outlining plans for future work.

Index Terms—Python, Arrays, Tensors, NumPy, CuPy, PyTorch, JAX, Tensorflow, Dask, MXNet

Introduction

Python users have a wealth of choices for libraries and frameworks for numerical computing [10][1][9][2][6][11][12][13], data science [14][15][16][17], machine learning [8], and deep learning [7][3][5][18]. New frameworks pushing forward the state of the art appear every year. One consequence of all this activity has been fragmentation in the fundamental building blocks—multidimensional arrays [19] (also known as tensors)—that underpin the scientific Python ecosystem (hereafter referred to as "the ecosystem").

This fragmentation comes with significant costs, from reinvention and re-implementation of arrays and associated application programming interfaces (APIs) to siloed technical stacks targeting only one array library to the proliferation of user guides providing guidance on how to convert between libraries. The APIs of each library are largely similar, but each have enough differences that end users have to relearn and rewrite code in order to work with multiple libraries. This process can be very painful as the transition is far from seamless and creates barriers for libraries wanting to support multiple array library backends.

The Consortium for Python Data API Standards (hereafter referred to as "the Consortium" and "we") aims to address this problem by standardizing a fundamental array data structure and an associated set of common APIs for working with arrays, thus facilitating interchange and interoperability.

Paper Overview

This paper is written as an introduction to the Consortium and the array API standard. The aim is to provide a high-level overview of the standard and its continued evolution and to solicit further engagement from the Python community.

After providing an overview of the Consortium, we first discuss standardization methodology. We then discuss the current status of the array API standard and highlight the main standardization areas. Next, we introduce tooling associated with the standard for testing compliance and shimming incompatible array library behavior. We conclude by outlining open questions and opportunities for further standardization. Links to the specification and all current repository artifacts, including associated tooling, can be found in the bibliography.

Consortium Overview

History

While the Python programming language was not explicitly designed for numerical computing, the language gained popularity in scientific and engineering communities soon after its release. The first array computing library for numerical and scientific computing in Python was Numeric, developed in the mid-1990s [20][1]. To better accommodate this library and its use cases, Python's syntax was extended to include indexing syntax [21].

In the early 2000s, Numarray introduced a more flexible data structure [22]. Numarray had faster operations for large arrays, but

† *These authors contributed equally.*
∗ *Corresponding author: asmeurer@quansight.com*
‡ *Quansight*
§ *NVIDIA Corporation*
¶ *Google*
‖ *Microsoft*
∗∗ *Amazon*
†† *LANL*
‡‡ *Databricks*

slower operations for small arrays. Subsequently, both Numeric and Numarray coexisted to satisfy different use cases.

In early 2005, the NumPy library unified Numeric and Numarray as a single array package by porting Numarray's features to Numeric [1]. This effort was largely successful and resolved the fragmentation at the time. For roughly a decade, NumPy was the only widely used array library. Building on NumPy, pandas was subsequently introduced in 2008 in order to address the need for a high performance, flexible tool for performing quantitative analysis on labeled tabular data [23].

Over the past 10 years, the rise of deep learning and the emergence of new hardware has led to a proliferation of new libraries and a corresponding fragmentation within the PyData array and dataframe ecosystem. These libraries often borrowed concepts from, or entirely copied, the APIs of older libraries, such as NumPy, and then modified and evolved those APIs to address new needs and use cases. Although the communities of each library individually discussed interchange and interoperability, no general coordination arose among libraries to avoid further fragmentation and to arrive at a common set of API standards until the founding of the Consortium.

The genesis for the Consortium grew out of many conversations among maintainers during 2019–2020. During those conversations, it quickly became clear that any attempt to create a new reference library to address fragmentation was infeasible. Unlike in 2005, too many different use cases and varying stakeholders now exist. Furthermore, the speed of innovation of both hardware and software is simply too great.

In May 2020, an initial group of maintainers and industry stakeholders[1] assembled to form the Consortium for Python Data API Standards and began drafting a specification for array APIs, which could then be adopted by existing array libraries and their dependents and by any new libraries which arise.

Objectives

Standardization efforts must maintain a balance between codifying what already exists and maintaining relevance with respect to future innovation. The latter aspect is particularly fraught, as relevance requires anticipating future needs, technological advances, and emerging use cases. Accordingly, if a standard is to remain relevant, the standardization process must be conservative in its scope, thorough in its consideration of current and prior art, and have clearly defined objectives against which success is measured.

To this end, we established four objectives for the array API standard. 1) Allow array-consuming libraries to accept and operate on arrays from multiple different array libraries. 2) Establish a common set of standardized APIs and behaviors, enabling more sharing and code reuse. 3) For new array libraries, offer a concrete API that can be adopted as-is. 4) Minimize the learning curve and friction for users as they switch between array libraries.

We explicitly omitted three notable possible objectives. 1) Make array libraries identical for the purpose of merging them. Different array libraries have different strengths (e.g., performance characteristics, hardware support, and tailored use cases, such as deep learning), and merging them into a single array library is neither practical nor realistic. 2) Implement a backend or runtime switching system in order to switch from one array library to another via a single setting or line of code. While potentially feasible, array consumers are likely to need to modify code in order to ensure optimal performance and behavior. 3) Support mixing multiple array libraries in a single function call. Mixing array libraries requires defining hierarchies and specifying rules for device synchronization and data localization. Such rules are likely to be specific to individual use cases.

Design Principles

In order to define the contours of the standardization process, we established the following design principles:

Functions. The standardized API should consist primarily of standalone functions. Function-based API design is the dominant pattern among array libraries, both in Python and in other frequently used programming languages supporting array computation, such as MATLAB [24] and Julia [25]. While method chaining and the fluent interface design pattern are also relatively common, especially among array libraries supporting deferred execution and operator fusion, function-based APIs are generally preferred. This mirrors design patterns used in underlying implementations, such as those written in C/C++ and Fortran, and more closely matches written mathematical notation.

Minimal array object. The standard should not require that an array object have any attributes or methods beyond what is necessary for inspection (e.g., shape, data type, and device) or for supporting operator overloading (i.e., magic methods).[2]

No dependencies. The standard and its implementations should not require any dependencies outside of Python itself.

Accelerator support. Standardized APIs and behaviors should be possible to implement for both central processing units (CPUs) and hardware-accelerated devices, such as graphics processing units (GPUs), tensor processing units (TPUs), and field-programmable gate arrays (FPGAs).

Compiler support. Standardized APIs and behaviors should be amenable to just-in-time (JIT) and ahead-of-time (AOT) compilation and graph-based optimization techniques, such as those used by PyTorch [3], JAX [4], and TensorFlow [5]. APIs and behaviors not amenable to compilation, such as APIs returning arrays having data-dependent output shapes or polymorphic return types, should either be omitted or specified as optional.[3] In general, the shape, data type, and device of the return value from any function should be predictable from its input arguments.

Distributed support. Standardized APIs and behaviors should be amenable to implementation in array libraries supporting distributed computing (e.g., Dask [6]).

Consistency. Except in scenarios involving backward compatibility concerns, naming conventions and design patterns should be consistent across standardized APIs.

Extensibility. Conforming array libraries may implement functionality which is not included in the array API standard. Array consumers thus bear responsibility for ensuring that their API usage is portable across specification-conforming array libraries.

Deference. Where possible, the array API standard should defer to existing, widely-used standards. For example, the accu-

1. Direct stakeholders include the maintainers of Python array and dataframe libraries and organizations which sponsor library development. Indirect stakeholders include maintainers of libraries which consume array and dataframe objects ("consuming libraries"), developers of compilers and runtimes with array- and dataframe-specific functionality, and end users, such as data scientists and application developers.

2. Notably, array strides should be considered an implementation detail and should not be required as a public Python attribute.

3. Copy-view mutation semantics, such as those currently supported by NumPy, should be considered an implementation detail and, thus, not suitable for standardization.

racy and precision of numerical functions should not be specified beyond the guidance included in IEEE 754 [26].

Universality. Standardized APIs and behaviors should reflect common usage among a wide range of existing array libraries. Accordingly, with rare exception, only APIs and behaviors having existing implementations and broad support within the ecosystem may be considered candidates for standardization.

Methodology

A foundational step in technical standardization is articulating a subset of established practices and defining those practices in unambiguous terms. To this end, the standardization process must approach the problem from two directions: design and usage.

The former direction seeks to understand both current implementation design (APIs, names, signatures, classes, and objects) and semantics (calling conventions and behavior). The latter direction seeks to quantify API consumers (who are the downstream users of a given API?), usage frequency (how often is an API consumed?), and consumption patterns (which optional arguments are provided and in what context?). By analyzing both design and usage, we sought to ground the standardization process and specification decisions in empirical data and analysis.

Design

To understand API design of array libraries within the ecosystem, we first identified a representative sample of commonly used array libraries. This sample included NumPy, CuPy, PyTorch, JAX, TensorFlow, Dask, and MXNet. Next, we extracted public APIs for each library by analyzing module exports and scraping public web documentation. The following APIs for computing the arithmetic mean provide an example of extracted API data:

```
numpy.mean(a, axis=None, dtype=None, out=None,
    keepdims=<no value>)
cupy.mean(a, axis=None, dtype=None, out=None,
    keepdims=False)
torch.mean(input, dim, keepdim=False, out=None)
jax.numpy.mean(a, axis=None, dtype=None, out=None,
    keepdims=False)
tf.math.reduce_mean(input_tensor, axis=None,
    keepdims=False, name=None)
dask.array.mean(a, axis=None, dtype=None, out=None,
    keepdims=False, split_every=None)
mxnet.np.mean(a, axis=None, dtype=None, out=None,
    keepdims=False)
```

We determined commonalities and differences by analyzing the intersection, and its complement, of available APIs across each array library. From the intersection, we derived a subset of common APIs suitable for standardization based on prevalence and ease of implementation. The common API subset included function names, method names, attribute names, and positional and keyword arguments. As an example of a derived API, consider the common API for computing the arithmetic mean:

```
mean(a, axis=None, keepdims=False)
```

To assist in determining standardization prioritization, we leveraged usage data (discussed below) to confirm API need and to inform naming conventions, supported data types, and optional arguments. We have summarized findings and published tooling [27] for additional analysis and exploration, including Jupyter notebooks [17], as public artifacts available on GitHub.

Usage

To understand usage patterns of array libraries within the ecosystem, we first identified a representative sample of commonly used Python libraries ("downstream libraries") which consume the aforementioned array libraries. The sample of downstream libraries included SciPy [9], pandas [23], Matplotlib [14], xarray [12], scikit-learn [8], statsmodels [16], and scikit-image [11], among others. Next, we ran downstream library test suites with runtime instrumentation enabled. We recorded input arguments and return values for each API invocation by inspecting the bytecode stack at call time [28]. From the recorded data, we generated inferred signatures for each function based on provided arguments and associated types, noting which downstream library called which empirical API and at what frequency. We organized the API results in human-readable form as type definition files and compared the inferred API to the publicly documented APIs obtained during design analysis.

The following is an example of two inferred API signatures for `numpy.mean`, with the docstring indicating the number of lines of code which invoked the function for each downstream library when running library test suites. Based on the example, we can infer that invoking the function with an array input argument is a more common usage pattern among downstream libraries than invoking the function with a list of floats.

```
@overload
def mean(a: numpy.ndarray):
    """
    usage.dask: 21
    usage.matplotlib: 7
    usage.scipy: 26
    usage.skimage: 36
    usage.sklearn: 130
    usage.statsmodels: 45
    usage.xarray: 1
    """

@overload
def mean(a: List[float]):
    """
    usage.networkx: 6
    usage.sklearn: 3
    usage.statsmodels: 9
    """
```

As a final step, we ranked each API according to relative usage using the Dowdall positional voting system [29] (a variant of the Borda count [30] that favors candidate APIs having high relative usage). From the rankings, we assigned standardization priorities, with higher ranking APIs taking precedence over lower ranking APIs, and extended the analysis to aggregated API categories (e.g., array creation, manipulation, statistics, etc.). All source code, usage data, and analysis are publicly available on GitHub [28][27].

Array API Standard

The Python array API standard specifies standardized APIs and behaviors for array and tensor objects and operations. The scope of the standard includes defining, but is not limited to, the following: 1) a minimal array object; 2) semantics governing array interaction, including type promotion and broadcasting; 3) an interchange protocol for converting array objects originating from different array libraries; 4) a set of required array-aware functions; and 5) optional extensions for specialized APIs and array behaviors. We discuss each of these standardization areas in turn.

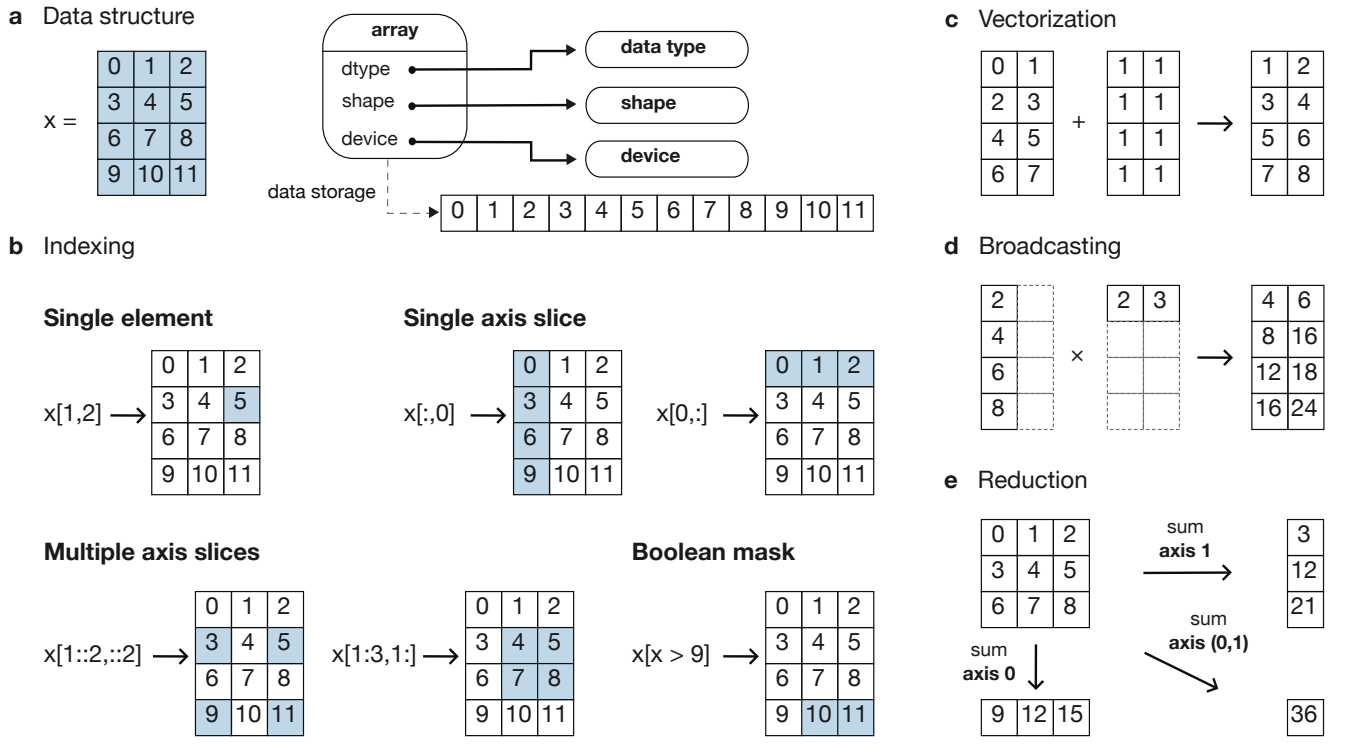

Fig. 1: *The array data structure and fundamental concepts.* ***a)*** *An array data structure and its associated metadata fields.* ***b)*** *Indexing an array. Indexing operations may access individual elements or sub-arrays. Applying a boolean mask is an optional indexing behavior and may not be supported by all conforming libraries.* ***c)*** *Vectorization obviates the need for explicit looping in user code by applying operations to multiple array elements.* ***d)*** *Broadcasting enables efficient computation by implicitly expanding the dimensions of array operands to equal sizes.* ***e)*** *Reduction operations act along one or more axes. In the example, summation along a single axis produces a one-dimensional array, while summation along two axes produces a zero-dimensional array containing the sum of all array elements.*

Array Object

An array object is a data structure for efficiently storing and accessing multidimensional arrays [19]. Within the context of the array API standard, the data structure is opaque—libraries may or may not grant direct access to raw memory—and includes metadata for interpreting the underlying data, notably "data type", "shape", and "device" (Fig. 1a).

An array data type ("dtype") describes how to interpret a single array element (e.g., integer, real- or complex-valued floating-point, boolean, or other). A conforming array object has a single dtype. To facilitate interoperability, conforming libraries must support and provide a minimal set of dtype objects (e.g., `int8`, `int16`, `int32`, `float32`, and `float64`). To ensure portability, data type objects must be provided by name in the array library namespace (e.g., `xp.bool`).

An array shape specifies the number of elements along each array axis (also referred to as "dimension"). The number of axes corresponds to the dimensionality (or "rank") of an array. For example, the shape `(3, 5)` corresponds to a two-dimensional array whose inner dimension contains five elements and whose outer dimension contains three elements. The shape `()` corresponds to a zero-dimensional array containing a single element.

An array device specifies the location of array memory allocation. A conforming array object is assigned to a single logical device. To support array libraries supporting execution on different device types (e.g., CPUs, GPUs, TPUs, etc.), conforming libraries must provide standardized device APIs in order to coordinate execution location. The following example demonstrates how an array-consuming library might use standardized device APIs to ensure execution occurs on the same device as the input.

```
def some_function(x):
    # Retrieve a standard-compliant namespace
    xp = x.__array_namespace__()

    # Allocate a new array on the same device as x
    y = xp.linspace(0, 2*xp.pi, 100, device=x.device)

    # Perform computation (on device)
    return xp.sin(y) * x
```

To interact with array objects, one uses "indexing" to access sub-arrays and individual elements, "operators" to perform logical and arithmetic operations (e.g., `+`, `-`, `*`, `/`, and `@`), and array-aware functions (e.g., for linear algebra, statistical reductions, and element-wise computation). Array indexing semantics extend built-in Python sequence `__getitem__()` indexing semantics to support element access across multiple dimensions (Fig. 1b).[4] Indexing an array using a boolean array (also known as "masking") is an optional standardized behavior.[5] The result of a mask operation is data-dependent and thus difficult to implement in array libraries relying on static analysis for graph-based optimization.

4. The array API standard includes support for in-place operations via `__setitem__()`; however, behavior is undefined if an in-place operation would affect arrays other than the target array (e.g., in array libraries supporting multiple "views" of the same underlying memory).

5. While not currently supported, integer array indexing may be included in a future revision of the array API standard.

Array Interaction

The Python array API standard further specifies rules governing expected behavior when an operation involves two or more array operands. For operations in which the data type of a resulting array object is resolved from operand data types, the resolved data type must follow type promotion semantics. Importantly, type promotion semantics are independent of array shape or contained values (including when an operand is a zero-dimensional array). For example, when adding one array having a `float32` data type to another array having a `float64` data type, the data type of the resulting array should be the promoted data type `float64`.

```
>>> x1 = xp.ones((2, 2), dtype=xp.float32)
>>> x2 = xp.ones((2, 2), dtype=xp.float64)
>>> y = x1 + x2
>>> y.dtype == xp.float64
True
```

In addition to type promotion, the array API standard specifies rules governing the automatic (and implicit) expansion of array dimensions to be of equal sizes (Fig. 1d). Standardized broadcasting semantics are the same as those popularized by NumPy [1].

Interchange Protocol

We expect that array library consumers will generally prefer to use a single array "type" (e.g., a NumPy `ndarray`, PyTorch `Tensor`, or Dask `array`) and will thus need a standardized mechanism for array object conversion. For example, suppose a data visualization library prefers to use NumPy internally but would like to extend API support to any conforming array object type. In such a scenario, the library would benefit from a reliable mechanism for accessing and reinterpreting the memory of externally provided array objects without triggering potential performance cliffs due to unnecessary copying of array data. To this end, the Python array API standard specifies an interchange protocol describing the memory layout of a strided, n-dimensional array in an implementation-independent manner.

The basis of this protocol is DLPack, an open in-memory structure for sharing tensors among frameworks [31]. DLPack is a standalone protocol with an ABI stable, header-only C implementation with cross hardware support. The array API standard builds on DLPack by specifying Python APIs for array object data interchange [32]. Conforming array objects must support `__dlpack__` and `__dlpack_device__` magic methods for accessing array data and querying the array device. A standardized `from_dlpack()` API calls these methods to construct a new array object of the desired type using zero-copy semantics when possible. The combination of DLPack and standardized Python APIs thus provides a stable, widely adopted, and efficient means for array object interchange.

Array Functions

To complement the minimal array object, the Python array API standard specifies a set of required array-aware functions for arithmetic, statistical, algebraic, and general computation. Where applicable, required functions must support vectorization (Fig. 1d), which obviates the need for explicit looping in user code by applying operations to multiple array elements. Vectorized abstractions confer two primary benefits: 1) implementation-dependent optimizations leading to increased performance and 2) concise expression of mathematical operations. For example, one can express element-wise computation of z-scores in a single line.

```
def z_score(x):
    return (x - xp.mean(x)) / xp.stdev(x)
```

In addition to vectorized operations, the array API standard includes, but is not limited to, functions for creating new arrays, with support for explicit device allocation, reshaping and manipulating existing arrays, performing statistical reductions across one, multiple, or all array axes (Fig. 1e), and sorting array elements. Altogether, these APIs provide a robust and portable foundation for higher-order array operations and general array computation.

Optional Extensions

While a set of commonly used array-aware functions is sufficient for many array computation use cases, additional, more specialized, functionality may be warranted. For example, while most data visualization libraries are unlikely to explicitly rely on APIs for computing Fourier transforms, signal analysis libraries supporting spectral analysis of time series are likely to require Fourier transform APIs. To accommodate specialized APIs, the Python array API standard includes standardized optional extensions.

An extension is a sub-namespace of a main namespace and is defined as a coherent set of standardized functionality which is commonly implemented across many, but not all, array libraries. Due to implementation difficulty (or impracticality), limited general applicability, a desire to avoid significantly expanding API surface area beyond what is essential, or some combination of the above, requiring conforming array libraries to implement and maintain extended functionality beyond their target domain is not desirable. Extensions provide a means for conforming array libraries to opt-in to supporting standardized API subsets according to need and target audience.

Specification Status

Following formation of the Consortium in 2020, we released an initial draft of the Python array API standard for community review in 2021. We have released two subsequent revisions:

v2021.12: The first full release of the specification, detailing purpose and scope, standardization methodology, future standard evolution, a minimal array object, an interchange protocol, required data types, type promotion and broadcasting semantics, an optional linear algebra extension, and array-aware functions for array creation, manipulation, statistical reduction, and vectorization, among others.

v2022.12: This revision includes errata for the v2021.12 release and adds support for single- and double-precision complex floating-type data types, additional array-aware APIs, an optional extension for computing fast Fourier transforms.

For future revisions, we expect annual release cadences; however, array API standard consumers should not assume a fixed release schedule.

Implementation Status

Reference Implementation

To supplement the Python array API standard, we developed a standalone reference implementation. The implementation is strictly compliant (i.e., any non-portable usage triggers an exception) and is available as the `numpy.array_api` submodule (discussed in [33]). In general, we do not expect for users to rely on the reference implementation for production use cases. Instead, the reference implementation is primarily useful for array-consuming libraries as a means for testing whether array library usage is guaranteed to be portable.

Ecosystem Adoption

Arrays are fundamental to scientific computing, data science, and machine learning. As a consequence, the Python array API standard has many stakeholders within the ecosystem. When establishing the Consortium, we thus sought participation from a diverse and representative cross-section of industry partners and maintainers of array and array-consuming libraries. To satisfy stakeholder needs, array library maintainers worked in close partnership with maintainers of array-consuming libraries throughout the array API standardization process to identify key use cases and achieve consensus on standardized APIs and behaviors.

Direct participation in the Consortium by array and array-consuming library maintainers has facilitated coordination across the ecosystem. In addition to the numpy.array_api reference implementation [34], several commonly used array libraries, including NumPy [35], CuPy [36], PyTorch [37], JAX [38], Dask [39], and MXNet [40], have either adopted or are in the process of adopting the array API standard. Increased array library adoption has increased array interoperability, which, in turn, has encouraged array-consuming libraries, such as SciPy [41] and scikit-learn [42] (discussed below), to begin adopting the standard by decoupling their implementations from specific array libraries. As array library adoption of the standard matures, we expect ecosystem adoption to accelerate.

Tooling

Test Suite

To facilitate adoption of the Python array API standard by libraries within the ecosystem, we developed a test suite to measure specification compliance [43]. The test suite covers all major aspects of the specification, such as broadcasting, type promotion, function signatures, special case handling, and expected return values.

Underpinning the test suite is Hypothesis, a Python library for creating unit tests [44]. Hypothesis uses property-based testing, a technique for generating arbitrary data satisfying provided specifications and asserting the truth of some "property" that should be true. Property-based testing is particularly convenient when authoring compliance tests, as the technique enables the direct translation of specification guidance into test code.

The test suite is the first example known to these authors of a Python test suite capable of running against multiple different libraries. As part of our work, we upstreamed strategies to Hypothesis for generating arbitrary arrays from any conforming array library, thus allowing downstream array consumers to test against multiple array libraries and their associated hardware devices.

Compatibility Layer

While we expect that maintainers of conforming array libraries will co-evolve library APIs and behaviors with those specified in the Python array API standard, we recognize that co-evolution is not likely to always proceed in unison due to varying release cycles and competing priorities. Varying timelines for adoption and full-compliance present obstacles for array-consuming libraries hoping to use the most recent standardized behavior, as such libraries are effectively blocked by the slowest array library release schedule.

To address this problem and facilitate adoption of the standard by array-consuming libraries, we developed a compatibility layer (named `array-api-compat`), which provides a thin wrapper around common array libraries [45]. The layer transparently intercepts API calls for any API which is not fully-compliant and polyfills non-compliant specification-defined behavior. For compliant APIs, it exposes the APIs directly, without interception, thus mitigating performance degradation risks due to redirection. To reduce barriers to adoption, the layer supports vendoring and has a small, pure Python codebase with no hard dependencies.

While the Python array API standard facilitates array interoperability in theory, the compatibility layer does so in practice, helping array-consuming libraries decouple adoption of the standard from the release cycles of upstream array libraries. Currently, the layer provides shims for NumPy, CuPy, and PyTorch and aims to support additional array libraries in the future. By ensuring specification-compliant behavior, we expect the compatibility layer to have a significant impact in accelerating adoption among array-consuming libraries.

Discussion

The principle aim of the Python array API standard is to facilitate interoperability of array libraries within the ecosystem. In achieving this aim, array-consuming libraries, such as those for statistical computing, data science, and machine learning, can decouple their implementations from specific array libraries. Decoupling subsequently allows end users to use the array library that is most applicable to their use case and to no longer be limited by the set of array libraries a particular array-consuming library supports.

In addition to improved developer ergonomics afforded by standardized APIs and increased interoperability, standardization allows end users and the authors of array-consuming libraries to use a declarative, rather than imperative, programming paradigm. This paradigm change has a key benefit in enabling users to opt into performance improvements based on their constraints and hardware capabilities. To assess the impact of this change, we worked with maintainers of scikit-learn and SciPy to measure the performance implications of specification adoption (Fig. 2).

scikit-learn

scikit-learn is a widely-used machine learning library. Its current implementation relies heavily on NumPy and SciPy and is a mixture of Python and Cython. Due to its dependence on NumPy for array computation, scikit-learn is CPU-bound, and the library is unable to capture the benefits of GPU- and TPU-based execution models. By adopting the Python array API standard, scikit-learn can decouple its implementation from NumPy and support non-CPU-based execution, potentially enabling increased performance.

To test this hypothesis, we examined the scikit-learn codebase to identify APIs which rely primarily on NumPy for their implementation. scikit-learn estimators are one such set of APIs, having methods for model fitting, classification prediction, and data projection, which are amenable to input arrays supporting alternative execution models. Having identified potential API candidates, we selected the estimator class for linear discriminant analysis (LDA) as a representative test case. Refactoring the LDA implementation was illustrative in several respects, as demonstrated in the following code snippet showing source code modifications[6]:

```
1   Xc = []
2   for idx, group in enumerate(self.classes_):
3 -     Xg = X[y == group, :]
4 -     Xc.append(Xg - self.means_[idx])
```

6. Source code modifications reflect those required for NumPy version 1.24.3 and Python array API standard version 2022.12.

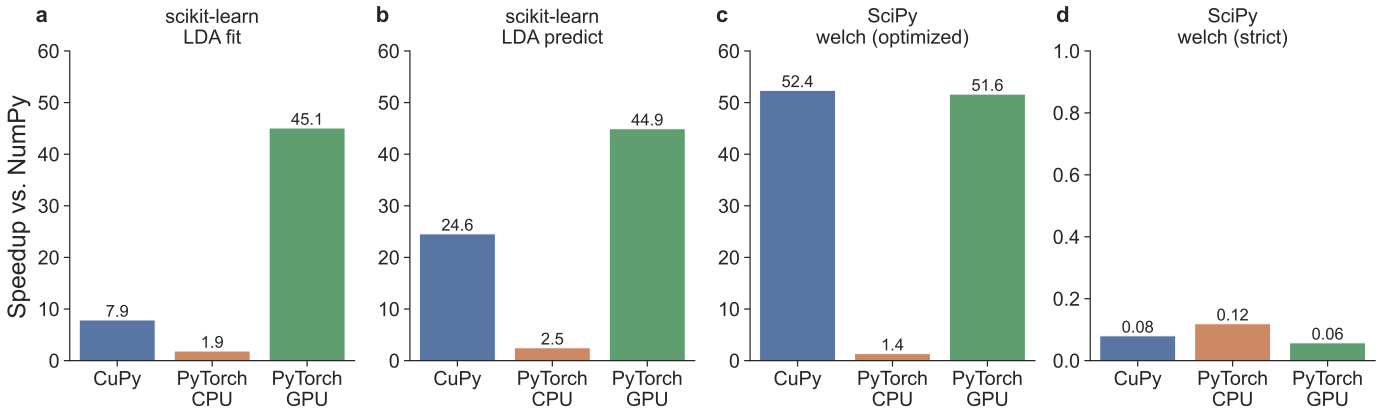

Fig. 2: *Benchmarks measuring performance implications of adoption in array-consuming libraries. Displayed timings are relative to NumPy. All benchmarks were run on Intel i9-9900K and NVIDIA RTX 2080 hardware.* ***a)*** *Fitting a linear discriminant analysis (LDA) model.* ***b)*** *Predicting class labels using LDA.* ***c)*** *Estimating power spectral density using Welch's method and library-specific optimizations.* ***d)*** *Same as* ***c****, but using a strictly portable implementation. Note that* ***d*** *has different vertical axis limits than* ***a-c****.*

```
+      Xg = X[y == group]
+      Xc.append(Xg - self.means_[idx, :])

- self.xbar_ = np.dot(self.priors_, self.means_)
+ self.xbar_ = self.priors_ @ self.means_

- Xc = np.concatenate(Xc, axis=0)
+ Xc = xp.concat(Xc, axis=0)

- std = Xc.std(axis=0)
+ std = xp.std(Xc, axis=0)

  std[std == 0] = 1.0
- fac = 1.0 / (n_samples - n_classes)
+ fac = xp.asarray(1.0 / (n_samples - n_classes))

- X = np.sqrt(fac) * (Xc / std)
+ X = xp.sqrt(fac) * (Xc / std)

  U, S, Vt = svd(X, full_matrices=False)

- rank = np.sum(S > self.tol)
+ rank = xp.sum(xp.astype(S > self.tol, xp.int32))
```

Indexing: *(lines 3-6)* NumPy supports non-standardized indexing semantics. To be compliant with the standard, 1) boolean masks must be the sole index and cannot be combined with other indexing expressions, and 2) the number of provided single-axis indexing expressions must equal the number of dimensions.

Non-standardized APIs: *(lines 8-9)* NumPy supports several APIs having no equivalent in the array API standard; `np.dot()` is one such API. For two-dimensional arrays, `np.dot()` is equivalent to matrix multiplication and was updated accordingly.

Naming conventions: *(lines 11-12)* NumPy contains several standard-compliant APIs whose naming conventions differ from those in the array API standard. In this and similar cases, adoption requires conforming to the standardized conventions.

Functions: *(lines 14-15)* NumPy supports several array object methods having no equivalent in the array API standard. To ensure portability, we refactored use of non-standardized methods in terms of standardized function-based APIs.

Scalars: *(lines 18-22)* NumPy often supports non-array input arguments, such as scalars, Python lists, and other objects, as "array-like" arguments in its array-aware APIs. While the array API standard does not prohibit such polymorphism, the standard does not require array-like support. In this case, we explicitly convert a scalar expression to a zero-dimensional array in order to ensure portability when calling `xp.sqrt()`.

Data types: *(lines 26-27)* NumPy often supports implicit type conversion of non-numeric data types in numerical APIs. The array API standard does not require such support, and, more generally, mixed-kind type promotion semantics (e.g., boolean to integer, integer to floating-point, etc.) are not specified. To ensure portability, we must explicitly convert a boolean array to an integer array before calling `xp.sum()`.

To test the performance implications of refactoring scikit-learn's LDA implementation, we generated a random two-class classification problem having 400,000 samples and 300 features.[7] We next devised two benchmarks, one for fitting an LDA model and the second for predicting class labels for each simulated sample. We ran the benchmarks and measured execution time for NumPy, PyTorch, and CuPy backends on Intel i9-9900K and NVIDIA RTX 2080 hardware. For PyTorch, we collected timings for both CPU and GPU execution models. To ensure timing reproducibility and reduce timing noise, we repeated each benchmark ten times and computed the average execution time.

Fig. 2a and Fig. 2b display results, showing average execution time relative to NumPy. When fitting an LDA model (Fig. 2a), we observe 1.9× higher throughput for PyTorch CPU, 7.9× for CuPy, and 45.1× for PyTorch GPU. When predicting class labels (Fig. 2b), we observe 2.5× higher throughput for PyTorch CPU, 24.6× for CuPy, and 44.9× for PyTorch GPU. In both benchmarks, using GPU execution models corresponded to significantly increased performance, thus supporting our hypothesis that scikit-learn can benefit from non-CPU-based execution models.

SciPy

SciPy is a collection of mathematical algorithms and convenience functions for numerical integration, optimization, interpolation, statistics, linear algebra, signal processing, and image processing, among others. Similar to scikit-learn, its current implementation relies heavily on NumPy. We thus sought to test whether SciPy could benefit from adopting the Python array API standard.

7. To ensure that observed performance is not an artifact of the generated dataset, we tested performance across multiple random datasets and did not observe a measurable difference across benchmark runs.

Following a similar approach to the scikit-learn benchmarks, we identified SciPy's signal processing APIs as being amenable to input arrays supporting alternative execution models and selected an API for estimating the power spectral density using Welch's method [46] as a representative test case. We then generated a representative synthetic test signal (a 2 Vrms sine wave at 1234 Hz, corrupted by 0.001 V^2/Hz of white noise sampled at 10 kHz) having 50,000,000 data points. We next devised two benchmarks, one using library-specific optimizations and a second strictly using APIs in the array API standard. We ran the benchmarks for the same backends, on the same hardware, and using the same analysis approach as the scikit-learn benchmarks discussed above.

Fig. 2c and Fig. 2d display results, showing average execution time relative to NumPy. When using library-specific optimizations (Fig. 2c), we observe 1.4× higher throughput for PyTorch CPU, 51.6× for PyTorch GPU, and 52.4× for CuPy. When omitting library-specific optimizations (Fig. 2d), we observe a 12-25× **decreased** throughput across all non-NumPy backends.

The source of the performance disparity is due to use of strided views in the optimized implementation. NumPy, CuPy, and PyTorch support the concept of strides, where a stride describes the number of bytes to move forward in memory to progress to the next position along an axis, and provide similar, non-standardized APIs for manipulating the internal data structure of an array. While one can use standardized APIs to achieve the same result, using stride "tricks" enables increased performance. This finding raises an important point. Namely, while the array API standard aims to reduce the need for library-specific code, it will never fully eliminate that need. Users of the standard may need to maintain similar library-specific performance optimizations to achieve maximal performance. We expect, however, that the maintenance burden should only apply for those scenarios in which the performance benefits significantly outweigh the maintenance costs.

Future Work

Consortium work is comprised of three focus areas: standardization, adoption, and coordination.

Standardization: Standardization is the core of Consortium efforts. The Python array API standard is a living standard, which should evolve to reflect the needs and evolution of array libraries within the ecosystem. As such, we expect to continue working with array and array-consuming library maintainers to codify APIs and behaviors suitable for standardization.

Adoption: To ensure the success of the Python array API standard, we work closely with maintainers of array and array-consuming libraries to facilitate adoption by soliciting feedback, addressing pain points, and resolving specification ambiguities. In the immediate future, we plan to release additional tooling for tracking adoption and measuring specification compliance. For the former, we are collecting static compliance data and will publish compatibility tables as part of the array API standard publicly available on-line. For the latter, we are developing an automated test suite reporting system to gather array API test suite results from array libraries as part of continuous integration. We expect these tools to be particularly valuable to array-consuming libraries in order to quickly assess API portability.

Coordination: Providing a forum for coordination among array libraries (and their consumers) was the primary motivating factor behind Consortium formation and is the most important byproduct of Consortium efforts. By facilitating knowledge exchange among array library communities, the Consortium serves as a critical bulwark against further fragmentation and siloed technical stacks. Preventing such fragmentation is to the ultimate benefit of array library consumers and their communities. Additionally, coordination allows for orienting around a shared long-term vision regarding future needs and possible solutions. We are particularly keen to explore the following areas and open questions: device standardization, extended data type support (including strings and datetimes), input-output (IO) APIs, support for mixing array libraries, parallelization, and optional extensions for deep learning, statistical computing, and, more generally, functionality which is out-of-scope, but needed in specific contexts.[8]

We should also note that array API standardization is not the only standardization effort spearheaded by the Consortium. We are also working to standardize APIs and behaviors for Python dataframe libraries, including an interchange protocol and a library-author focused dataframe object and associated set of APIs. This work will be discussed in a future paper.

Conclusion

We introduced the Consortium and the Python array API standard, which specifies standardized APIs and behaviors for array and tensor objects and operations. In developing an initial specification draft, we analyzed common array libraries in the ecosystem and determined a set of common APIs suitable for standardization. In consultation with array and array-consuming library maintainers, we published two specification revisions codifying APIs and behaviors for array objects and their interaction, array interchange, and array-aware functions for array creation and manipulation, statistical reduction, and linear algebra. In addition, we released tooling to facilitate adoption of the array API standard: 1) a test suite for measuring specification compliance and 2) a compatibility layer to allow array-consuming libraries to adopt the standard without having to wait on upstream release cycles.

We further explored performance implications of adopting the array API standard in two commonly-used array-consuming libraries: scikit-learn and SciPy. For the former, we found that adoption enabled scikit-learn to use GPU-based execution models, resulting in significantly increased performance. For the latter, we found similar performance gains; however, in order to realize the performance gains, we needed to use library-specific optimizations. This finding highlights a limitation of the standard. Namely, while the array API standard aims to reduce the need for library-specific code, it will never fully eliminate that need. Users of the standard may need to maintain similar library-specific performance optimizations to achieve maximal performance.

Our work demonstrates the usefulness of the Consortium and the array API standard in facilitating array interoperability within the ecosystem. In addition to shepherding standardization and promoting adoption of the array API standard, the Consortium provides a critical forum for coordinating efforts among array and array-consuming library maintainers. Such coordination is critical to the long-term success and viability of the ecosystem and its communities. Having established a blueprint for standardization methodology and process, the Consortium is also leading a similar effort to standardize Python dataframe APIs and behaviors, thus working to reduce fragmentation for the two fundamental data structures underpinning the ecosystem—arrays and dataframes.

8. To participate in Consortium efforts, consult the Python array API standard public issue tracker [47].

REFERENCES

[1] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant, "Array programming with NumPy," *Nature*, vol. 585, no. 7825, pp. 357–362, Sep. 2020, https://doi.org/10.1038/s41586-020-2649-2. [Online]. Available: https://www.nature.com/articles/s41586-020-2649-2

[2] R. Okuta, Y. Unno, D. Nishino, S. Hido, and C. Loomis, "CuPy: A NumPy-Compatible Library for NVIDIA GPU Calculations," in *Proceedings of Workshop on Machine Learning Systems (LearningSys) in The Thirty-first Annual Conference on Neural Information Processing Systems (NIPS)*, 2017. [Online]. Available: https://www.semanticscholar.org/paper/CuPy-%3A-A-NumPy-Compatible-Library-for-NVIDIA-GPU-Okuta-Unno/a59da4639436f582e483347a4833e7659fd3e598

[3] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "PyTorch: an imperative style, high-performance deep learning library," in *Proceedings of the 33rd International Conference on Neural Information Processing Systems*. Red Hook, NY, USA: Curran Associates Inc., Dec. 2019, no. 721, pp. 8026–8037.

[4] J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne, and Q. Zhang, "JAX: composable transformations of Python+NumPy programs," http://github.com/google/jax, 2018. [Online]. Available: http://github.com/google/jax

[5] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: a system for large-scale machine learning," in *Proceedings of the 12th USENIX conference on Operating Systems Design and Implementation*, ser. OSDI'16. USA: USENIX Association, Nov. 2016, pp. 265–283.

[6] M. Rocklin, "Dask: Parallel Computation with Blocked algorithms and Task Scheduling," in *Proceedings for the Annual Scientific Computing with Python Conference*, Austin, Texas, 2015, pp. 126–132, https://doi.org/10.25080/Majora-7b98e3ed-013. [Online]. Available: https://conference.scipy.org/proceedings/scipy2015/matthew_rocklin.html

[7] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, "MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems," Dec. 2015, https://doi.org/10.48550/arXiv.1512.01274. [Online]. Available: http://arxiv.org/abs/1512.01274

[8] 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 (JMLR)*, vol. 12, pp. 2825–2830, 2011. [Online]. Available: www.jmlr.org/papers/v12/pedregosa11a.html

[9] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. J. van der Walt, M. Brett, J. Wilson, K. J. Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. J. Carey, I. Polat, Y. Feng, E. W. Moore, J. VanderPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, and P. van Mulbregt, "SciPy 1.0: fundamental algorithms for scientific computing in Python," *Nature Methods*, vol. 17, no. 3, pp. 261–272, Mar. 2020, https://doi.org/10.1038/s41592-019-0686-2. [Online]. Available: https://www.nature.com/articles/s41592-019-0686-2

[10] K. J. Millman and M. Aivazis, "Python for Scientists and Engineers," *Computing in Science & Engineering*, vol. 13, no. 2, pp. 9–12, Mar. 2011, https://doi.org/10.1109/MCSE.2011.36.

[11] S. J. van der Walt, J. L. Schönberger, J. Nunez-Iglesias, F. c. Boulogne, J. D. Warner, N. Yager, E. Gouillart, and T. Yu, "scikit-image: image processing in Python," *PeerJ*, vol. 2, p. e453, Jun. 2014, https://doi.org/10.7717/peerj.453. [Online]. Available: https://peerj.com/articles/453

[12] S. Hoyer and J. Hamman, "xarray: N-D labeled Arrays and Datasets in Python," vol. 5, no. 1, p. 10, Apr. 2017, https://doi.org/10.5334/jors.148. [Online]. Available: https://openresearchsoftware.metajnl.com/articles/10.5334/jors.148

[13] H. Abbasi, "Sparse: A more modern sparse array library," in *Proceedings of the 17th Python in Science Conference*, Austin, Texas, 2018, pp. 65–68, https://doi.org/10.25080/Majora-4af1f417-00a. [Online]. Available: https://conference.scipy.org/proceedings/scipy2018/hameer_abbasi.html

[14] J. D. Hunter, "Matplotlib: A 2D Graphics Environment," *Computing in Science & Engineering*, vol. 9, no. 3, pp. 90–95, May 2007, https://doi.org/10.1109/MCSE.2007.55.

[15] F. Pérez, B. E. Granger, and J. D. Hunter, "Python: An Ecosystem for Scientific Computing," *Computing in Science & Engineering*, vol. 13, no. 2, pp. 13–21, Mar. 2011, https://doi.org/10.1109/MCSE.2010.119.

[16] S. Seabold and J. Perktold, "Statsmodels: Econometric and Statistical Modeling with Python," in *Proceedings for the Annual Scientific Computing with Python Conference*, Austin, Texas, 2010, pp. 92–96, https://doi.org/10.25080/Majora-92bf1922-011. [Online]. Available: https://conference.scipy.org/proceedings/scipy2010/seabold.html

[17] T. Kluyver, B. Ragan-Kelley, F. Pérez, B. E. Granger, M. Bussonnier, J. Frederic, K. Kelley, J. B. Hamrick, J. Grout, S. Corlay, P. Ivanov, D. Avila, S. Abdalla, C. Willing, and J. D. Team, "Jupyter notebooks - a publishing format for reproducible computational workflows," in *International Conference on Electronic Publishing*, 2016, https://doi.org/10.3233/978-1-61499-649-1-87.

[18] R. Frostig, M. Johnson, and C. Leary, "Compiling machine learning programs via high-level tracing," in *Proceedings of SysML Conference*, 2018. [Online]. Available: https://mlsys.org/Conferences/doc/2018/146.pdf

[19] S. van der Walt, S. C. Colbert, and G. Varoquaux, "The NumPy Array: A Structure for Efficient Numerical Computation," *Computing in Science & Engineering*, vol. 13, no. 2, pp. 22–30, Mar. 2011, https://doi.org/10.1109/MCSE.2011.37.

[20] P. F. Dubois, K. Hinsen, and J. Hugunin, "Numerical Python," *Computer in Physics*, vol. 10, no. 3, pp. 262–267, May 1996, https://doi.org/10.1063/1.4822400. [Online]. Available: https://doi.org/10.1063/1.4822400

[21] J. Hugunin, "Extending Python for Numerical Computation," http://hugunin.net/papers/hugunin95numpy.html, 1995. [Online]. Available: http://hugunin.net/papers/hugunin95numpy.html

[22] P. Greenfield, J. T. Miller, J.-c. Hsu, and R. L. White, "numarray: A New Scientific Array Package for Python," in *PyCon DC*, 2003. [Online]. Available: https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.112.9899

[23] W. McKinney, "pandas: a Foundational Python Library for Data Analysis and Statistics," 2011. [Online]. Available: https://www.semanticscholar.org/paper/pandas:-a-Foundational-Python-Library-for-Data-and-McKinney/1a62eb61b2663f8135347171e30cb9dc0a8931b5

[24] C. Moler and J. Little, "A history of MATLAB," *Proceedings of the ACM on Programming Languages*, vol. 4, no. HOPL, pp. 81:1–81:67, Jun. 2020, https://doi.org/10.1145/3386331. [Online]. Available: https://dl.acm.org/doi/10.1145/3386331

[25] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah, "Julia: A Fresh Approach to Numerical Computing," *SIAM Review*, vol. 59, no. 1, pp. 65–98, Jan. 2017, https://doi.org/10.1137/141000671. [Online]. Available: https://epubs.siam.org/doi/10.1137/141000671

[26] "IEEE Standard for Floating-Point Arithmetic," *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pp. 1–84, Jul. 2019, https://doi.org/10.1109/IEEESTD.2019.8766229.

[27] C. for Python Data API Standards, "Array API Comparison," https://github.com/data-apis/array-api-comparison, 2022. [Online]. Available: https://github.com/data-apis/array-api-comparison

[28] ——, "Python Record API," https://github.com/data-apis/python-record-api, 2020. [Online]. Available: https://github.com/data-apis/python-record-api

[29] J. Fraenkel and B. Grofman, "The Borda Count and its real-world alternatives: Comparing scoring rules in Nauru and Slovenia," *Australian Journal of Political Science*, vol. 49, no. 2, pp. 186–205, Apr. 2014, https://doi.org/10.1080/10361146.2014.900530. [Online]. Available: https://doi.org/10.1080/10361146.2014.900530

[30] P. Emerson, "The original Borda count and partial voting," *Social Choice and Welfare*, vol. 40, no. 2, pp. 353–358, Feb. 2013, https://doi.org/10.1007/s00355-011-0603-9. [Online]. Available: https://doi.org/10.1007/s00355-011-0603-9

[31] DLPack, "Open In Memory Tensor Structure," https://github.com/dmlc/dlpack, 2023. [Online]. Available: https://github.com/dmlc/dlpack

[32] ——, "Python Specification for DLPack," https://dmlc.github.io/dlpack/latest/python_spec.html, 2023. [Online]. Available: https://dmlc.github.io/dlpack/latest/python_spec.html

[33] R. Gommers, S. Hoyer, and A. Meurer, "NEP 47 — Adopting the array API standard — NumPy Enhancement Proposals," https://numpy.org/

neps/nep-0047-array-api-standard.html, Jan. 2021. [Online]. Available: https://numpy.org/neps/nep-0047-array-api-standard.html
[34] A. Meurer, "Implementation of the NEP 47 (adopting the array API standard) by asmeurer · Pull Request #18585 · numpy/numpy," https://github.com/numpy/numpy/pull/18585, Mar. 2021. [Online]. Available: https://github.com/numpy/numpy/pull/18585
[35] S. Berg, "Road to NumPy 2.0," https://mail.python.org/archives/list/numpy-discussion@python.org/thread/XYA5KZNL362Q5KWLKS5QFBQNRT5N2ZJO/#XCJU55EXSQPN5W7UWHDKURBU7EKBBTD2, Jan. 2023. [Online]. Available: https://mail.python.org/archives/list/numpy-discussion@python.org/thread/XYA5KZNL362Q5KWLKS5QFBQNRT5N2ZJO/#XCJU55EXSQPN5W7UWHDKURBU7EKBBTD2
[36] Y.-L. L. Fang, "Adopt Python Array API standard · Issue #4789 · cupy/cupy," https://github.com/cupy/cupy/issues/4789, Mar. 2021. [Online]. Available: https://github.com/cupy/cupy/issues/4789
[37] P. Meier, "Python Array API Compatibility Tracker · Issue #58743 · pytorch/pytorch," https://github.com/pytorch/pytorch/issues/58743, May 2021. [Online]. Available: https://github.com/pytorch/pytorch/issues/58743
[38] J. VanderPlas, "Initial implementation of the Python Array API standard · Pull Request #16099 · google/jax," https://github.com/google/jax/pull/16099, May 2023. [Online]. Available: https://github.com/google/jax/pull/16099
[39] T. White, "Python Array API in Dask issue tracking · Issue #8917 · dask/dask," https://github.com/dask/dask/issues/8917, Apr. 2022. [Online]. Available: https://github.com/dask/dask/issues/8917
[40] N. Yyc, "Python Array API standardization · Issue #20501 · apache/mxnet," https://github.com/apache/mxnet/issues/20501, Aug. 2021. [Online]. Available: https://github.com/apache/mxnet/issues/20501
[41] I. Yashchuk, "Using Array API standard for functions implemented using pure Python and NumPy API · Issue #15354 · scipy/scipy," https://github.com/scipy/scipy/issues/15354, Jan. 2022. [Online]. Available: https://github.com/scipy/scipy/issues/15354
[42] T. Fan, "Path for Adopting the Array API spec · Issue #22352 · scikit-learn/scikit-learn," https://github.com/scikit-learn/scikit-learn/issues/22352, Jan. 2022. [Online]. Available: https://github.com/scikit-learn/scikit-learn/issues/22352
[43] C. for Python Data API Standards, "Test Suite for Array API Compliance," https://github.com/data-apis/array-api-tests, 2022. [Online]. Available: https://github.com/data-apis/array-api-tests
[44] D. MacIver, Z. Hatfield-Dodds, and M. Contributors, "Hypothesis: A new approach to property-based testing," *Journal of Open Source Software*, vol. 4, no. 43, p. 1891, 11 2019, https://doi.org/10.21105/joss.01891. [Online]. Available: http://dx.doi.org/10.21105/joss.01891
[45] C. for Python Data API Standards, "Array API compatibility library," https://github.com/data-apis/array-api-compat, 2023. [Online]. Available: https://github.com/data-apis/array-api-compat
[46] P. Welch, "The use of fast Fourier transform for the estimation of power spectra: A method based on time averaging over short, modified periodograms," *IEEE Transactions on Audio and Electroacoustics*, vol. 15, no. 2, pp. 70–73, Jun. 1967, https://doi.org/10.1109/TAU.1967.1161901.
[47] C. for Python Data API Standards, "Array API standard," https://github.com/data-apis/array-api, 2022. [Online]. Available: https://github.com/data-apis/array-api

A Modified Strassen Algorithm to Accelerate Numpy Large Matrix Multiplication with Integer Entries

Anthony Breitzman[‡*]

Abstract—Numpy is a popular Python library widely used in the math and scientific community because of its speed and convenience. We present a Strassen type algorithm for multiplying large matrices with integer entries. The algorithm is the standard Strassen divide and conquer algorithm but it crosses over to Numpy when either the row or column dimension of one of the matrices drops below 128. The algorithm was tested on a MacBook, an I7 based Windows machine as well as a Linux machine running a Xeon processor and we found that for matrices with thousands of rows or columns and integer entries, the Strassen based algorithm with crossover performed 8 to 30 times faster than regular Numpy on such matrices. Although there is no apparent advantage for matrices with real entries, there are a number of applications for matrices with integer coefficients.

Index Terms—Strassen, Numpy, Integer Matrix

Introduction

A recent article [1] suggests that Python is rapidly becoming the *Lingua Franca* of machine learning and scientific computing because of powerful frameworks such as Numpy, SciPy, and TensorFlow. These libraries offer great flexibility while boosting the performance of Python because they are written in compiled C and C++.

In this short paper we present a modified Strassen-based [2] algorithm for multiplying large matrices of arbitrary sizes containing integer entries. The algorithm uses Strassen's algorithm for several divide and conquer steps before crossing over to a regular Numpy matrix multiplication. For large matrices the method is 8 to 30 times faster than calling Numpy.matmul or Numpy.dot to multiply the matrices directly. The method was tested on a variety of hardware and the speed advantage was consistent for cases with integer entries. There is no such advantage for matrices with floating point entries however as [3] points out, there are numerous applications for large matrices with integer entries, including high precision evaluation of so-called holonomic functions (e.g. exp, log, sin, Bessel functions, and hypergeometric functions) as well as areas of Algebraic Geometry to name just two. Integer matrices are also frequently used as adjacency matrices in graph theory applications and are also used extensively in combinatorics.

To give the reader some early perspective, we will see later in the paper that some of the matrix multiplies that we do with the suggested algorithm take approximately two minutes using the new algorithm, but take 44 minutes using Numpy.matmul.

It is suggested in [4] that Numpy may be the single most-imported non-stdlib module in the entire Pythonverse. Therefore, an algorithm that speeds Numpy for large integer matrices may be of interest to a large audience.

* *Corresponding author: breitzman@rowan.edu*
‡ *Rowan University Department of Computer Science*

Motivating Exploration with Baseline Timings

For motivation consider the well-known standard algorithm for multipyling a pair of *NxN* matrices, as found in [5] as well as any algorithms book.

```
#multiply matrix A and B and put
#the product into C.
#A,B,C are assumed to be square
#matrices of the same dimension.
#No error checking is done.
def multiply(A, B, C):
  for i in range(N):
    for j in range( N):
      C[i][j] = 0
      for k in range(N):
         C[i][j]+=A[i][k]*B[k][j]
```

It is clear from the three nested loops that this algorithm has $O(N^3)$ running time.

Strassen's algorithm [2] is described most easily in Figure 1 which is modified from GeeksForGeeks [5]. We see that to multiply two $N \times N$ matrices via Strassen's method requires seven multiplications plus eighteen additions or subtractions of matrices that are size $(N/2) \times (N/2)$. The additions and subtractions will cost $O(N^2)$ and therefore the time complexity of Strassen's algorithm is $T(N) = 7T(N/2) + O(N^2)$ which by the Master Theorem [6] is $O(N^{\log_2 7} \simeq O(N^{2.81})$. Python code for an initial implementation of the standard Strassen algorithm can be found in [5].

To get a baseline for our improved algorithms below we show how the standard multiplication and the Geeks-for-Geeks implementation of the Strassen algorithm perform compared to Numpy.matmul on several large square matrices with integer coefficients. Timings are provided in Table 1. Unsurprisingly, the Numpy implementation of matrix multiply is orders of magnitude faster than the other methods. This is expected because Numpy is written in compiled C and as discussed above is known for its speed and efficiency. The table contains a column where we compute the current timing divided by the previous timing. As noted above the complexity of Strassen's algorithm is $O(N^{\log_2 7})$ thus when we double the size of N we expect the timing to increase about 7-fold. The current/previous column shows that

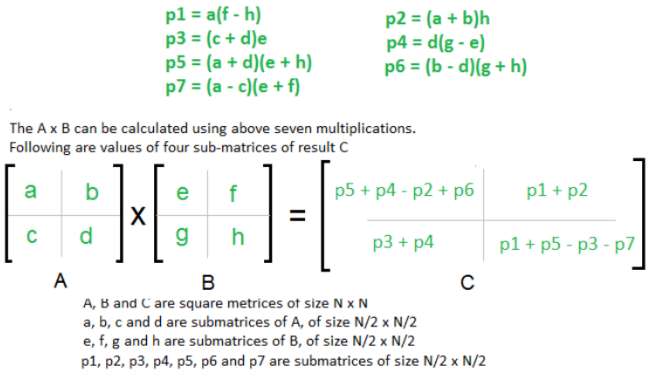

Fig. 1: *Illustration of Strassen's Algorithm for multiplying 2 Square Matrices (Modified from GeeksForGeeks)*

this is the case. Similarly we expect the standard algorithm's timing to increase about 8-fold when we double the N and this seems to be the case as well. Still, the Strassen algorithm as implemented here is not a practical algorithm in spite of the lower complexity. Although it would start to be faster than the standard matrix multiplication for $N = 4096$ and larger, it would not rival the Numpy multiplication until N reached 10^{16}

Implementing Strassen with a Crossover to Numpy

It is clear from the initial timings in Table 1 that to improve the Strassen implementation we should crossover to Numpy at some level of our recursion rather than go all the way to the base case.

As long as we are modifying the algorithm we should also generalize it so that is will work on any size matrices. The current strassen function described in Figure 1 will crash if given a matrix with odd row dimension or odd column dimension. We can easily fix this by padding matrices with a row of zeros in the case of an odd row dimension or by padding with a column of zeros in the case of an odd column dimension. Code for padding a single row or column can be found below.

```
"""add row of zeros to bottom of matrix"""
def padRow(m):
    x = []
    for i in range(len(m[0])):
      x.append(0)
    return(np.vstack((m,x)))

def padColumn(m):
"""add column of zeros to right of matrix"""
    x = []
    for i in range(len(m)):
      x.append(0)
    return(np.hstack((m,np.vstack(x))))
```

Since the padded rows (or columns) will need to be removed from the product at each level one might wonder whether padding once to a power of 2 would be more efficient? For example, a matrix with 17 rows and 17 columns will be padded to 18×18, but then each of its 9×9 submatrices will be padded to 10×10 which will require 5×5 submatrices to be padded and so on. Cases like this could be avoided by padding the original matrix to 32×32. This was tested however, and it was found that padding of a single row at multiple levels of recursion is considerably faster than padding to the next power of 2.

To ensure that the new version of Strassen based matrix multiplier shown below works as expected, more than a million matrix multiplications of various sizes and random values were computed and compared to Numpy.matmul to ensure both gave the same answer.

```
#x,y, are matrices to be multiplied. crossoverCutoff
#is the dimension where recursion stops.
def strassenGeneral(x, y,crossoverCutoff):
 #Base case when size <= crossoverCutoff
 if len(x) <= crossoverCutoff:
     return np.matmul(x,y)
 if len(x[0])<= crossoverCutoff:
     return np.matmul(x,y)

 rowDim = len(x)
 colDim = len(y[0])
 #if odd row dimension then pad
 if (rowDim & 1 and True):
     x = padRow(x)
     y = padColumn(y)

 #if odd column dimension then pad
 if (len(x[0]) & 1 and True):
    x = padColumn(x)
    y = padRow(y)
 if (len(y[0]) & 1 and True):
     y = padColumn(y)

 #split the matrices into quadrants.
 a, b, c, d = split(x)
 e, f, g, h = split(y)

 #Compute the 7 products, recursively (p1, p2...p7)
 if (len(x) > crossoverCutoff):
  p1 = strassenGeneral(a, f - h,crossoverCutoff)
  p2 = strassenGeneral(a + b, h,crossoverCutoff)
  p3 = strassenGeneral(c + d, e,crossoverCutoff)
  p4 = strassenGeneral(d, g - e,crossoverCutoff)
  p5 = strassenGeneral(a + d, e + h,crossoverCutoff)
  p6 = strassenGeneral(b - d, g + h,crossoverCutoff)
  p7 = strassenGeneral(a - c, e + f,crossoverCutoff)
 else:
  p1 = np.matmul(a, f - h)
  p2 = np.matmul(a + b, h)
  p3 = np.matmul(c + d, e)
  p4 = np.matmul(d, g - e)
  p5 = np.matmul(a + d, e + h)
  p6 = np.matmul(b - d, g + h)
  p7 = np.matmul(a - c, e + f)

 #combine the 4 quadrants into a single matrix
 c = np.vstack((np.hstack((p5+p4-p2+p6,p1+p2)),
         np.hstack((p3+p4,p1+p5-p3-p7))))

 x = len(c) - rowDim
 if (x > 0):
     c = c[:-x, :]  #delete padded rows
 x = len(c[0]) - colDim
 if (x > 0):
     c = c[:,:-x]  #delete padded columns

 return c
```

Timings of the Strassen Algorithm with Crossover to Numpy for Square Matrices

Before checking the performance on random inputs we check the performance on square matrices of size $2^n \times 2^n$ for various n. The results for the first machine which is a MacBook Pro 16 with a 6-Core Intel Core i7 at 2.6 GHz with 16GB of RAM is shown in Table 2. The column headings are given shorthand names but they can be described as follows. The Numpy column contains timings in seconds for Numpy.matmul. The Strassen column contains timings in seconds for the standard Strassen algorithm shown discussed above modified from [5]. The Strassen16, Strassen32, etc. columns represent timings from the Python code

	Numpy		Strassen 1		Standard Multiply	
Matrix Size	Time (seconds)	Current/ Previous	Time (seconds)	Current/ Previous	Time (seconds)	Current/ Previous
128x128	0.002	-	3.777	-	1.869	-
256x256	0.02	8.728	26.389	6.986	15.031	8.043
512x512	0.222	10.999	188.781	7.154	125.279	8.334

TABLE 1: *Timing for Base Algorithms on Matrices with Integer Entries. (Intel Core I7-9700 CPU @ 3.00 GHz, 8 Cores)*

Matrix Size	Numpy	Strassen	Strassen16	Strassen32	Strassen64	Strassen128	Strassen256	Strassen512	Standard
128 x 128	0.00	3.88	0.02	0.00	0.00	0.00	0.00	0.00	1.32
256 x 256	0.03	26.85	0.13	0.03	0.01	0.01	0.01	0.01	10.67
512 x 512	0.27	188.09	0.90	0.19	0.09	0.08	0.11	0.20	86.63
1024 x 1024	3.75	———	6.70	1.41	0.64	0.63	0.82	1.45	———
2048 x 2048	82.06	———	44.03	9.29	4.24	4.23	5.44	9.84	———
4096 x 4096	988.12	———	322.82	68.06	31.61	31.10	40.14	72.56	———
8192 x 8192	14722.33	———	2160.77	457.28	211.77	211.02	270.69	483.54	———

TABLE 2: *Timings (seconds) for Matrix Multiplication on Square Matrices with Integer Entries. MacBook Pro 16 with Core i7 @ 2.6 GHz*

Matrix Size	Numpy		Strassen		Strassen128		Standard	
	Time (s)	Current / Previous	Time (s)	Current / Previous	Time (s)	Current / Previous	Time (s)	Current / Previous
128 x 128	0.00		3.88		0.00		1.32	
256 x 256	0.03	11.30	26.85	6.93	0.01	7.39	10.67	8.07
512 x 512	0.27	10.20	188.09	7.00	0.08	7.48	86.63	8.12
1024 x 1024	3.75	13.69	———		0.63	7.72	———	
2048 x 2048	82.06	21.89	———		4.23	6.67	———	
4096 x 4096	988.12	12.04	———		31.10	7.35	———	
8192 x 8192	14722.33	14.90	———		211.02	6.78	———	

TABLE 3: *Timings (seconds) for Matrix Multiplication on Square Matrices with Integer Entries. MacBook Pro 16 with Core i7 @ 2.6 GHz*

Matrix Size	Numpy		Strassen		Strassen128		Standard	
	Time (s)	Current / Previous	Time (s)	Current / Previous	Time (s)	Current / Previous	Time (s)	Current / Previous
128 x 128	0.00		3.76		0.00		1.96	
256 x 256	0.02	8.80	27.67	7.36	0.01	6.96	15.60	7.95
512 x 512	0.22	10.77	183.88	6.64	0.10	7.06	124.48	7.98
1024 x 1024	1.94	8.97	1283.43	6.98	0.68	7.03	1002.26	8.05
2048 x 2048	77.42	439.91	8979.96	7.00	4.84	7.07	8426.06	8.41
4096 x 4096	760.60	9.82	63210.78	7.04	35.40	7.31	68976.25	8.19
8192 x 8192	7121.69	9.36	441637.97	6.99	239.26	6.76	549939.81	7.97

TABLE 4: *Timings (seconds) for Matrix Multiplication on Square Matrices with Integer Entries. Windows 11 with Core i7 @ 3.0 GHz*

Matrix Size	Numpy		Strassen		Strassen128		Standard	
	Time (s)	Current / Previous	Time (s)	Current / Previous	Time (s)	Current / Previous	Time (s)	Current / Previous
128 x 128	0.00		4.58		0.00		1.82	
256 x 256	0.03	9.56	32.71	7.14	0.02	7.91	15.11	8.29
512 x 512	0.45	17.77	228.34	6.98	0.11	6.76	122.98	8.14
1024 x 1024	4.21	9.38	———		0.78	7.26	———	
2048 x 2048	98.00	23.27	———		5.61	7.21	———	
4096 x 4096	1029.60	10.51	———		41.88	7.46	———	
8192 x 8192	10050.31	9.76	———		287.43	6.86	———	

TABLE 5: *Timings (seconds) for Matrix Multiplication on Square Matrices with Integer Entries. Linux with Xeon E5-2680 v3 @ 2.50GHz*

for `strassenGeneral` shown above with various crossover levels. The Standard column contains timings for the standard matrix multiplication algorithm previously discussed. We see in Table 2 that using a Strassen type algorithm and crossing over to Numpy when Matrix size is 128 gives a very slight advantage over crossing over at 64. Crossing over at larger or smaller values is slower than crossing over at size 128. We also see that not crossing over at all is even slower than the standard matrix multiplication for these sizes. Since the non-crossover Strassen algorithm and the standard matrix multiplication are not competitive and very slow, we stopped timing them after the 512×512 case because they would have taken a very long time to compute.

Table 3 is similar to Table 2 except we've removed all but the best crossover case for Strassen (crossover 128) and added columns to show the current time divided by the previous time. These latter columns are instructive because for Strassen we expect that if we double the size of the matrices the timing should increase seven-fold and it does. Similarly for the standard algorithm when we double the input size we expect the timing to increase eight-fold which it does. We don't exactly know what to expect for Numpy without closely examining the code, but we see that for the largest 2 cases when we double the size of the inputs the timing increases 12 to 15-fold. This suggests that if we further increase the size of the matrices that the Strassen type algorithm with a crossover at size 128 will continue to be much faster than the Numpy computation for square matrices with integer entries.

Normally, we would expect a matrix multiplication to increase no more than eight-fold when we double the inputs. This suggests that Numpy is tuned for matrices of size 128×128 or smaller. Alternatively, perhaps at larger sizes there are more cache misses in the Numpy algorithm. Without a close examination of the Numpy code it is not clear which is the case, but the point is that a divide and conquer algorithm such as Strassen combined with Numpy will perform better than Numpy alone on large matrices with integer entries.

Timings from a second machine are shown in Table 4. These timings are for the same experiment as above on a Windows 11 Machine with 3.0 GHz Core i7-9700 with 8 cores and 32 GB of RAM. In this case we see again that using a Strassen type algorithm that crosses over to Numpy at size 128 is considerably faster than using Numpy alone for large matrices with integer entries. Moreover we see that for the largest cases if we double the matrix size, the timings for the Strassen based algorithm will continue to grow seven-fold while the Numpy timings will grow ten-fold for each doubling of input-size.

Since both of these trials were based on Intel i7 chips, we ran a third experiment on a Linux machine with an Intel Xeon E5-2680 v3 @ 2.50GHz with 16 GB of RAM. Timings from this machine are in Table 5 and are similar to the previous tables.

Timings of the Strassen Algorithm with Crossover to Numpy for Arbitrary Matrices

Although the Python function `strassenGeneral` shown above will work for Arbitrary sized matrices, to this point we have only shown timings for square matrices $N \times N$ where N is a power of 2. The reason for this is that growth rates in timings when N increases are easier to track for powers of 2. However, to show that the Strassen type algorithm with crossover is viable in general we need to test for a variety of arbitrary sizes. For this experiment it is not possible to show the results in simple tables such as Table 2 through Table 5.

To motivate the next experiment consider the sample output shown below:

```
(1701 x 1267) * (1267 x 1678)
numpy (seconds)  15.43970187567
numpyDot (seconds)  15.08170314133
a @ b (seconds)  15.41474305465
strassen64 (seconds)  3.980883831158
strassen128 (seconds)  2.968686999753
strassen256 (seconds)  2.88325377367
DC64 (seconds)  6.42917919531
DC128 (seconds)  4.37878428772
DC256 (seconds)  4.12086373381

(1659 x 1949) * (1949 x 1093)
numpy (seconds)  33.79341135732
numpyDot (seconds)  33.8062295187
a @ b (seconds)  33.6903500761
strassen64 (seconds)  2.929703416
strassen128 (seconds)  2.54137444496
strassen256 (seconds)  2.75581365264
DC64 (seconds)  4.581859096884
DC128 (seconds)  4.08950223028
DC256 (seconds)  4.01872271299

(1386 x 1278) * (1278 x 1282)
numpy (seconds)  7.96956253983
numpyDot (seconds)  7.54114297591
a @ b (seconds)  8.81335245259
strassen64 (seconds)  2.425855960696
strassen128 (seconds)  1.823907148092
strassen256 (seconds)  1.74107060767
DC64 (seconds)  3.8810345549
DC128 (seconds)  2.672704061493
DC256 (seconds)  2.603429134935
```

This snippet of output shows three different matrix multiplies using three different variations of three different methods on

the Linux machine with the Xeon processor mentioned above. To illustrate what this output means consider the first block of output which represents a 1701×1267 matrix multiplied by a 1267×1678 matrix. The first three timings are variations of Numpy. The first is Numpy.matmul, the second is Numpy.dot and the third is called via the @ operator [4] which is really just an infix operator that should be the same as Numpy.matmul. The next three timings are for the Strassen type algorithm with crossover to Numpy at size 64, 128, and 256. The third set of timings are Divide and Conquer matrix multiplications that crossover to Numpy at size 64, 128, and 256. These latter three methods were added since much of the increase in efficiency of the Strassen type algorithms is due to their divide and conquer approach which allows us to compute Numpy multiplications on smaller matrices. We don't show the source code for this approach because it is not faster than the Strassen approach, however it can be produced with a simple modification of the code in `strassenGeneral`. The Strassen algorithm divides the first matrix into sub-matrices a,b,c,d and the second matrix into e,f,g,h and reassembles via seven clever products. The regular divide and conquer approach creates the final product as the four submatrices $a*e+b*g$, $a*f+b*h$, $c*e+d*g$, and $c*f+d*h$. This uses eight products but is more straightforward than Strassen and allows for recursively calling itself until crossing over to Numpy for the smaller products.

We note for the three arbitrary size matrix multiplies shown above that the Strassen based approaches are fastest, and the alternative divide and conquer approaches are two to three times faster than the Numpy method but slower than the Strassen method.

To create a good experiment we set three variables $dim1$, $dim2$, $dim3$ to random integers between 1000 and 8000 and then created two matrices one of size $(dim1 \times dim2)$ and the other of size $(dim2 \times dim3)$. Both were filled with random integers and multiplied using the 9 methods described above. We then put this experiment into a loop to repeat several thousand times. In actuality we stopped the experiment on the MacBook and the Windows machine after about 2 weeks and we stopped the Linux machine after a few hours because the latter machine is a shared machine used by students at Rowan and the timings are not accurate when it has many users.

The question is how do we present the results of several hundred such experiments on random sized matrices in a compact manner? Since we have a large number of different dimension multiplies they cannot easily be put into a table so instead we decided to organize the results by elapsed time. To see how consider Figure 2. We bin the `Strassen128` results into round number of seconds and we see the x-axis of Figure 2 shows the number of seconds of `Strassen128`. Let us consider the case of 102 seconds. The matrix multiply $(6977 \times 4737) * (4737 \times 7809)$ took 101.56 seconds using `Strassen128` and took 2482.76 seconds using Numpy. Meanwhile the matrix multiply $(7029 \times 7209) * (7209 \times 6283)$ using `Strassen128` took 101.80 seconds compared to 2792.11 seconds using Numpy. These are the only 2 cases that round to 102 seconds for `Strassen128` so they get bucketed together and averaged. The Average `Strassen128` time for these 2 cases is 101.68 seconds and the average Numpy time for these 2 cases is 2637.43 seconds. In the Figure we normalize by `Strassen128` so the `Strassen128` value for 102 seconds is 1.0 and the Numpy value for 102 seconds is $2637.43/101.68 = 25.94$. Thus for matrix multiplies that take 102 seconds for `Strassen128` the Numpy routines take almost 26 times as long which in this case is 44 minutes versus less than 2 for the `Strassen128` routine.

Now that we've described how Figure 2 is derived it is useful to describe several things shown by the Figure. First note that for large matrix multiplies that take at least 15 seconds for the Strassen type algorithm that crosses over at size 128, the regular Numpy algorithms all take at least 8 times as long and in some cases up to 30 times as long. Moreover the general trend is increasing so that if we tested even larger sizes we would expect the disparity to continue to increase. Another item to notice is there is really no difference between Numpy.matmul, Numpy.dot or the infix operator a@b as expected. Also notice that the Strassen algorithms with crossover are almost twice as fast as the more straightforward divide and conquer algorithm discussed above. The last item to notice is the crossing over at size 128 seems to work best, just as in the square cases of Table 2.

Figure 3 is similar to Figure 2 except these timings are done on the Windows 11 machine described above. Here we see that the Numpy algorithms take between 8 and 16 times as long as the Strassen type algorithm that crosses over to Numpy at size 128. One other difference between the Mac and Windows machine is that crossing over at size 64 is better than crossing over at size 128 more frequently on the Windows machine.

Since the run-time to compute these last 2 figures is more than several weeks, we did not repeat the experiment on the shared machine with the Xeon processor, however we did run it for several hours and the Strassen128 algorithm seems to be 8 to 16 times faster than Numpy for cases longer than 15 seconds just as with the Mac and Windows machines.

Conclusions

Numpy is a Python library which is widely used in the math and scientific community because of its speed. In this paper we presented a Strassen type algorithm that greatly improves on Numpy performance for large matrices with integer entries. For integer matrices with row dimension or column dimension in the thousands the algorithm can be 8 to 30 times faster than Numpy. The algorithm is the standard Strassen divide and conquer algorithm but it crosses over to Numpy when either the row or column dimension of one of the matrices drops below 128. The algorithm was tested on a MacBook, an I7 based Windows machine as well as a Linux machine running a Xeon processor with similar results. Although there is no apparent advantage for matrices with real entries, there are a number of applications for matrices with integer coefficients.

References

[1] Z. Fink, S. Liu, J. Choi, M. Diener, and L. V. Kale, "Performance evaluation of python parallel programming models: Charm4py and mpi4py," *2021 IEEE/ACM 6th International Workshop on Extreme Scale Programming Models and Middleware (ESPM2)*, pp. 38–44, 2021, https://doi.org/10.1109/ESPM254806.2021.00010.

[2] V. Strassen, "Gaussian elimination is not optimal," *Numerische Mathematik*, pp. 354–356, 1969, https://doi.org/10.1007/BF02165411.

[3] D. Harvey and J. V. der Hoeven, "On the complexity of integer matrix multiplication," *Journal of Symbolic Computation*, pp. 1–8, 2018, https://doi.org/10.1016/j.jsc.2017.11.001.

[4] Python.org, "Pep 465: A dedicated infix operator for matrix multiplication," Available at https://peps.python.org/pep-0465/, 2014.

[5] GeeksforGeeks, "Strassen's matrix multiplication - geeksforgeeks," Available at https://www.geeksforgeeks.org/strassens-matrix-multiplication/, 2022.

[6] T. Cormen, C. Leiserson, R. Rivest, and C. Stein, *Introduction to Algorithms*. Cambridge, MA: MIT Press, 2009.

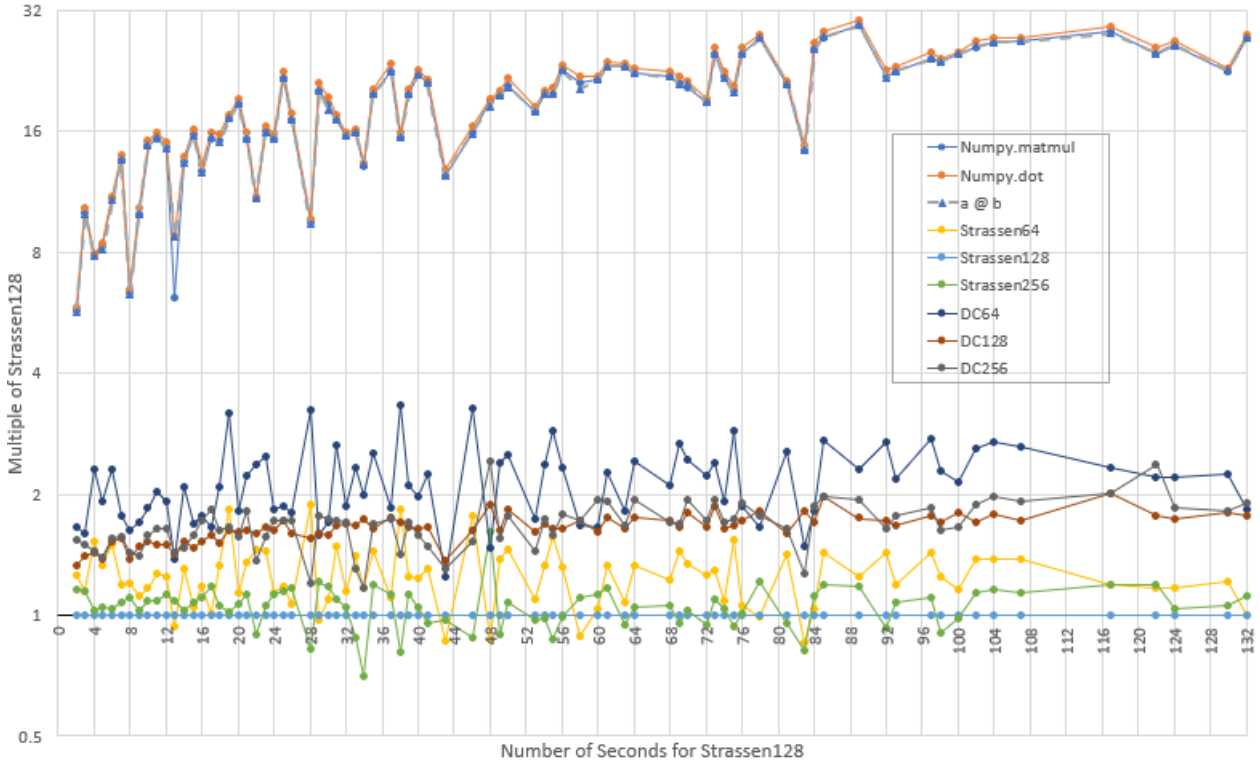

Fig. 2: *Timing of Multiple Algorithms Relative to Strassen128 on MacBook Pro 16 with Core i7 @ 2.6 GHz.*

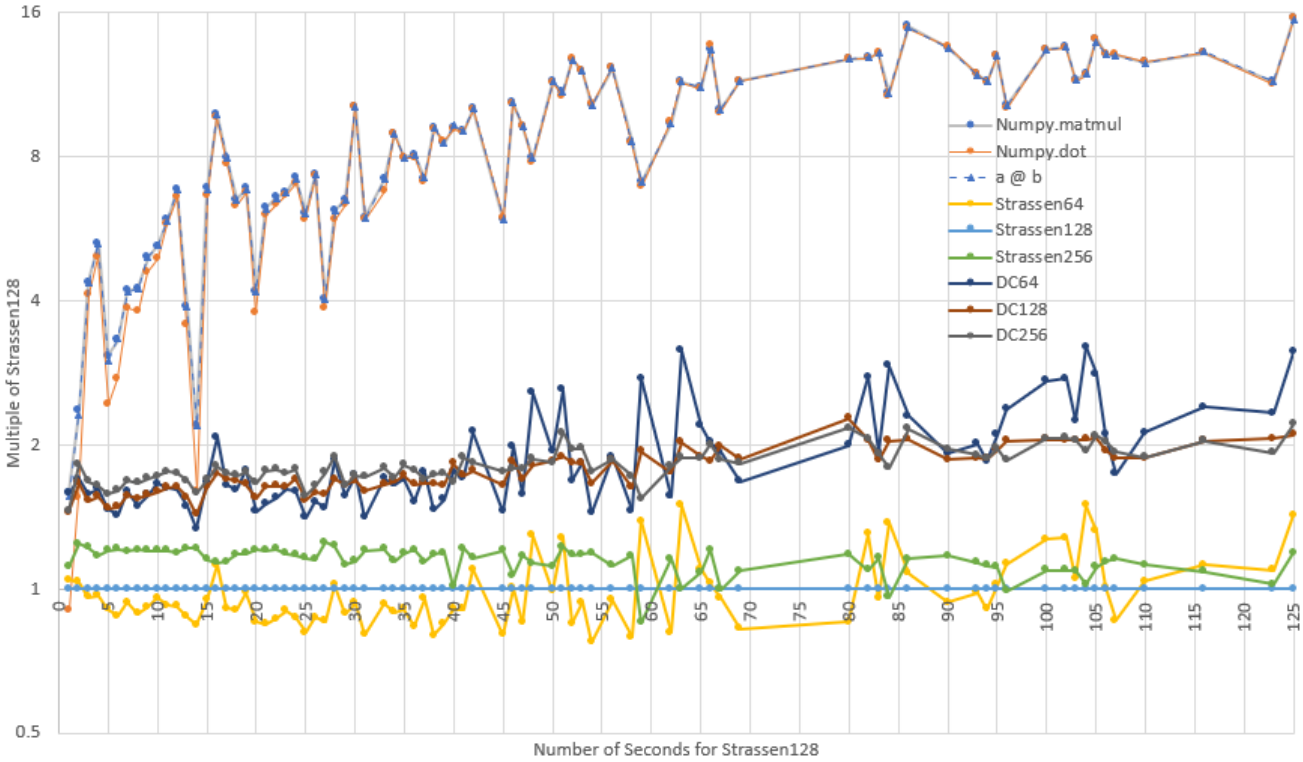

Fig. 3: *Timing of Multiple Algorithms Relative to Strassen128 on Windows 11 with Core i7 @ 3.0 GHz.*

An Accessible Python based Author Identification Process

Anthony Breitzman‡*

Abstract—Author identification also known as 'author attribution' and more recently 'forensic linguistics' involves identifying true authors of anonymous texts. The Federalist Papers are 85 documents written anonymously by a combination of Alexander Hamilton, John Jay, and James Madison in the late 1780's supporting adoption of the American Constitution. All but 12 documents have confirmed authors based on lists provided before the author's deaths. Mosteller and Wallace in 1963 provided evidence of authorship for the 12 disputed documents, however the analysis is not readily accessible to non-statisticians. In this paper we replicate the analysis but in a much more accessible way using modern text mining methods and Python. One surprising result is the usefulness of filler-words in identifying writing styles. The method described here can be applied to other authorship questions such as linking the Unabomber manifesto with Ted Kaczynski, identifying Shakespeare's collaborators, etc. Although the question of authorship of the Federalist Papers has been studied before, what is new in this paper is we highlight a process and tools that can be easily used by Python programmers, and the methods do not rely on any knowledge of statistics or machine learning.

Index Terms—Federalist, Author Identification, Attribution, Forensic Linguistics, Text-Mining

Introduction

Author attribution is a long-standing problem involving identifying true authors in anonymous texts. Recently the problem has garnered headlines with several high profile cases that were made possible with computers and text mining methods. In 2017 *The Discovery Channel* created a TV series called Manhunt:Unabomber that showed how Forensic Linguistics was used to determine that Ted Kaczynski was the author of the Unabomber manifesto [1]. In 2016 a headline from *The Guardian* shook the literary world: "Christopher Marlowe credited as one of Shakespeare's co-writers" [2]. It was long suspected that Shakespeare collaborated with others, but since Marlowe was always considered his biggest rival, it was quite a surprise that the two collaborated. See [3] for other examples including the best seller Primary Colors about the Clinton campaign that was published anonymously and the question of authorship of "Twas the Night Before Christmas" as well as other examples

* *Corresponding author: breitzman@rowan.edu*
‡ *Rowan University Department of Computer Science*

About this Paper

Purpose of this Paper

While forensic linguistics may be a recent name for such attribution, the idea of using statistical modeling to identify authors goes back to at least 1963 when Mosteller and Wallace published their ground-breaking study of the Federalist Papers [4]. Since that study was published in a Statistics journal, it requires a thorough understanding of statistics to understand it. Because our audience consists mostly of Software Engineers instead of Statisticians, we present a more accessible analysis of the Federalist Papers which can be applied to other author attribution problems. In this paper we endeavor to show a self-contained process that can be used for author attribution problems as well as other text-analysis problems such as gender identification of texts, genre classification, or sentiment analysis.

The Contribution of this Paper

The use of the Federalist Papers as a case study in author attribution is not new and dates to 1963 [4]. However, this paper's contribution is that it shows a process for author attribution and text mining in general that requires only knowledge of Python and requires no previous background in statistics or machine learning.

Outline of the Remaining Paper

We first describe how rudimentary author attribution was done before 1963. We then briefly describe the notion of Exploratory Data Analysis by way of a key table before showing the Python tools necessary for building said table. We then discuss how to build a dictionary of terms for each potential author and use Python to turn that dictionary into a set of probabilities that can be used as a Naive Bayes classifier. We then present a self-contained explanation of Naive Bayes and use it to predict the author of the disputed Federalist Papers. Finally, we show how a Python programmer who has little background in machine learning, can still successfully run numerous machine learning models to do predictions.

The Disputed Federalist Papers as a Case Study

This brief history is shortened from [4] which itself is a much shortened history from [5] and [6]. The Federalist Papers were a series of essays written by Alexander Hamilton, John Jay, and James Madison published under the pseudonym "Publius" in New York newspapers in 1787 and 1788 in support of ratification of the constitution. It is surmised that the authors were not anxious to claim the essays for decades because the opinions in the essays

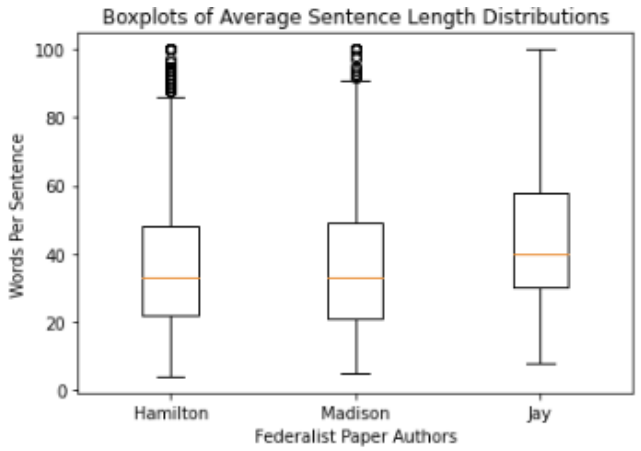

Fig. 1: *Boxplots showing Sentence Length Statistics Federalist Authors*

sometimes opposed positions each later supported [4]. Hamilton was famously killed in a duel in 1804 but he left a list of the essays he wrote with his lawyer before his death. Madison later generated his own list a decade later and attributed any discrepancies between the lists as "owing doubtless to the hurry in which (Hamilton's) memorandum was made out" [5]. Of the 85 essays, the 5 essays written by Jay are not in dispute. Another 51 by Hamilton, 14 by Madison, and 3 joint essays coauthored by Hamilton and Madison are also not in dispute. However, 12 essays (Federalist Nos. 49-58, 62 amd 63) were claimed by both Hamilton and Madison in their respective lists [4].

Similarities of Hamilton and Madison as Writers

Before Mosteller used advanced statistical modeling for author attribution, the standard approach was to look at things like sentence length to identify authors. In [7] Mosteller explains why this won't work with Hamilton and Madison because they are too similar.

> The writings of Hamilton and Madison are difficult to tell apart because both authors were masters of the popular Spectator style of writing-complicated and oratorical. Never use a short word if a long one will do. Double negatives are good, triple even better. To illustrate, in 1941 Frederick Williams and I counted sentence lengths for the undisputed papers and got means of 34.55 and 34.59 words, respectively, for Hamilton and Madison, and average standard deviations for sentence lengths of 19.2 and 20.3. [7]

To illustrate the quote above, consider the boxplot in Figure 1 of the non-disputed Federalist Papers of Hamilton, Madison, and Jay.

We see in Figure 1 that not only do Hamilton and Madison have the same median sentence length, but they have the same 25-percentile and 75-percentile sentence length and very similar minimum and maximum sentence lengths. In comparison John Jay tends to use longer sentences. In general, before 1963 this kind of analysis was used for author attribution, and it often works. However, as we see, Hamilton and Madison were very similar writers. The boxplot above is easily generated with matplotlib and a sentence tokenizer discussed below. We omit the code for space considerations, however all of the code discussed in this paper can be found at https://github.com/AbreitzmanSr/SciPy2023-AuthorAttribution.

Exploratory Data Analysis

Before jumping into modeling and code examples, we'll start with a key table that will suggest that Madison is the author of most if not all of the disputed papers. Table 1 contains a list of Hamilton's and Madison's favorite words. (Although John Jay is included in the table, he is not really of interest in this study because he has laid no claim to the disputed papers. The only reason the 12 papers are disputed is because both Hamilton and Madison had claimed authorship of them.)

Note that Hamilton uses "upon" many times in place of "on". In the disputed papers both terms are used at the Madison rate rather than the Hamilton rate.

Madison uses "whilst" instead of "while". While is never used in the disputed papers but "whilst" is used in half of them.

Several words like "democratic", "dishonorable", "precision", "inconveniency", etc. are not used in any Hamilton documents but are used in both the disputed papers and Madison documents.

"While", "enough", "nomination", "kind" appear in Hamilton documents but either not at all in the disputed papers or at the Madison rate within the disputed papers

Generating the previous table is an example of what Data Scientists call Exploratory Data Analysis which is an initial investigation on data to discover patterns and trends, spot anomalies, and generate statistical summaries which might help us check assumptions and perform hypotheses about our data.

The previous table suggests Madison is the likely author of most of the disputed Federalist Papers. But the table did materialize out of nowhere. There are 2 key components to the previous table: We need a method to identify words that have a high probability of being used by one author but not the other and we need a way to identify usage per 1000 words for each author

Both of those components are easily done using Python's NLTK (Natural Language Tool-kit) library [8].

Building the Favorite Words Table

Project Gutenberg [9] has the Federalist Papers as a plain-text e-book with each essay as an individual chapter. The Python code required to put the plain text of the book into a long string is below.

```
import re
from urllib import request

#utility functions for slicing text
def left(s, amount):
  return s[:amount]

def right(s, amount):
  return s[-amount:]

#Get Federalist Papers
url="https://www.gutenberg.org/cache/epub/1404/pg1404.txt"

response=request.urlopen(url)
raw=response.read()
text=raw.decode("utf-8-sig")

#replace multiple spaces with single space
text=re.sub("\s+", " ", text)

#kill all the front matter of the book
text=right(text,len(text)-text.find('FEDERALIST No.'))
```

	% of Papers Containing Word					Usage Per 1000 Words				
word	Hamilton	Madison	Joint	Disputed	Jay	Hamilton	Madison	Joint	Disputed	Jay
upon	100	21.4	66.6	16.6	20	3.012	0.161	0.312	0.112	0.107
on	98	100	100	100	100	3.037	6.817	6.094	7.077	4.721
very	72.5	85.7	100	91.6	60	0.583	1.04	0.937	2.209	1.394
community	62.7	14.2	33.3	25	20	0.558	0.046	0.156	0.187	0.107
while	39.2	0	0	0	40	0.291	0	0	0	0.214
enough	35.2	0	33.3	0	0	0.267	0	0.156	0	0
nomination	13.7	0	0	0	0	0.178	0	0	0	0
consequently	5.8	57.1	0	41.6	40	0.032	0.277	0	0.337	0.429
lesser	3.9	35.7	0	16.6	20	0.016	0.161	0	0.149	0.107
whilst	1.9	57.1	66.6	50	0	0.008	0.277	0.312	0.337	0
although	1.9	42.8	0	33.3	80	0.008	0.161	0	0.149	0.536
composing	1.9	42.8	33.3	16.6	0	0.008	0.254	0.156	0.074	0
recommended	1.9	35.7	0	8.3	20	0.008	0.138	0	0.037	0.429
sphere	1.9	35.7	0	16.6	0	0.008	0.184	0	0.112	0
pronounced	1.9	28.5	0	16.6	0	0.008	0.115	0	0.074	0
respectively	1.9	28.5	0	16.6	0	0.008	0.138	0	0.074	0
enlarge	0	28.5	0	16.6	0	0	0.115	0	0.074	0
involves	0	28.5	0	16.6	0	0	0.092	0	0.074	0
stamped	0	28.5	33.3	0	0	0	0.092	0.156	0	0
crushed	0	21.4	0	8.3	0	0	0.069	0	0.037	0
democratic	0	21.4	0	8.3	0	0	0.069	0	0.037	0
dishonorable	0	21.4	0	8.3	0	0	0.069	0	0.037	0
precision	0	21.4	0	8.3	0	0	0.069	0	0.037	0
reform	0	21.4	33.3	16.6	0	0	0.161	0.156	0.074	0
transferred	0	21.4	0	8.3	0	0	0.069	0	0.037	0
universally	0	21.4	0	8.3	20	0	0.069	0	0.037	0.107
bind	0	14.2	0	8.3	20	0	0.069	0	0.037	0.107
derives	0	14.2	33.3	8.3	0	0	0.069	0.156	0.037	0
drawing	0	14.2	0	8.3	0	0	0.069	0	0.037	0
function	0	14.2	0	8.3	0	0	0.069	0	0.037	0
inconveniency	0	14.2	0	16.6	0	0	0.069	0	0.074	0
obviated	0	14.2	0	8.3	0	0	0.069	0	0.037	0
patriotic	0	14.2	0	25	20	0	0.069	0	0.112	0.107
speedy	0	14.2	0	8.3	0	0	0.069	0	0.037	0

TABLE 1: *Favorite Words of Hamilton and Madison*

```
#kill back matter
text=left(text,
       text.find('*** END OF THE PROJECT GUTENBERG'))
```

Project Gutenberg [9] has the Federalist Papers stored as a book with the individual papers as chapters. In the next code snippet we reorganize the text so that each author's Federalist papers are contained in a list. For example the variable `hamilton` will contain a list of Hamilton's 51 known Federalist Papers.

```
#returns the main text of a Federalist paper.
def getFedText(s):
 if (len(s)>0):
  t = s + ' PUBLIUS' #additional sentinel in case
                     #it's not there.
                          #(in most cases it is)
  i = t.find('PUBLIUS')
  t = left(t,i)
  i = t.find('State of New York')
  t = right(t,len(t)-(i+19))
  return t.strip()
 else:
  return ""

#Break Federalist papers up into individual texts
FedChapters=re.split('\sFEDERALIST No\. \d*\s',' '+text)


#Store Hamilton's Federalist papers in a Hamilton
#list, Madison's in a Madison list, etc.
hamilton = []
jay = []
```

```
madison = []
joint = []
disputed = []
for i in range(len(FedChapters)):
  if (i in {2,3,4,5,64}):
   jay.append([i,[getFedText(FedChapters[i])]])
  else:
   if (i in {18,19,20}):
     joint.append([i,[getFedText(FedChapters[i])]])
   else:
     if (i in {49,50,51,52,53,54,55,56,57,58,62,63}):
        disputed.append(
                    [i,[getFedText(FedChapters[i])]])
     else:
       if (i in {10,14,37,38,39,40,41,42,43,
                 44,45,46,47,48}):
         madison.append(
                    [i,[getFedText(FedChapters[i])]])
       else:
        if (i > 0):
          hamilton.append(
                    [i,[getFedText(FedChapters[i])]])
```

Introduction to NLTK Tokenizers

NLTK [8] makes it easy to make lists of sentences, lists of words, count sentences, count words in sentences etc. Here's an example of how to first split a text into sentences and then make a Python list of each word in each sentence. (This could be done with split() but we would need multiple sentence delimiters and we would lose the punctuation if we weren't careful.)

```
from nltk.tokenize import sent_tokenize
from nltk.tokenize import word_tokenize

text_2sentences = "A short sentence. Another
                   short sentence."

sentences = sent_tokenize(text_2sentences)
for x in sentences:
  print(word_tokenize(x))


['A','short','sentence','.']
['Another','short','sentence','.']
```

We will leverage the NLTK word tokenizer to build dictionaries of word frequencies for each author.

```
from nltk.tokenize import word_tokenize

hamiltonDicts=[]#list of dictionaries containing
               #word freq for each of Hamilton's
               #Federalist Papers
madisonDicts=[]
disputedDicts=[]
jointDicts=[]

def getDocDict(str1):
#returns a dictonary containing frequencies of
#any word in string.
#e.g. str1 = 'quick brown fox is quick.'
# returns {quick:2, brown:1, fox:1, is:1}
x = {}
words = word_tokenize(str1.lower().strip())
for b in words:
      if b in x:
          x[b]+=1
      else:
          x[b]=1
return(x)

for a in hamilton:
  hamiltonDicts.append(getDocDict(a[1][0]))

for a in madison:
  madisonDicts.append(getDocDict(a[1][0]))

for a in joint:
  jointDicts.append(getDocDict(a[1][0]))

for a in disputed:
  disputedDicts.append(getDocDict(a[1][0]))
```

It is now straightforward to identify word usage for each author. That is, given a word such as "upon" it is easy to identify the percent of each author's Federalist papers that mention "upon." It's also easy to identify the usage of "upon" per thousand words for each author. What we haven't addressed is how to find words that are favorites of Hamilton but not Madison and vice-versa. We will do that by building a Naive Bayes dictionary for each author, but we will assume no prior knowledge of Naive Bayes to do so.

The code below creates a document frequency distribution of every word mentioned in the Federalist Papers. That is, for every word mentioned, we count how many documents the word appears in. We then remove any word that is only mentioned in one or two documents because it will have no discriminating value. Similarly we remove any word that appears in all documents because the only words mentioned in all documents are so-called stopwords like "is", "and","the" that both authors use. Note words like "while" and "whilst" might be considered stopwords, but these will be kept because they are used by only one of the authors and thus will not reach the 80 document threshold to be discarded.

```
completeDict={}#dictionary containing any word
               #mentioned in any of the Federalist
               #papers and the number of Federalist
               #Papers containing the word.

kills = [',','.',"'",'',';','-',')','(']
authDicts = [hamiltonDicts,madisonDicts,
             jointDicts,disputedDicts]
for authDict in authDicts:
 for a in authDict:
  for x in a:
      if (x not in kills):
       if x in completeDict:
          completeDict[x]+=1
       else:
          completeDict[x]=1

trimDict = set()  #subset of completeDict
                  #that contains useful words
for a in completeDict:
  x = completeDict[a]
  if (x >= 3 and x < 80):
      trimDict.add(a)

print(len(completeDict),len(trimDict))


8492 3967
```

At this point `completeDict` contains document frequencies for the 8,492 unique words in all the Federalist papers and `trimDict` contains the subset of 3,967 potentially useful words. We now need to find words that are much more likely to be used by Hamilton than Madison and vice-versa. For each word in `trimDict` we will compute the probability that Hamilton or Madison used it. The words where Hamilton's probability is 5+ times more likely than Madison (or vice-versa) is an interesting word that gets selected for the previously shown Table 1.

The code below will help us get each author's favorite words. For each word in `trimDict` we will count how often each author uses it. We next total up all of the word frequencies for each

author and store them in the denominators `hamiltonNBdenom` and `madisonNBdenom`.

```
#build Naive Bayes Dictionaries
#for Hamilton and Madison
hamiltonNBwordDicts = {}
madisonNBwordDicts = {}

hamiltonNBdenom = madisonNBdenom = 0

for a in trimDict: #this is equivalent
                   #to Laplace Smoothing
  hamiltonNBwordDicts[a]=madisonNBwordDicts[a]=1
  hamiltonNBdenom += 1
  madisonNBdenom += 1

for dictionary in hamiltonDicts:
 for word in dictionary:
  if (word in trimDict):
   hamiltonNBwordDicts[word]+=dictionary[word]
   hamiltonNBdenom +=dictionary[word]

for dictionary in madisonDicts:
 for word in dictionary:
  if (word in trimDict):
    madisonNBwordDicts[word]+=dictionary[word]
    madisonNBdenom += dictionary[word]
```

For those unfamiliar with Naïve Bayes we are just computing word frequencies of the potentially useful words for each author and making sure no word probability is 0. (This is called Laplace Smoothing, but essentially we're trying to avoid cases where Hamilton uses a word very few times but Madison uses it 0 times (or vice-versa) because that will pollute our table with a bunch of useless words.) We need a denominator (consisting of the sum of frequencies of all words) in order to compute a probability of an author using the word, then the probability of an author using that word is just the frequency of the word divided by the denominator.

It is now straightforward to identify words that are favorites of Hamilton but not Madison and vice-versa as follows:

```
interesting = []
tableData = []
j = 0
for i,a in enumerate(trimDict):
  h1 = hamiltonNBwordDicts[a]/hamiltonNBdenom
  m1 = madisonNBwordDicts[a]/madisonNBdenom
  if (m1/h1 > 5 or h1/m1 > 5):
    interesting.append(a)
    if (j < 10):
       tableData.append([a,m1/h1,h1/m1])
       j+=1

from tabulate import tabulate
print (tabulate(tableData,
     headers=["FavoriteWord","Mad. Pr/Ham. Pr",
              "Ham.  Pr/Mad. Pr"]))
```

FavoriteWord	Mad. Pr/Ham. Pr	Ham. Pr/Mad. Pr
enumeration	6.08567	0.164321
surely	10.1428	0.0985923
defined	5.07139	0.197185
whilst	16.482	0.0606722
respectively	8.87493	0.112677
address	5.07139	0.197185
usurped	5.07139	0.197185
while	0.12191	8.20279
obviated	5.79072	0.17269
upon	0.0557395	17.9406

We of course cut off the table of "interesting" words because of space considerations. As expected, we see that the the probability of "whilst" being used by Madison is 16 times as likely as it being used by Hamilton. Similarly, "upon" being used by Hamilton is 18 times as likely as it being used by Madison. To get the table of author favorite words shown above in Table 1 we just need to calculate the percentage of papers from each author that contain the words, and also compute the usage per 1000 words for each author. Both of those calculations are straightforward so we omit the code, however it can be found at at https://github.com/AbreitzmanSr/SciPy2023-AuthorAttribution.

Naive Bayes Model

We now have everything we need to build a model to predict the author of the disputed Federalist Papers. We assume no prior knowledge of Naive Bayes, but the interested reader can see [10] or many other books for a full derivation. For our purposes we only care that: $P(Author|word1, word2, \ldots, wordN) = P(word1|Author) * P(word2|Author) * \ldots * P(wordN|author)/k$. That is, the conditional probability that a paper ($word1$ through $wordN$) is authored by Hamilton or Madison is equal to the product of the probabilities of each word belonging to the authors then divided by a constant k. (The equality is only true if the words are independent. Since we don't care about the actual probabilities, but only which author has the larger value, we don't need independence.) The constant k is actually another probability that is hard to compute, but since it's the same for both authors all we really need is the following pseudocode:

```
Text = [word1, word2, ..., wordN]
if (P(word1|Hamilton)*P(word2|Hamilton)*...*
   P(wordN|Hamilton) >
   P(word1|Madison)*P(word2|Madison)*...*
   P(wordN|Madison)):
      return(Hamilton)
else:
      return(Madison)
```

The actual Python code shown below is slightly different than the pseudocode above. Since we are computing the product of thousands of very small values there is a risk of underflow so instead of the product of many small constants we compute the sum of the logs of many small constants (e.g. Log(a*b) = Log(a) + Log(b)). Thus, the Python code looks like the following:

```
import math
#given a document return 'hamilton' if NaiveBayes prob
#suggests Hamilton authored it. similarly return
#'madison' if he is the likely author
def NB_federalist_predict(docDict,vocab1=trimDict):
 h_pr = m_pr = 0
 for word in docDict:
  if (word in vocab1):
   h_pr += float(docDict[word])*(math.log(
        hamiltonNBwordDicts[word]/hamiltonNBdenom))
   m_pr += float(docDict[word])*(math.log(
        madisonNBwordDicts[word]/madisonNBdenom))

 if (h_pr > m_pr):
    return('hamilton')
 else:
    return('madison')

def check_accuracy(vocab1=trimDict):
  right = wrong = 0
  for a in hamiltonDicts:
    if NB_federalist_predict(a,vocab1)=='hamilton':
      right+=1
    else:
      wrong+=1
```

```
  for a in madisonDicts:
    if NB_federalist_predict(a,vocab1)=='madison':
      right+=1
    else:
      wrong+=1
  return([100*right/(right+wrong),right,wrong])

  print('% correct:',check_accuracy()[0])
```

```
% correct: 100.0
```

The `NB_federalist_predict` is a Naive Bayes classifier which takes in a document dictionary such as the elements in `hamiltonDicts` or `madisonDicts` we defined earlier in the paper. We check the accuracy of the classifier with the straightforward function `check_accuracy` that simply looks at the predictions for all the known Hamilton papers and all the known Madison papers and counts the correct and erroneous author predictions.

The classifier will work with any vocabulary but defaults to `trimDict` if no vocabulary is provided. We will see below that this allows us to run the classifier on various word lists which may be useful for our analysis.

The last line shows that the Naive Bayes classifier correctly predicts 100% of the undisputed papers from Hamilton and Madison. The next thing to check is the 12 disputed papers and see if they are attributed to Madison as the authors in [4] found. For those familiar with machine learning or data mining we call the known Federalist papers, the "training" set and the disputed papers the "test" set.

Predicting Authors for the Disputed Papers

We saw how the predict function works above on the undisputed papers. Now to see how various word sets can be used to predict who wrote the disputed papers consider the code and output below:

```
#the following checks accuracy on the training set and
#then identifies how many of the disputed papers are
#by each author
def Federalist_report(words=trimDict):
  if (len(words)<10):
      print(words)
  else:
      temp = words[:9]
      temp.append('...')
      print(temp)
  print(str(check_accuracy(words)[0])+'% accuracy')
  madison = hamilton = 0
  for a in disputedDicts:
      if (NB_federalist_predict(a,words)=='madison'):
          madison+=1
      else:
          hamilton+=1
  print("disputed papers: madison:"+str(madison)+
        ', hamilton:'+str(hamilton)+'\n')

Federalist_report(interesting)
Federalist_report(['although','composing','involves',
       'confederation','upon'])
Federalist_report(['although','obviated','composing',
       'whilst','consequently','upon'])
Federalist_report(['against','within','inhabitants',
       'whilst','powers','upon','while'])
Federalist_report(['against','upon','whilst',
       'inhabitants','within'])
Federalist_report(['against','within','inhabitants',
       'whilst','upon'])
Federalist_report(['against','while','whilst','upon',
                   'on'])
Federalist_report(['concurrent','upon','on',
       'very','natural'])
Federalist_report(['while','upon','on','inconveniency'])
```

```
['enumeration', 'surely', 'whilst', 'respectively',
 'relief', 'reform', 'jury', 'dishonorable',
 'term', '...']
100.0% accuracy
disputed papers: madison:12, hamilton:0

['although', 'composing', 'involves', 'confederation',
 'upon']
100.0% accuracy
disputed papers: madison:12, hamilton:0

['although', 'obviated', 'composing', 'whilst',
 'consequently', 'upon']
96.92307692307692% accuracy
disputed papers: madison:12, hamilton:0

['against', 'within', 'inhabitants', 'whilst', 'powers',
 'upon', 'while']
100.0% accuracy
disputed papers: madison:12, hamilton:0

['against', 'upon', 'whilst', 'inhabitants', 'within']
96.92307692307692% accuracy
disputed papers: madison:12, hamilton:0

['against', 'within', 'inhabitants', 'whilst', 'upon']
96.92307692307692% accuracy
disputed papers: madison:12, hamilton:0

['against', 'while', 'whilst', 'upon', 'on']
96.92307692307692% accuracy
disputed papers: madison:12, hamilton:0

['concurrent', 'upon', 'on', 'very', 'natural']
98.46153846153847% accuracy
disputed papers: madison:12, hamilton:0

['while', 'upon', 'on', 'inconveniency']
95.38461538461539% accuracy
disputed papers: madison:12, hamilton:0
```

The `Federalist_report` function shown above does two things. It shows the vocabulary we are using to test on. It checks the accuracy on the undisputed Federalist Papers (the training set) and then counts how many of the disputed papers (the testing set) the Naive Bayes model attributes to Madison and Hamilton. We see that for several different subsets of the author Favorite words from Table 1 the model suggests Madison is the author of all 12 of the disputed papers. We also see that for each word-set the accuracy is at least 95% with several word-sets yielding 100% accuracy.

More Advanced Models

Our hand-built Naive Bayes model was useful for showing how to build a probability dictionary which was useful for our exploratory data analysis and ultimately the model was sufficient for identifying Madison as the likely author of the disputed papers. However, Python programmers have an excellent library for running more sophisticated models called Scikit-learn [11]. The advantage of the Scikit-learn library is it has numerous built-in models that all take the same parameters. Thus we can prepare the data set once and run multiple models without needing to know how the underlying machine learning models work.

Below we show how to run multiple models using only the words "against," "within," "inhabitants," "whilst," and "upon" on the undisputed and disputed Federalist Papers in less than 50 lines of code.

```
def mPercent(results):
 mcount = 0
 tcount = 0
 for a in results:
  if (a == 'm'):
   mcount+=1
   tcount+=1
 print('% Disputed attributed to Madison:',
        100.0*mcount/tcount,"\n")


"""
Build and test multiple models via SKlearn.
X is a dataframe consisting of known Hamilton
  and Madison papers.
y is a data frameconsisting of author labels.
X_test is a dataframe consisting of disputed
papers
"""
smallVocab5 = ['against','within','inhabitants',
               'whilst','upon']
tfidf = sklearn.feature_extraction.text.
        TfidfVectorizer(analyzer="word",
        binary=False,min_df=2,
        vocabulary=smallVocab5)

X_transformed = tfidf.fit_transform(X)
lb = sklearn.preprocessing.LabelEncoder()
y_transformed = lb.fit_transform(y)
X_test_transformed = tfidf.transform(X_test)

models = [
 KNeighborsClassifier(3),
 DecisionTreeClassifier(max_depth=5),
 RandomForestClassifier(n_estimators=25,max_depth=3),
 LinearSVC(),
 SVC(gamma=2, C=1),
 ComplementNB(),
 AdaBoostClassifier()
]

CV = 5
cv_df = pd.DataFrame(index=range(CV * len(models)))
for model in models:
 model_name = model.__class__.__name__
 accuracies = cross_val_score(model, X_transformed,
            y_transformed,scoring='accuracy',cv=CV)
 avgAccur = 0
 for fold_idx, accuracy in enumerate(accuracies):
   print(model_name,"fold:",fold_idx,
              "accuracy:",str(accuracy)[:5])
 print(model_name,"avg accuracy:",
         str(accuracies.mean())[:5])
 model.fit(X_transformed, y_transformed)
 y_final_predicted=model.predict(X_test_transformed)
 y_final_predicted_labeled=
    lb.inverse_transform(y_final_predicted)
 mPercent(y_final_predicted_labeled)
```

```
KNeighborsClassifier fold: 0 accuracy: 1.0
KNeighborsClassifier fold: 1 accuracy: 1.0
KNeighborsClassifier fold: 2 accuracy: 1.0
KNeighborsClassifier fold: 3 accuracy: 1.0
KNeighborsClassifier fold: 4 accuracy: 1.0
KNeighborsClassifier avg accuracy: 1.0
% Disputed attributed to Madison: 100.0

DecisionTreeClassifier fold: 0 accuracy: 1.0
DecisionTreeClassifier fold: 1 accuracy: 0.846
DecisionTreeClassifier fold: 2 accuracy: 1.0
DecisionTreeClassifier fold: 3 accuracy: 1.0
DecisionTreeClassifier fold: 4 accuracy: 1.0
DecisionTreeClassifier avg accuracy: 0.969
% Disputed attributed to Madison: 100.0

RandomForestClassifier fold: 0 accuracy: 1.0
RandomForestClassifier fold: 1 accuracy: 0.846
RandomForestClassifier fold: 2 accuracy: 1.0
RandomForestClassifier fold: 3 accuracy: 1.0
RandomForestClassifier fold: 4 accuracy: 1.0
RandomForestClassifier avg accuracy: 0.969
% Disputed attributed to Madison: 100.0

LinearSVC fold: 0 accuracy: 1.0
LinearSVC fold: 1 accuracy: 1.0
LinearSVC fold: 2 accuracy: 1.0
LinearSVC fold: 3 accuracy: 1.0
LinearSVC fold: 4 accuracy: 1.0
LinearSVC avg accuracy: 1.0
% Disputed attributed to Madison: 100.0

SVC fold: 0 accuracy: 1.0
SVC fold: 1 accuracy: 1.0
SVC fold: 2 accuracy: 1.0
SVC fold: 3 accuracy: 1.0
SVC fold: 4 accuracy: 1.0
SVC avg accuracy: 1.0
% Disputed attributed to Madison: 100.0

ComplementNB fold: 0 accuracy: 0.923
ComplementNB fold: 1 accuracy: 1.0
ComplementNB fold: 2 accuracy: 1.0
ComplementNB fold: 3 accuracy: 1.0
ComplementNB fold: 4 accuracy: 1.0
ComplementNB avg accuracy: 0.985
% Disputed attributed to Madison: 100.0

AdaBoostClassifier fold: 0 accuracy: 1.0
AdaBoostClassifier fold: 1 accuracy: 0.846
AdaBoostClassifier fold: 2 accuracy: 1.0
AdaBoostClassifier fold: 3 accuracy: 1.0
AdaBoostClassifier fold: 4 accuracy: 1.0
AdaBoostClassifier avg accuracy: 0.969
% Disputed attributed to Madison: 100.0
```

The code snippet above puts multiple Scikit-learn models [11] into a list and loops through each. Inside the loop a 5-fold cross validation is run on the training data consisting of all known Hamilton and Madison essays. (This just means that we randomly cut the training set into 5 slices (called folds) and test on each fold individually while using the remaining folds for training the model.)

The models are then run on the disputed papers and a function called `mPercent` is called that calculates how many of the disputed papers were written by Madison.

We note that the 5-fold cross validation is 100% accurate for each fold for the K-Nearest Neighbors model, and the Support-Vector classifiers. For the other models 4 out of 5 folds were 100% accurate and overall the models were 97% accurate or better. All of the models predicted that the disputed papers were written by Madison.

Note Scikit-learn offers multiple Naive Bayes classifiers. The Complement Naive Bayes model was chosen above because it was empirically shown by [12] to outperform other Naive Bayes models on text classification tasks.

One Last Simple Model

We've seen several subsets of Table 1 that accurately identify the authors of the known Federalist papers and also identify Madison as the author of the disputed papers. The reader may be wondering what is the smallest set of words that can be used to make such predictions? From Table 1 it's clear that "while", "whilst", and "upon" can mostly distinguish between papers authored by Hamilton or Madison. The use of "while" suggests Hamilton, while the use of "whilst" often suggests Madison, particularly if the rate is above 0.25 mentions per 1,000 words. If neither "while,"

or "whilst" is mentioned we can look for "upon." Both authors use "upon", but if the rate of "upon" is at 0.9 mentions per 1,000 words or above, then it is almost certainly authored by Hamilton.

The description above can be made into a very simple decision tree. A decision tree can be made into a series of if-then statements, yielding the simple model below.

```
#return usage rate per 1000 words of a target word
#e.g. if target=='upon' appears 3 times in a 1500
#word essay, we return a rate of 2 per 1000 words.
def rate_per_1000(docDict,target):
 if (target in docDict):
     wordCount=0
     for a in docDict:
         wordCount+=docDict[a]
     return(1000*docDict[target]/wordCount)
 else:
     return(0)

#given a document dictionary, predict if it was
#authored by Hamilton or Madison
def federalist_decison_tree(docDict):
 if ('while' in docDict):
     return('hamilton')
 else:
     if (rate_per_1000(docDict,'whilst') >= .25):
             return('madison')
     if (rate_per_1000(docDict,'upon') >= .9):
             return('hamilton')
     else:
             return('madison')
```

The simple model above is 100% accurate on the known documents, and predicts Madison as the author of the 12 disputed documents. In general, it is not recommended that we base an attribution on only three words because of a potential of overfitting, but it's interesting that these two authors that are rather similar in style, can be differentiated with such a simple model.

Conclusions

In this brief paper we presented a number of ways to solve the problem of disputed author identification. First we did some exploratory data analysis using each author's favorite words. We showed that the steps to build a Naive Bayes dictionary were useful in helping us to find those favorite words. We built a Naive Bayes model that suggested that James Madison is the likely author of the disputed Federalist Papers. We showed how the Scikit-learn [11] library could be used to build and test numerous models very quickly and easily and noted that each of these models also point to Madison as the author. Finally, we built a very simple decision tree using only the words "while," "whilst," and "upon" which also points to Madison as the author. Note that while this is a case-study of the Federalist Papers, the methods shown here can easily be applied to other author identification problems or other text-mining tasks where we need to tokenize and explore large bodies of text.

REFERENCES

[1] C. Luu, "Fighting words with the unabomber," *JSTOR.org*. [Online]. Available: https://daily.jstor.org/fighting-words-unabomber/

[2] D. Alberge, "Christopher marlowe credited as one of shakespeare's co-writers," *The Guardian*, 2016. [Online]. Available: https://www.theguardian.com/culture/2016/oct/23/christopher-marlowe-credited-as-one-of-shakespeares-co-writers

[3] D. Foster, *Author Unknown: On the Trail of Anonymous*. Henry Holt and Company, 2000.

[4] F. Mosteller and D. L. Wallace, "Inference in an authorship problem," *Journal of the American Statistical Association*, pp. 275–309, 1963, https://doi.org/10.2307/2283270. [Online]. Available: https://www.jstor.org/stable/2283270?origin=JSTOR-pdf

[5] D. Adair, "The authorship of the disputed federalist papers," *The William and Mary Quarterly*, pp. 97–122, 1944, https://doi.org/10.2307/1921883.

[6] ——, "The authorship of the disputed federalist papers: Part ii," *The William and Mary Quarterly*, p. 235–264, 1944, https://doi.org/10.2307/1923729.

[7] F. Mosteller, "A statistical study of the writing styles of the authors of "the federalist" papers," *Proceedings of the American Philosophical Society*, 1987. [Online]. Available: https://www.jstor.org/stable/986786

[8] E. Loper and S. Bird, "Nltk: The natural language toolkit," 2002.

[9] A. Hamilton, J. Jay, and J. Madison, "The project gutenberg ebook of the federalist papers," Available at https://www.gutenberg.org/cache/epub/1404/pg1404.txt.

[10] D. Jurafsky and J. Martin, *Speech and Language Processing (Draft of 3rd Ed.)*, 2023.

[11] F. Pedregosa *et al.*, "Scikit-learn: Machine learning in python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[12] J. Rennie, L. Shih, J. Teevan, and D. Karger, "Tackling the poor assumptions of naive bayes text classifiers," *ICML*, vol. 3, pp. 616–623, 2023.

Biomolecular Crystallographic Computing with Jupyter

Blaine H. M. Mooers[‡§¶‖*]

Abstract—The ease of use of Jupyter notebooks has helped biologists enter scientific computing, especially in protein crystallography, where a collaborative community develops extensive libraries, user-friendly GUIs, and Python APIs. The APIs allow users to use the libraries in Jupyter. To further advance this use of Jupyter, we developed a collection of code fragments that use the vast *Computational Crystallography Toolbox* (*cctbx*) library for novel analyses. We made versions of this library for use in JupyterLab and Colab. We also made versions of the snippet library for the text editors VS Code, Vim, and Emacs that support editing live code cells in Jupyter notebooks via the GhostText web browser extension. Readers of this paper may be inspired to adapt this latter capability to their domains of science.

Index Terms—literate programming, reproducible research, scientific rigor, electronic notebooks, JupyterLab, Jupyter notebooks, Colab notebook, OnDemand notebooks, computational structural biology, computational crystallography, biomolecular crystallography, protein crystallography, biomolecular structure, computational molecular biophysics, biomedical research, data visualization, scientific communication, GhostText, text editors, snippet libraries, SciPy software stack, interactive software development

Introduction

Biomolecular crystallography involves the determination of the molecular structure of proteins and nucleic acids and their complexes by using X-rays, neutrons, or electrons. The molecular structure determines the protein's biological function, so the experimentally determined structures provide valuable insights vital for understanding biology and developing new therapies in medicine. The recent *resolution revolution* in cryo-electron microscopy (cryo-EM) [1] and the breakthrough in protein structure prediction with neural networks now provide complementary sources of insights into biomolecular structure [2], [3], [4]. However, the crystallographic approach continues to play a vital role because it still supplies the most precise structures, [5].

About half of the crystal structures of protein molecules are refined with the program *Phenix* [6]. This program has a user-friendly GUI that supports standard analyses [7]. Phenix runs on top of *cctbx* [8]. The Computational Crystallography Toolbox (*cctbx*) provides a transparent API, so most users of *Phenix* are barely aware that it relies on *cctbx*. However, nonstandard analyses are not available in *Phenix* and require accessing the functions in the *cctbx* library (e.g., [9]). The backend *cctbx* was written in C++ in the early 2000s for speed and to provide customized data structures for crystallography. Likewise, the GUI-driven *Olex2* small molecule refinement program uses *cctbx* for many of its crystallographic computations [10].

To ease the use of *cctbx* by general users, the C++ interfaces, classes, and functions of *cctbx* are exposed to Python via the *Boost.Python* Library [11]. Recently, dependency management in *cctbx* was reworked by leveraging the Anaconda infrastructure to ease its installation. In spite of these conveniences, the widespread adoption of Python by field practitioners over the past decade, and the presence of several on-line tutorials about *cctbx*, many structural biologists still find *cctbx* hard to master and adoption has remained low. This difficulty drove several groups to develop software libraries (e.g. *reciprocalspaceship* [12], *GEMMI* [13]) that reinvent some features of *cctbx* while utilizing the more familiar *pandas* DataFrames in place of *cctbx*'s customized data structures. In contrast to these new competitors, *cctbx* has more extensive coverage of advanced crystallographic data analysis tasks and is more thoroughly tested as the result running underneath Phenix for almost two decades. *cctbx* remains the ultimate library for building advanced crystallographic data analyses tools, so the field would benefit if *cctbx* were easier to use.

To foster adoption of *cctbx*, we present a collection of *cctbx* code snippets to be used in Jupyter notebooks [14]. Jupyter provides an excellent platform for exploring the *cctbx* library and prototyping new analysis tools. The Python API of *cctbx* simplifies running *cctbx* in Jupyter via a kernel specific for its conda environment. We formatted our snippet library for several snippet extensions for the Classic Notebook and for Jupyter Lab. To overcome the absence of tab triggers in the Jupyter ecosystem to invoke the insertion of snippets, we also made the snippets available for leading text editors. The user can use the GhostText browser plugin to edit the contents of a Jupyter cell in a full-powered external editor. GhostText enables the user to experience the joy of interactive computing in Jupyter while working from the comfort of their favorite text editor. These multiple modalities of using *cctbx* in Jupyter that we describe below may inspire workers in other domains to build similar snippet libraries for domain-specific software.

Results

We provide a survey of the snippet library that we have customized for several snippet extensions in JupyterLab and Google Colab.

* *Corresponding author: blaine-mooers@ouhsc.edu*
‡ *Department of Biochemistry and Molecular Biology, University of Oklahoma Health Sciences Center, Oklahoma City, OK 97104*
§ *Stephenson Cancer Center, University of Oklahoma Health Sciences Center, Oklahoma City, OK 97104*
¶ *Laboratory of Biomolecular Structure and Function, University of Oklahoma Health Sciences Center, Oklahoma City, OK 97104*
‖ *Biomolecular Structure Core, Oklahoma COBRE in Structural Biology, University of Oklahoma Health Sciences Center, Oklahoma City, OK 97104*

jupyterlabcctbxsnips

We developed the *jupyterlabcctbxsnips* library of code templates for the JupyterLab extension *jupyterlab-snippets* (`https://github.com/QuantStack/jupyterlab-snipp`) Access to the code templates or snippets requires the editing of the Jupyter notebook from inside of JupyterLab, a browser-based IDE for displaying, editing, and running Jupyter notebooks.

JupyterLab supports more comprehensive workflows for academic work than what is possible in the Classic Jupyter Notebook application. For example, it enables the writing or editing of a document in a pane next to the Jupyter notebook. This variant is useful for writing documentation, protocols, tutorials, blog posts, and manuscripts next to the notebook that is being described. The document can be a plain text, html, markdown, LaTeX, or even an org-mode file if one activates the text area with GhostText while running one of several advanced text editors (see the section below about GhostText). The editing of a document next to the related Jupyter notebook supports reproducible research and reduces costly context switching.

We made a variant of the library, *jupyterlabcctbxsnipsplus* (`https://github.com/MooersLab/jupyterlabcctbxsnipsp`) that has a copy of the code in a block comment (Fig. 1). In the commented code, suggested sites for editing are indicated by tab stops that are marked with dollar signs.

```
[ ]: """
     from iotbx import mtz
     mtz_obj = mtz.object(file_name="/Users/blaine/${1:3nd4}.mtz")
     mtz_obj.show_summary()
     """

     from iotbx import mtz
     mtz_obj = mtz.object(file_name="/Users/blaine/3nd4.mtz")
     mtz_obj.show_summary()

     # Description:  Read mtz file into a mtz object and print summary.
     # Source:  NA
```

Fig. 1: *A snippet from the* jupyterlabcctbxsnipsplus *library with duplicate code in a comment block. The dollar sign marks the start of a tab stop. The comment block guides the editing of the active code.*

The figure below (Fig. 2) shows part of the cascading menus for the *jupyerlabcctbxsnipsplus* library after it has been installed successfully. The submenus correspond to the names of subfolders in the *cctbx+* folder in the snippets folder, which was manually created inside of the Jupyter folder in the local library folder (i.e., `~/Library/Jupyter/multimenus_snippets/cctbx+` on macOS).

Each final menu item is linked to a Python snippet file. The selection of a snippet file by clicking on it with the left-mouse button inserts its content into a new cell below the current cell.

In contrast, the *mtzOjbectSummary.py* snippet was selected from the *cctbx* submenu and lacks the comment block. This code was inserted in the current notebook cell (Fig. 3). The code in this cell was be executed by entering **Shift-Enter**.

The *mtzObjectSummary.py* snippet prints a summary of an mtz file. A mtz file is a binary file that contains diffraction data in a highly customized data structure. The data in this mtz file has columns of I(+) and I(-). These are the Bijvoet pairs of diffraction intensities. These pairs are related by symmetry and should have equal intensity values within experimental error. The differences in intensities are a measure of the presence of

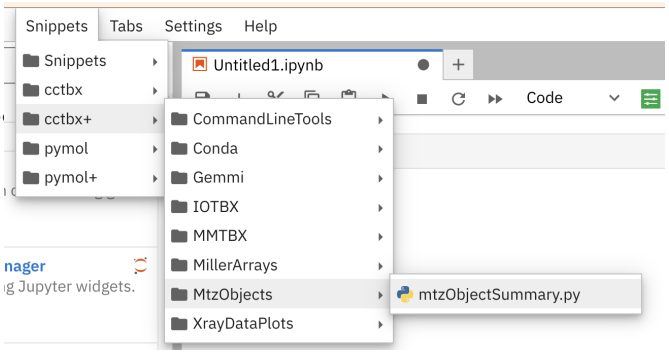

Fig. 2: *The cascading menus for the* jupyterlabcctbxsnipsplus *library for the jupyterlab-snippets version 0.4.1 extension in JupyterLab version 3.5.2.*

```
[11]: from iotbx import mtz
      mtz_obj = mtz.object(file_name="/Users/blaine/3hz7.mtz")
      mtz_obj.show_summary()

Title: phenix.cif_as_mtz
Space group symbol from file: P43212
Space group number from file: 96
Space group from matrices: P 43 21 2 (No. 96)
Point group symbol from file: 422
Number of crystals: 2
Number of Miller indices: 6482
Resolution range: 25.5941 1.99683
History:
Crystal 1:
  Name: HKL_base
  Project: HKL_base
  Id: 0
  Unit cell: (46.046, 46.046, 82.811, 90, 90, 90)
  Number of datasets: 1
  Dataset 1:
    Name: HKL_base
    Id: 0
    Wavelength: 0
    Number of columns: 0
Crystal 2:
  Name: crystal_0
  Project: project
  Id: 2
  Unit cell: (46.046, 46.046, 82.811, 90, 90, 90)
  Number of datasets: 1
  Dataset 1:
    Name: dataset
    Id: 1
    Wavelength: 0.97976
    Number of columns: 11
    label   #valid  %valid    min       max type
    H         6482 100.00%   0.00     22.00 H: index h,k,l
    K         6482 100.00%   0.00     16.00 H: index h,k,l
    L         6482 100.00%   0.00     41.00 H: index h,k,l
    F(+)      6448  99.48%   0.00    476.80 G: F(+) or F(-)
    SIGF(+)   6448  99.48%   0.00     10.19 L: standard deviation
    F(-)      5005  77.21%   0.00    393.03 G: F(+) or F(-)
    SIGF(-)   5005  77.21%   0.00      8.88 L: standard deviation
    I(+)      6448  99.48% -76.20 227334.00 K: I(+) or I(-)
    SIGI(+)   6448  99.48%   4.00   4014.00 M: standard deviation
    I(-)      5005  77.21% -98.90 154474.00 K: I(+) or I(-)
    SIGI(-)   5005  77.21%  10.20   2081.00 M: standard deviation
```

Fig. 3: *The code and output from the* mtzObjectSummary.py *snippet in JupyterLab.*

anomalous scattering. Anomalous scattering can be measured for elements like sulfur and phosphorus that are part of the native protein and nucleic acid structures and heavier elements like metals that are naturally occurring as part of metalloproteins or that were purposefully introduced by soaking crystals or that were incorporated covalently into the protein (e.g., selenomethionine) or nucleic acid (e.g., 5-bromouracil) during its synthesis.

The anomalous differences can be used to determine the positions of the anomalous scattering atoms. Once the positions of the anomalous scatterers are known, it is possible to work out the positions of the lighter atoms in the protein. We use these data to make the I(+) vs I(-) scatter plot below (Fig. 4). The mtz file contains data for SirA-like protein (DSY4693) from *Desultobacterium hafniense*, Northeast Structural Genomics Consortium Target DhR2A. The diffraction data were retrieved

from the Protein Data Bank, a very early open science project that recently celebrated its 50th anniversary [15].

The I(+) vs I(-) plot was made after reading the X-ray data into a *cctbx* Miller array, a data structure designed for handling X-ray data in *cctbx*. The I(+) and I(-) were eventually read into separate lists. We plot the two lists against each other in a scatter plot using *matplotlib* [16]. There is no scatter from the $x = y$ line in this plot if there is no anomalous signal. The larger the anomalous signal, the greater the scatter. The departure from this line is expected to be greater for intensities of large magnitude.

[12]:
```
%matplotlib inline
import matplotlib.pyplot as plt
import matplotlib as mpl
import matplotlib.ticker as ticker
from matplotlib.ticker import MultipleLocator #, FormatStrFormatter
from matplotlib.ticker import FuncFormatter
from iotbx.reflection_file_reader import any_reflection_file

# >>> change the mtz file name
hkl_file = any_reflection_file('3hz7.mtz')
miller_arrays = hkl_file.as_miller_arrays(merge_equivalents=False)
Iobs = miller_arrays[1]
i_plus, i_minus = Iobs.hemispheres_acentrics()
ipd = i_plus.data()
ip=list(ipd)
imd = i_minus.data()
im = list(imd)
len(im)

comma_fmt = FuncFormatter(lambda x, p: format(int(x), ','))

mpl.rcParams['savefig.dpi'] = 600
mpl.rcParams['figure.dpi'] = 600

# Set to width of a one column on a two-column page.
# May want to adjust settings for a slide.
fig, ax = plt.subplots(figsize=[3.25, 3.25])
ax.scatter(ip,im,c='k',alpha=0.3,s=5.5)
ax.set_xlabel(r'I(+)',fontsize=12)
ax.set_ylabel(r'I(-)',fontsize=12)
ax.xaxis.set_major_locator(MultipleLocator(50000.))
ax.yaxis.set_major_locator(MultipleLocator(50000.))
ax.get_xaxis().set_major_formatter(comma_fmt)
ax.get_yaxis().set_major_formatter(comma_fmt)

plt.xticks(fontsize=12)
plt.yticks(fontsize=12)
ax.grid(False)

# >>> change name of the figure file
plt.savefig('3hz7IpIm.pdf',bbox_inches='tight')
```

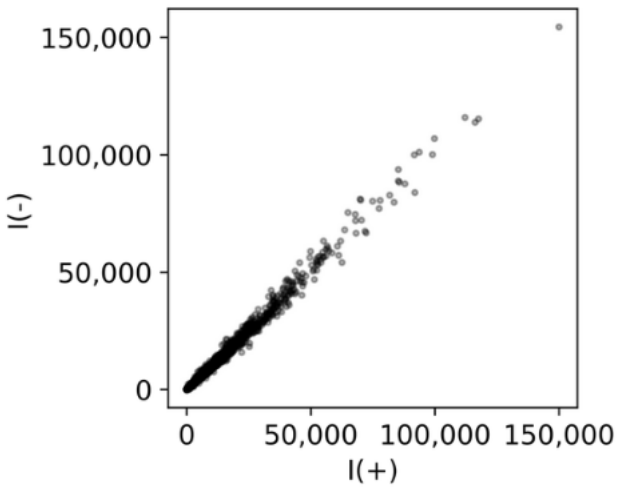

Fig. 4: *The code snippet to generate a Ip versus Im plot and the corresponding plot generated by the code.*

Plots of this nature are useful for detecting very weak anomalous signals from native anomalous scatters like sulfur and phosphorus. The collection of the anomalous signal from native scatters enables structure determination without having to spend the extra time and money to introduce heavier atoms that are not native to the protein. The measurement of the very weak signal from native anomalous scatterers is still at the edge of what is technically possible. It has rarely been achieved with in-house instruments. Success generally requires the faster multi-million dollar detectors at beamlines, tunable wavelengths of synchrotron radiation available at one of 30+ laboratories around the world, and cryogenic temperatures (-173 C) maintained by a cryostream of nitrogen gas that slows radiation damage long enough to collect complete datasets.

However, recently, several groups have completed successful native phasing experiments at room temperature by collecting data from large numbers of crystals and merging the data [17], [18]. The advantages of room temperature data collection include avoidance of conformational changes in the protein induced by supercooling the crystal. The room temperature data were collected from each crystal briefly before radiation damage degraded the diffraction too much. This is a remarkable achievement because the merging of diffraction data from many crystals in various orientations enhances the experimental error; this error can mask the weak anomalous signal that is being sought.

The plot (Fig. 4) was adapted from an example in the *reciprocalspaceship* project from the Hekstra Lab [12]. This new project takes a more Pythonic approach than *cctbx* by utilizing many of the packages in the SciPy stack that did not exist when *cctbx* was initiated. For example, it uses the *pandas* package to manage diffraction data whereas *cctbx* uses a custom C++ data structure for diffraction data that predates *pandas* by almost a decade. The utilization of *pandas* enables easier integration with the other components of the SciPy software stack including machine learning packages.

The *cctbx* is most easily installed into its own environment by using Anaconda with the command `conda create -n my_env -c conda-forge cctbx-base python=3.11`.

The atomic coordinates of the biomolecular structures are the other major type of data that are intimately associated with diffraction data. The fixed file format of Protein Data Bank coordinate files with the file extension of *pdb* originated in the 1970s with the birth of the Protein Data Bank, but very large biological macromolecules have been determined over the past two decades that exceeded the limits on the number of atoms permitted in one file. To address this and other shortcomings of the PDB file format, the PDBx/mmCIF (Protein Data Bank Exchange macromolecular Crystallographic Information Framework) file format recently became the new data standard [19]. The *cctbx* has been adapted to read mmCIF files.

taggedcctbxsnips

The Elyra-snippets extension for Jupyter Lab supports the use of tagged snippets (`https://elyra.readthedocs.io/en/latest/user_guide/co`
Each snippet is in a separate JavaScript file with the *json* file extension 5.

Each snippet file has a set of metadata. These data include a list of tags. The tags are used to find the snippet while editing a Jupyter notebook in JupyterLab. We made a version of the cctbxsnips library for the Elyra-snippets extension (`https://github.com/MooersLab/taggedcctbxsnips`).

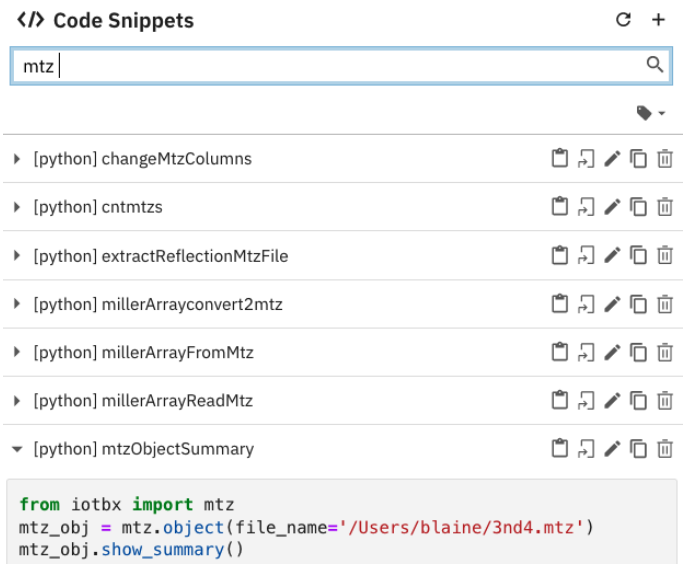

Fig. 5: *Snapshot of a list of snippets in JupyterLab supported by the Elyra-snippet extension. The 80* cctbx *snippets have been narrowed to seven snippets by entering the* `mtz` *tag. Additional tags can be entered to further narrow the list of candidate snippets.*

To add a new snippet, click on the + in the upper right of the Code Snippets icon (Fig. 6). This will open new GUI (see below) for creating a snippet. The value of *Name* should be one word or a compound word. The value of *Description* describes in one or more sentences what the snippet does. The values of the *Tags* field are used to narrow the listing of snippets in the menu. The value of the *Source* is the programming language; the value is Python in this example. The *Code* can be entered by selecting code in a notebook cell or copying and pasting from a script file.

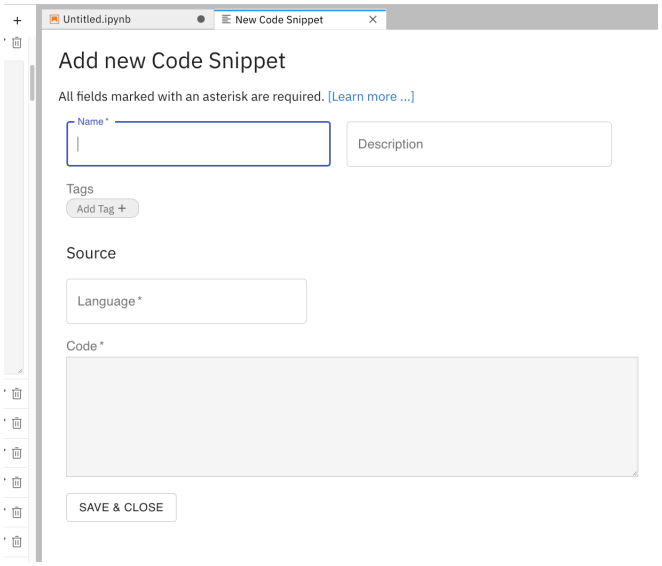

Fig. 6: *The GUI to create a new snippet via the Elyra-snippet extension for JupyterLab.*

colabcctbxsnips

The Google Colab notebook enables the running of software on Google's servers in a computational notebook that resembles the Jupyter notebook. Colab notebooks are useful for workshop settings where there is no time for installing software on a heterogeneous mix of operating systems when the attendees are following the presentation by using their own computers.

Colab notebooks do no support external extensions, but they have built-in support for snippets. A particular snippet library is stored in a dedicated Google Colab notebook rather than in individual files. The notebook of snippets is stored on the user's Google Drive account. While the software installed in a Colab session is lost upon logging out, the snippets remain available on the next login.

After the snippet notebook is installed, the user opens a new notebook to use the snippets. From that new notebook, the list of snippets will be exposed by clicking on the **<>** icon in the left margin of the notebook. Click on the `Insert` button in the upper righthand corner of the snippet to copy the snippet to the current code cell in the notebook.

We developed the *colabcctbxsnips* library and stored it in a Colab Notebook (`https://github.com/MooersLab/colabcctbxsnips`). Two snippets have the code for installing *mamba* and then *cctbx* (Fig. 7). These code snippets have to be run before *cctbx* can be accessed. The two code fragments require less than two minutes to install the software.

Fig. 7: *Snippets from the* cctbx *library for installing* mamba *and then* cctbx *on Google Colab.*

The Colab snippet system also lacks support for tab triggers and tab stops. We address this problem by supplying a copy of the snippet with the sites of the tab stops marked up like a yasnippet snippet. Unlike the case of the *jupyterlabcctbxsnipsplus* library, the marked up copy of the code snippet is displayed only in the preview of the snippet and is not inserted into the code cell along with the active code (Fig. 8).

Snippets for OnDemand Notebooks at HPCs

We have also worked out how to deploy this snippet library in OnDemand notebooks at High-Performance Computing centers. These notebooks resemble Colab notebooks in that JupyterLab extensions cannot be installed. However, they do not have any alternate support for accessing snippets from menus in the GUI. Instead, we had to create IPython magics for each snippet that load

millerArrays mtz2array Insert

- Read a mtz file into a miller array.
- The $ below marks a site for editing.

```
from iotbx.reflection_file_reader import any_reflection_file
hkl_file = any_reflection_file("${1:3hz7.mtz}")
miller_arrays = hkl_file.as_miller_arrays(merge_equivalents=False)
```

```
from iotbx.reflection_file_reader import any_reflection_file
hkl_file = any_reflection_file("3hz7.mtz")
miller_arrays = hkl_file.as_miller_arrays(merge_equivalents=False)
```

View source notebook

Fig. 8: *Preview of a Colab code snippet. The preview contains two copies of the code. The bottom copy of the code will be inserted into the current code cell. The top copy of the code serves as a guide to sites to be edited. The dollar sign marks the start of a tab stop where the enclosed placeholder value may need to be changed.*

the snippet's code into the code cell. This system would also work on Colab and may be preferred by expert users because the snippet names used to invoke the Ipython magic are under autocompletion. We offer a variant library that inserts a commented out copy of the code that has been annotated with the sites that are to be edited by the user.

cctbxsnips *for leading text editors*

To support the use of the *cctbx* code snippets in text editors, we made versions of the library for Emacs, Vim, Neovim, Visual Studio Code, Atom, and Sublime Text3. We selected these text editors because they are the most advanced and most popular with software developers and because they are supported by the GhostText project described below .

For Emacs, we developed a library for use with the yasnippets package (`https://github.com/MooersLab/cctbxsnips-Emacs`). Emacs supports repel-driven software development, which resembles the interactive software development experience in Jupyter notebooks. Emacs also supports the use of literate programming in several kinds of documents, including the very popular org-mode document [20]. Code blocks in org documents can be given a **jupyter** option with a Jupyter kernel name that enables running a specific Jupyter kernel including one mapped to a conda environment that has the *cctbx* package installed. A similar example using the molecular graphics package PyMOL is demonstrated in this short video (`https://www.youtube.com/watch?v=ZTocGPS-Uqk&t=2s`).

Using GhostText to edit Jupyter cells from a favorite text editor

By adding the GhostText extension (`https://ghosttext.fregante.com/`) to the web browser and a server to one of several leading text editors, it is possible to send the text from the browser through a WebSocket to a server in the text editor. Thus, it is possible to edit the contents of a computational notebook cell from inside a text editor. Changes made in the text editor instantly appear in the notebook and vice versa. By applying the power of a text editor to computational notebooks, experienced developers can continue to use familiar editing commands and tools in their preferred text editor.

GhostText is a Javascript program developed by Federico Brigante, a prolific JavaScript developer. Versions of the extension are available for the Google Chrome, Firefox, Edge, Opera, and Safari. The extension for the Google Chrome browser works in the Brave browser, and the extension for Firefox works in the Waterfox and Icecat browsers. GhostText was developed initially for Sublime Text 3, so Sublime Text 3 can serve as a positive control even if another editor in the list is the favored editor. (Sublime Text 3 is available for most platforms for a free trial period of infinite length.)

The snippet extensions for the Classic Jupyter Notebook and JupyterLab lack support for tab triggers to insert snippets as you type and tab stops inside the snippet to advance to sites in the snippet that may need to be edited. These two features are standard in the software that supports the use of snippet libraries in most text editors.

As a quick reminder, tab triggers in text editors insert chunks of computer code after the user enters the tab trigger name and hits the TAB key (Fig. 9). The tab trigger name can be as short as several letters. Many text editors and IDEs have pop-up menus that aid the selection of the correct tab trigger. Tab stops are sites within the code snippet where the cursor advances to after entering TAB again. These sites often have placeholder values that can be either edited or accepted by entering TAB again. Sites with identical placeholder values can be mirrored so that a change in value at one site is propagated to the other tab stops with the same placeholder value. The absence of tab stops can increase the number of bugs introduced by the developer by overlooking parameter values in the code snippet that need to be changed to adapt the snippet to the current program.

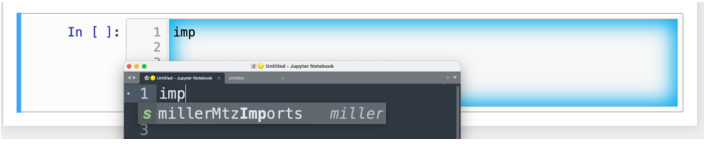

Fig. 9: *Example of a tab trigger being entered in Sublime Text 3 editor and appearing in a Jupyter Notebook cell. A pop-up menu lists the available snippets. The list was narrowed to one snippet by the entry of three letters.*

The text editor also needs to be extended with a server that enables two-way communication with the web page via a WebSocket. Edits made on the browser side of the WebSocket are immediately sent to an open page in the Text Editor and vice versa; however, the text editor's snippets and other editing tools only work in the text editor. The connection can be closed from either side of the WebSocket. It is closed on the web browser side via an option in GhostTest's pulldown menu, and it closed on the text editor side by closing the active buffer.

Here, we describe the setup for Emacs as an example of configuring a text editor to use GhostText. The server for *Emacs* is provided by the *atomic-chrome* package that is available in both the Milkypostman's Emacs Lisp Package Archive (MELPA) and on GitHub (`https://github.com/alpha22jp/atomic-chrome`). The configuration for *atomic-chrome* in the Emacs initialization file (e.g., init.el) is listed below (Fig. 10). The third line in Code listing 1 sets the default Emacs mode (equivalent to a programming language scope): We set it to Python for Jupyter code cells. Atomic-chrome uses text-mode by default. You can change the default mode to other programming languages that you may use in Jupyter, like Julia or R. The last three lines specify the Emacs mode to be used when text is imported from the text

areas on github.com, Overleaf.com, and 750words.com. Similar configuration options are available in the other text editors, or you manually change the language scope for the window with the text imported from Jupyter.

```
(use-package atomic-chrome)

(atomic-chrome-start-server)

(setq atomic-chrome-default-major-mode 'python-mode)

(setq atomic-chrome-extension-type-list '(ghost-text))

(setq atomic-chrome-server-ghost-text-port 4001)

(setq atomic-chrome-url-major-mode-alist

      '(("github\\.com" . gfm-mode)

        ("overleaf.com" . latex-mode)

        ("750words.com" . latex-mode)))
```

Fig. 10: *Emacs lisp code to configure the atomic-chrome package for Emacs. This configuration opens Jupyter notebooks in the Python major mode and the 750words.com webpage in the LaTeX major mode.*

GhostText provides keyboard shortcuts to improve productivity. These shortcuts keep the developer's hands on the keyboard and avoid breaks in context by moving the hand to the mouse. The shortcut by operating system is as follows: macOS, command-shift-K; Linux, control-shift-H; and Windows, control-shift-K.

To support the use of *GhostText* to edit electronic notebooks containing code from the *cctbx* library, we have made variants of a collection of *cctbx* snippets for *Visual Studio Code*, *Atom*, *Sublime Text 3*, *Vim*, *NeoVim*, and *Emacs*. For *Vim* and *NeoVim*, the snippets are available for the *UltiSnips*, *Snipmate*, and *neosnippets* plugins. The snippets are available for download on GitHub (`https://github.com/MooersLab`). From our experience, Sublime Text 3 has the easiest setup while Emacs provides the highest degree of customization. The *cctbx* snippet library was previously only available for use in Jupyter notebooks via extensions for the Classic Jupyter Notebook application or Jupyter Lab.

Note that the snippet library cannot be used with the program *nteract* (`https://nteract.io/`). The *nteract* is an easy-to-install and use desktop application for editing and running Jupyter notebooks offline. The ease of installation makes the *nteract* application popular with new users of Jupyter notebooks. Obviously *nteract* is not browser-based, so it cannot work with *GhostText*. *nteract* has yet to be extended to support the use of code snippet libraries, but *nteract* allows the switching of jupyter kernels between code cells.

While the focus of this report is on Jupyter and Colab notebooks, the *cctbxsnips* snippet library can be used to aid the development of Python scripts in plain text files, which have the advantage of easier version control. The snippets can also be used in other kinds of literate programming documents that operate off-line like org-mode files in Emacs and the *Quarto* (`http://quarto.org`) markdown representation of Jupyter notebooks. *Quarto* is available for several leading text editors. In the later case, you may have to extend the scope of the editing session in the editor to include Python source code.

Discussion

What is new

We report a set of code template libraries for doing biomolecular crystallographic computing in Jupyter. These template libraries only need to be installed once because they persist between logins.

We also include support for Colab notebooks where the snippets also persist between logins but other installed software is lost upon logging out of a session. The templates include the code for installing the software required for crystallographic computing. The installation templates automate as many as seven installation steps. Once the user runs the installation code at the top of a given Colab notebook, the user only needs to rerun these blocks of code upon logging into Colab to be able to reinstall the software during later sessions. The user can also modify the installation templates to install the crystallographic software on their local machine and then run the notebook in Jupyter Classic and JupyterLab. The template libraries presented here lower an important barrier to the use of Colab by those interested in crystallographic computing on the cloud.

We also report the use of GhostText to edit notebook code cells in Jupyter notebooks and text documents in JupyterLab. This capability enables a user to use an external text editor to edit code. The user can thereby take advantage of the support for tab triggers and tab stops in the external editor. This support can ensure faster and more accurate writing and editing of new code.

Relation to other work with snippet libraries

This snippet library is among the first that is domain specific. Most snippet libraries are for programming languages or for hypertext languages like HTML, markdown, and LaTeX. The average snippet in these libraries also tends to be quite short, and the sizes of the libraries tend to be small. The audience for these general purpose libraries are the millions of professional programmers and web page developers. We reasoned that domain-specific snippet libraries with long code fragments are a great coding tool that should be brought to the aid of the tens of thousands of workers in biological crystallography.

The other area where domain-specific snippets have been provided is in molecular graphics. A pioneering scripting wizard provided templates for use in the early molecular graphics program RasMol [21]. In addition, the conscript program provided a converter from *RasMol* to *PyMOL* [22]. We also provided snippets for *PyMOL*, which has about 100,000 users, for use in text editors [23] and Jupyter notebooks [24]. The former supports tab triggers and tab stops; the latter does not.

Opportunities for interoperability

The code template libraries can encourage synergistic interoperability between software packages. That is, the development of notebooks that use two or more software packages and even two or more programming languages. More general and well-known examples of interoperability include the Cython module in Python that enables the running of C++ code inside Python [25], the *reticulate* library that enables the running of Python code in R [26], and the *PyCall* package in Julia that enables the running of the Python packages in Julia (`https://github.com/JuliaPy/PyCall.jl`). The latter package is widely used to run *matplotlib* in Julia. Interoperability already occurs in computational crystallography between *CCP4* [27], *clipper* [28], *GEMMI* [13], *reciprocalspaceship* [12],

Careless [29], and *cctbx* and to a limited extent between *cctbx* and *PyMOL*. The snippet libraries reported here can promote taking advantage of this interoperability in Jupyter and Colab notebooks. We hope that our effort will help raise awareness of interoperability issues among the community.

Snippets in the age of AI-assisted autocompletion

Snippet libraries of domain specific software may not be as redundant as they first appear in the age of chatbots. The code fragments of domain-specific libraries have a limited presence on GitHub, so they may be underrepresented in large language models. In addition, chatbots are designed to return text rather than code. However, *copilot* and *tabnine* were designed for code completion and are good at autosuggesting code fragments. Via GhostText, it is possible to run *copilot* or *tabnine* in a text editor while editing Jupyter notebook cells.

Conclusions

Our explorations suggest that code snippets for domain-specific software libraries have several roles to play in supporting the use of such libraries. First, the snippets illustrate possible uses of the library, thereby, playing educational and inspirational roles. Second, the snippets can speed up the assembly of scripts while reducing the time spent on debugging, thereby, playing a productivity enhancement role. We hope that the *cctbxnsips* library will inspire the creation of similar libraries in other domains.

Acknowledgments

This work was supported in part by the following grants: Oklahoma Center for the Advancement of Science and Technology HR20-002, National Institutes of Health grants R01 CA242845, P30 CA225520, and P30 AG050911-07S1. In addition, we thank the Biomolecular Structure Core of the NIH supported Oklahoma COBRE in Structural Biology (PI: Ann West, P20 GM103640 and P30 GM145423).

References

[1] W. Kühlbrandt, "The resolution revolution," *Science*, vol. 343, no. 6178, pp. 1443–1444, 2014, https://doi.org/10.1126/science.1251652.

[2] J. Jumper, R. Evans, A. Pritzel, T. Green, M. Figurnov, O. Ronneberger, K. Tunyasuvunakool, R. Bates, A. Žídek, A. Potapenko *et al.*, "Highly accurate protein structure prediction with alphafold," *Nature*, vol. 596, no. 7873, pp. 583–589, 2021, https://doi.org/10.1038/s41586-021-03819-2.

[3] M. Mirdita, K. Schütze, Y. Moriwaki, L. Heo, S. Ovchinnikov, and M. Steinegger, "Colabfold: making protein folding accessible to all," *Nature Methods*, vol. 19, pp. 1–4, 2022, https://doi.org/10.1038/s41592-022-01488-1.

[4] R. Chowdhury, N. Bouatta, S. Biswas, C. Floristean, A. Kharkar, K. Roy, C. Rochereau, G. Ahdritz, J. Zhang, G. M. Church *et al.*, "Single-sequence protein structure prediction using a language model and deep learning," *Nature Biotechnology*, vol. 40, no. 11, pp. 1617–1623, 2022.

[5] A. Foerster and C. Schulze-Briese, "A shared vision for macromolecular crystallography over the next five years," *Structural Dynamics*, vol. 6, no. 6, p. 064302, 2019, https://doi.org/10.1063/1.5131017.

[6] D. Liebschner, P. V. Afonine, M. L. Baker, G. Bunkóczi, V. B. Chen, T. I. Croll, B. Hintze, L.-W. Hung, S. Jain, A. J. McCoy *et al.*, "Macromolecular structure determination using x-rays, neutrons and electrons: recent developments in phenix," *Acta Crystallographica Section D: Structural Biology*, vol. 75, no. 10, pp. 861–877, 2019, https://doi.org/10.1107/S2059798319011471.

[7] N. Echols, R. W. Grosse-Kunstleve, P. V. Afonine, G. Bunkóczi, V. B. Chen, J. J. Headd, A. J. McCoy, N. W. Moriarty, R. J. Read, D. C. Richardson *et al.*, "Graphical tools for macromolecular crystallography in phenix," *Journal of Applied Crystallography*, vol. 45, no. 3, pp. 581–586, 2012, https://doi.org/10.1107/S0021889812017293.

[8] R. W. Grosse-Kunstleve, N. K. Sauter, N. W. Moriatry, and P. D. Adams, "The computational crystallography toolbox: crystallographic algorithms in a reusable software framework," *Journal Application Crystallography*, vol. 35, no. 1, pp. 126–136, 2002, https://doi.org/10.1107/S0021889801017824.

[9] E. De Zitter, N. Coquelle, P. Oeser, T. R. Barends, and J.-P. Colletier, "Xtrapol8 enables automatic elucidation of low-occupancy intermediate-states in crystallographic studies," *Communications Biology*, vol. 5, no. 1, p. 640, 2022, https://doi.org/10.1038/s42003-022-03575-7.

[10] L. J. Bourhis, O. V. Dolomanov, R. J. Gildea, J. A. Howard, and H. Puschmann, "The anatomy of a comprehensive constrained, restrained refinement program for the modern computing environment–olex2 dissected," *Acta Crystallographica Section A: Foundations and Advances*, vol. 71, no. 1, pp. 59–75, 2015, https://doi.org/10.1107/S2053273314022207.

[11] D. Abrahams and R. W. Grosse-Kunstleve, "Building hybrid systems with boost.python," *C/C++ Users Journal*, vol. 21, no. 7, 2003. [Online]. Available: https://www.osti.gov/biblio/815409

[12] J. B. Greisman, K. M. Dalton, and D. R. Hekstra, "Reciprocalspaceship: A python library for crystallographic data analysis," *Journal of Applied Crystallography*, vol. 54, no. 5, 2021, https://doi.org/10.1101/2021.02.03.429617.

[13] M. Wojdyr, "Gemmi: A library for structural biology," *Journal of Open Source Software*, vol. 7, no. 73, p. 4200, 2022, https://doi.org/10.21105/joss.04200.

[14] T. Kluyver, B. Ragan-Kelley, F. Pérez, B. Granger, M. Bussonnier, J. Frederic, K. Kelley, J. Hamrick, J. Grout, S. Corlay, P. Ivanov, D. Avila, S. Abdalla, C. Willing, and J. development team, "Jupyter notebooks - a publishing format for reproducible computational workflows," in *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, F. Loizides and B. Scmidt, Eds. IOS Press, pp. 87–90, https://doi.org/10.3233/978-1-61499-649-1-87.

[15] wwPDB consortium, "Protein Data Bank: the single global archive for 3D macromolecular structure data," *Nucleic Acids Research*, vol. 47, no. D1, pp. D520–D528, 10 2018, https://doi.org/10.1093/nar/gky949.

[16] J. D. Hunter, "Matplotlib: A 2d graphics environment," vol. 9, no. 3, pp. 90–95, https://doi.org/10.1109/MCSE.2007.55.

[17] F. Yabukarski, T. Doukov, D. A. Mokhtari, S. Du, and D. Herschlag, "Evaluating the impact of x-ray damage on conformational heterogeneity in room-temperature (277 k) and cryo-cooled protein crystals," *Acta Crystallographica Section D: Structural Biology*, vol. 78, no. 8, 2022, https://doi.org/10.1107/S2059798322005939.

[18] J. B. Greisman, K. M. Dalton, C. J. Sheehan, M. A. Klureza, I. Kurinov, and D. R. Hekstra, "Native sad phasing at room temperature," *Acta Crystallographica Section D: Structural Biology*, vol. 78, no. 8, pp. 986–996, 2022, https://doi.org/10.1107/s2059798322006799.

[19] J. D. Westbrook, J. Y. Young, C. Shao, Z. Feng, V. Guranovic, C. L. Lawson, B. Vallat, P. D. Adams, J. M. Berrisford, G. Bricogne *et al.*, "Pdbx/mmcif ecosystem: foundational semantic tools for structural biology," *Journal of Molecular Biology*, vol. 434, no. 11, p. 167599, 2022, https://doi.org/10.1016/j.jmb.2022.167599.

[20] E. Schulte, D. Davison, T. Dye, C. Dominik *et al.*, "A multi-language computing environment for literate programming and reproducible research," *Journal of Statistical Software*, vol. 46, no. 3, pp. 1–24, 1 2012, https://doi.org/10.18637/jss.v046.i03. [Online]. Available: http://www.jstatsoft.org/v46/i03

[21] R. M. Horton, "Scripting wizards for chime and rasmol," *Biotechniques*, vol. 26, no. 5, pp. 874–6, 1999, https://doi.org/10.2144/99265ir01.

[22] S. E. Mottarella, M. Rosa, A. Bangura, H. J. Bernstein, and P. A. Craig, "Conscript: Rasmol to pymol script converter," *Biochem Mol Biol Educ*, vol. 38, no. 6, pp. 419–22, 2010, https://doi.org/10.1002/bmb.20450.

[23] B. H. Mooers and M. E. Brown, "Templates for writing pymol scripts," *Protein Science*, vol. 30, no. 1, pp. 262–269, 2021, https://doi.org/10.1002/pro.3997.

[24] B. H. Mooers, "A pymol snippet library for jupyter to boost researcher productivity," *Computing in Science and Engineering*, vol. 23, no. 2, pp. 47–53, 2021, https://doi.org/10.1109/MCSE.2021.3059536.

[25] S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D. S. Seljebotn, and K. Smith, "Cython: The best of both worlds," *Computing in Science & Engineering*, vol. 13, no. 2, pp. 31–39, 2011, https://doi.org/10.1109/mcse.2010.118.

[26] K. Ushey, J. Allaire, and Y. Tang, *reticulate: Interface to 'Python'*, 2023, r package version 1.28. [Online]. Available: https://CRAN.R-project.org/package=reticulate

[27] J. Agirre, M. Atanasova, H. Bagdonas, C. Ballard, A. Baslé, J. Beilsten-Edmands, R. Borges, D. Brown, J. Burgos-Mármol, J. Berrisford *et al.*, "The ccp4 suite: integrative software for macromolecular crystallogra-

phy," *Acta Crystallographica Section D: Structural Biology*, vol. 79, no. 6, pp. 449–461, 2023, https://doi.org/10.1107/S2059798323003595.

[28] S. McNicholas, T. Croll, T. Burnley, C. M. Palmer, S. W. Hoh, H. T. Jenkins, E. Dodson, K. Cowtan, and J. Agirre, "Automating tasks in protein structure determination with the clipper python module," *Protein Science*, vol. 27, no. 1, pp. 207–216, 2018, https://doi.org/10.1002/pro.3299.

[29] K. M. Dalton, J. B. Greisman, and D. R. Hekstra, "A unifying bayesian framework for merging x-ray diffraction data," *Nature Communications*, vol. 13, no. 1, p. 7764, 2022, https://doi.org/10.1038/s41467-022-35280-8.

Bayesian Statistics with Python, No Resampling Necessary

Charles Lindsey[‡*]

Abstract—TensorFlow Probability is a powerful library for statistical analysis in Python. Using TensorFlow Probability's implementation of Bayesian methods, modelers can incorporate prior information and obtain parameter estimates and a quantified degree of belief in the results. Resampling methods like Markov Chain Monte Carlo can also be used to perform Bayesian analysis. As an alternative, we show how to use numerical optimization to estimate model parameters, and then show how numerical differentiation can be used to get a quantified degree of belief. How to perform simulation in Python to corroborate our results is also demonstrated.

Index Terms—Bayesian statistics, resampling, maximum likelihood, numerical differentiation

Introduction

Some machine learning algorithms output only a single number or decision. It can be useful to have a measure of confidence in the output of the algorithm, a quantified degree of belief. Bayesian statistical methods can be used to provide both estimates and confidence for users.

A model with parameters $\boldsymbol{\theta}$ governs the process we are investigating. We begin with a prior belief about the probability distribution of $\boldsymbol{\theta}$, the density $\pi(\boldsymbol{\theta})$.

Then the data we observed gives us a refined belief about the distribution $\boldsymbol{\theta}$. We obtain the posterior density $\pi(\boldsymbol{\theta}|\mathbf{x})$.

We can estimate values of $\boldsymbol{\theta}$ with the posterior mode of $\pi(\boldsymbol{\theta}|\mathbf{x})$, $\widehat{\boldsymbol{\theta}}$.

Then we can estimate the posterior variance of $\boldsymbol{\theta}$, and with some knowledge of $\pi(\boldsymbol{\theta}|\mathbf{x})$ obtain confidence in our estimate $\widehat{\boldsymbol{\theta}}$.

Normal Approximation to the Posterior

We will use numerical optimization to obtain the posterior mode $\widehat{\boldsymbol{\theta}}$, maximizing the posterior $\pi(\boldsymbol{\theta}|\mathbf{x})$.

The posterior is proportional (where the scaling does not depend on $\boldsymbol{\theta}$) to the prior and likelihood (or density of the data).

$$\pi(\boldsymbol{\theta}|\mathbf{x}) \propto L(\boldsymbol{\theta}|\mathbf{x})\pi(\boldsymbol{\theta})$$

As in maximum likelihood, we directly maximize the log-posterior, $\log \pi(\boldsymbol{\theta}|\mathbf{x})$ because it is more numerically stable.

* Corresponding author: charles.lindsey@revionics.com
‡ Revionics, an Aptos Company

Now, as described in section 4.1 of [1] , we can approximate $\ln \pi(\boldsymbol{\theta}|\mathbf{x})$ using a second order Taylor Expansion around $\widehat{\boldsymbol{\theta}}$.

$$\begin{aligned}\log \pi(\boldsymbol{\theta}|\mathbf{x}) \quad &\approx \quad \log \pi(\widehat{\boldsymbol{\theta}}|\mathbf{x}) + (\boldsymbol{\theta}-\widehat{\boldsymbol{\theta}})^T S(\boldsymbol{\theta})|_{\boldsymbol{\theta}=\widehat{\boldsymbol{\theta}}} \\ &\quad + \frac{1}{2}(\boldsymbol{\theta}-\widehat{\boldsymbol{\theta}})^T H(\widehat{\boldsymbol{\theta}})(\boldsymbol{\theta}-\widehat{\boldsymbol{\theta}})\end{aligned}$$

Where $S(\boldsymbol{\theta})$ is the score function

$$S(\boldsymbol{\theta}) = \frac{\delta}{\delta \boldsymbol{\theta}} \log \pi(\boldsymbol{\theta}|\mathbf{x})$$

and $H(\boldsymbol{\theta})$ is the Hessian function.

$$H(\boldsymbol{\theta}) = \frac{\delta}{\delta \boldsymbol{\theta}^T} S(\boldsymbol{\theta})$$

We assume that $\widehat{\boldsymbol{\theta}}$ is in the interior of the parameter space (or support) of $\boldsymbol{\theta}$. Also, $\pi(\boldsymbol{\theta}|\mathbf{x})$ is a continuous function of $\boldsymbol{\theta}$.

Finally the Hessian matrix, $H(\boldsymbol{\theta})$ is negative definite, so $-H(\boldsymbol{\theta})$ is positive definite. This means that we can invert $-H(\boldsymbol{\theta})$ and get a matrix that is a valid covariance.

With these assumptions, as the sample size $n \to \infty$ the quadratic approximation for $\log \pi(\boldsymbol{\theta}|\mathbf{x})$ becomes more accurate. At the posterior mode $\boldsymbol{\theta} = \widehat{\boldsymbol{\theta}}$, $\log \pi(\boldsymbol{\theta}|\mathbf{x})$ is maximized and $0 = S(\boldsymbol{\theta})|_{\boldsymbol{\theta}=\widehat{\boldsymbol{\theta}}}$.

Given this, we can exponentiate the approximation to get

$$\pi(\boldsymbol{\theta}|\mathbf{x}) \approx \pi(\widehat{\boldsymbol{\theta}}|\mathbf{x}) \exp(\frac{1}{2}(\boldsymbol{\theta}-\widehat{\boldsymbol{\theta}})^T H(\widehat{\boldsymbol{\theta}})(\boldsymbol{\theta}-\widehat{\boldsymbol{\theta}}))$$

So for large n, the posterior distribution of $\boldsymbol{\theta}$ is approximately proportional to a multivariate normal density with mean $\widehat{\boldsymbol{\theta}}$ and covariance $-H(\widehat{\boldsymbol{\theta}})^{-1}$.

$$\boldsymbol{\theta}|x \approx_D N(\widehat{\boldsymbol{\theta}}, -H(\widehat{\boldsymbol{\theta}})^{-1})$$

Another caveat for this result is that the prior should be proper, or at least lead to a proper posterior. By proper we mean that the function corresponds to a probability density function. Our asymptotic results are depending on probabilities integrating to 1.

We could get a quantified degree of beief by using resampling methods like Markov chain Monte Carlo (MCMC) [1] directly. We would have to use fewer assumptions. However, resampling can be computationally intensive.

Parameter Constraints and Transformations

Optimization can be easier if the parameters are defined over the entire real line. Parameters that do not follow this rule are plentiful. Variances are only positive. Probabilities are in [0,1].

We can perform optimization over the real line by creating unconstrained parameters from the original parameters of interest. These are continuous functions of the constrained parameters, which may be defined on intervals of the real line.

For example, the unconstrained version of a standard deviation parameter σ is $\psi = \log\sigma$. The parameter ψ is defined over the entire real line.

It will be useful for us to consider the constrained parameters as being functions of the unconstrained parameters. So $\sigma = exp(\psi)$ is our constrained parameter of ψ.

So the posterior mode of the constrained parameters $\boldsymbol{\theta_c}$ is $\widehat{\boldsymbol{\theta}}_{\boldsymbol{c}} = g(\widehat{\boldsymbol{\theta}})$. We will call g the **constraint** function.

Then we can use the delta method [2] on g to get the posterior distribution of the constrained parameters.

A first-order Taylor approximation of $g(\boldsymbol{\theta})$ at $\widehat{\boldsymbol{\theta}}$ yields

$$g(\boldsymbol{\theta}) \approx g(\widehat{\boldsymbol{\theta}}) + \left\{ \frac{\delta}{\delta\widehat{\boldsymbol{\theta}}} g(\widehat{\boldsymbol{\theta}}) \right\} (\boldsymbol{\theta} - \widehat{\boldsymbol{\theta}})$$

Remembering that the posterior of $\boldsymbol{\theta}$ is approximately normal, the rules about linear transformations for multivariate normal random vectors tell us that

$$\begin{aligned} &\boldsymbol{\theta_c}|x = g(\boldsymbol{\theta})|x \approx_D \\ &N\left[g(\widehat{\boldsymbol{\theta}}), \left\{ \frac{\delta}{\delta\widehat{\boldsymbol{\theta}}} g(\widehat{\boldsymbol{\theta}}) \right\}^T \left\{ -H(\widehat{\boldsymbol{\theta}})^{-1} \right\} \left\{ \frac{\delta}{\delta\widehat{\boldsymbol{\theta}}} g(\widehat{\boldsymbol{\theta}}) \right\} \right] \end{aligned}$$

We could use Numpy's **matmul** function to multiply the component matrices together. The **inv** function in the **lingalg** library could be used to invert the Hessian. So referring the to the gradient of g as **dg**, the following python code could be used to compute the constrained covariance.

```
np.matmul(
    np.matmul(dg,
              np.linalg.inv(hessian)),
    np.transpose(dg))
```

This involved a first-order approximation of g. Earlier we used a second order approximation for taking the numeric derivative. Why would we just do a first-order here? Traditionally the delta-method is taught and used as only a first-order method. Usually the functions used in the delta method are not incredibly complex. It is *good enough* to to use the first-order approximation.

Hessian and Delta Approximation

To be able to use the normal approximation, we need $\widehat{\boldsymbol{\theta}}$, $H(\widehat{\boldsymbol{\theta}})^{-1}$, and $\frac{\delta}{\delta\widehat{\boldsymbol{\theta}}} g(\widehat{\boldsymbol{\theta}})$. As mentioned before, we use numerical optimization to get $\widehat{\boldsymbol{\theta}}$. Ideally, we would have analytic expressions for H and the derivatives of g.

This can be accomplished with automatic differentiation [3], which will calculate the derivatives analytically. We can also perform numerical differentiation to get the Hessian and the gradient of the constraint function g. This will be less accurate than an analytic expression, but may be less computationally intensive in large models.

But once you learn how to take one numeric derivative, you can take the numeric derivative of anything. So using numerical differentiation is a very flexible technique that we can easily apply to all the models we would use.

Numerical Differentiation

So numeric derivatives can be very pragmatic, and flexible. How do you compute them? Are they accurate? We use section 5.7 of [4] as a guide.

The derivative of the function f with respect to x is

$$f'(x) = \lim_{h\to 0} \frac{f(x+h) - f(x)}{h}$$

To approximate $f'(x)$ numerically, couldn't we just plugin a small value for h and compute the scaled difference? Yes. And that is basically what happens. We do a little more work to choose h and use a second-order approximation instead of a first-order.

We can see that the scaled difference is a first-order approximation by looking at the Taylor series expansion around x.

Taylor's theorem with remainder gives

$$\begin{aligned} f(x+h) &= f(x) + ((x+h)-x)f'(x) + .5((x+h)-x)^2 f''(\varepsilon) \\ &= f(x) + -hf'(x) + .5h^2 f''(\varepsilon) \end{aligned}$$

where ε is between x and $x+h$.

Now we can rearrange to get

$$\frac{f(x+h) - f(x)}{h} - f'(x) = .5hf''(\varepsilon)$$

The right hand side is the truncation error, ε_t since it's linear in h, the bandwidth we call the this approximation a first order method.

We can do second-order approximations for $f(x+h)$ and $f(x-h)$ and get a more accurate second order method of approximation for $f'(x)$.

$$\begin{aligned} f(x+h) &= f(x) + ((x+h)-x)f'(x) \\ &\quad + \frac{((x+h)-x)^2 f''(x)}{2!} + \frac{((x+h)-x)^3 f'''(\varepsilon_1)}{3!} \\ f(x-h) &= f(x) + ((x-h)-x)f'(x) \\ &\quad + \frac{((x-h)-x)^2 f''(x)}{2!} + \frac{((x-h)-x)^3 f'''(\varepsilon_2)}{3!} \end{aligned}$$

were ε_1 is between x and $x+h$ and ε_2 is between $x-h$ and x.

Then we have

$$\frac{f(x+h) - f(x-h)}{2h} - f'(x) = h^2 \frac{f'''(\varepsilon_1) + f'''(\varepsilon_2)}{12}$$

This is quadratic in h. The first term takes equal input from both sides of x, so we call it a centered derivative.

So we choose a small value of h and plug it into $\frac{f(x+h)-f(x-h)}{2h}$ to approximate $f'(x)$.

Our derivation used a single input function f. The idea applies to partial derivatives of multi-input functions as well. The inputs that you aren't taking the derivative with respect to are treated as fixed parts of the function.

Choosing a Bandwidth

In practice, second order approximation actually involves two sources of error. Roundoff error, ε_r arises from being unable to represent x and h or functions of them with exact binary represetation.

$$\varepsilon_r \approx \varepsilon_f \frac{|\, f(x)\,|}{h}$$

where ε_f is the fractional accuracy with which f is computed. This is generally machine accuracy. If we are using NumPy [5] this would be

$$\varepsilon_f = \text{np.finfo(float).eps}$$

Minimizing the roundoff error and truncation error, we obtain

$$h \sim \varepsilon_f^{1/3} \left(\frac{f}{f'''} \right)^{1/3}$$

where $(f/f''')^{1/3}$ is shorthand for the ratio of $f(x)$ and the sum of $f'''(\varepsilon_1) + f'''(\varepsilon_2)$.

We use shorthand here because because we are not going to approximate f''' (we are already approximating f'), so there is no point in writing it out.

Call this shorthand

$$\left(\frac{f}{f'''} \right)^{1/3} = x_c$$

the curvature scale, or characteristic scale of the function f.

There are several algorithms for choosing an optimal scale. The better the scale chosen, the more accurate the approximation is. A good rule of thumb, which is computationally quick, is to just use the absolute value of x.

$$x_c = \mid x \mid$$

Then we would use

$$h = \varepsilon_f^{1/3} \mid x \mid$$

But what if x is 0? This is simple to handle, we just add $\varepsilon_f^{1/3}$ to $x_c = \mid x \mid$

$$h = \varepsilon_f^{1/3} (\mid x \mid + \varepsilon_f^{1/3})$$

Now, [4] also suggests performing a final sequence of assignment operations that ensures x and $x+h$ differ by an exactly representable number. You assign $x+h$ to a temporary variable $temp$. Then h is assigned the value of $temp - h$.

In Python, the code would simply be

```
temp = x + h
h = temp - x
```

Estimating Confidence Intervals after Optimization

With the posterior mode, variance, and normal approximation to the posterior. It is simple to create confidence (credible) intervals for the parameters.

Let's talk a little bit about what these intervals are. For the parameter γ we want a $(1-\alpha)$ interval (u,l) (defined on the observed data generated by a realization of γ) to be defined such that

$$\Pr(\gamma \in (u,l)) = 1 - \alpha$$

The frequentist confidence interval does not meet this criteria. γ is just one fixed value, so it is either in the interval, or it isn't! The probability is 0 or 1. A credible interval (Bayesian confidence interval) can meet this criteria.

Suppose that we are able to use the normal approximation for $\gamma|\mathbf{x}$

$$\gamma|\mathbf{x} \approx_D N(\hat{\gamma}, \hat{\sigma}_\gamma^2)$$

Then we have

$$\begin{aligned} 1-\alpha &= \Pr(l \le \gamma \le u|\mathbf{x}) \\ &= \Pr(l - \hat{\gamma} \le \gamma - \hat{\gamma} \le u - \hat{\gamma}|\mathbf{x}) \\ &= \Pr\left(\frac{l-\hat{\gamma}}{\hat{\sigma}_\gamma} \le \frac{\gamma - \hat{\gamma}}{\hat{\sigma}_\gamma} \le \frac{u-\hat{\gamma}}{\hat{\sigma}_\gamma} |\mathbf{x} \right) \end{aligned}$$

Now, $(\gamma - \hat{\gamma})/\hat{\sigma}_\gamma^2$ is $N(0,1)$, standard normal. So we can use the standard normal quantiles in solving for l and u.

The upper $\alpha/2$ quantile of the standard normal distribution, $z_{\alpha/2}$ satisfies

$$\Pr(Z \ge z_{\alpha/2}) = \alpha/2$$

for standard normal Z.

Noting that the standard normal is symmetric, if we can find l and u to satisfy

$$\begin{aligned} \frac{l-\hat{\gamma}}{\hat{\sigma}_\gamma} &= -z_{\alpha/2} \\ \frac{u-\hat{\gamma}}{\hat{\sigma}_\gamma} &= z_{\alpha/2} \end{aligned}$$

then we have a valid Bayesian confidence interval.

Simple calculation shows that the solutions are

$$\begin{aligned} l &= -z_{\alpha/2}\hat{\sigma}_\gamma + \hat{\gamma} \\ u &= z_{\alpha/2}\hat{\sigma}_\gamma + \hat{\gamma} \end{aligned}$$

The $z_{\alpha/2}$ quantile can be easily generated using **scipy.stats** from SciPy [6]. We would use the **norm.ppf** function.

In Python, we would have

```
z_alpha_2 = scipy.stats.norm.ppf(1-alpha/2)
l = -z_alpha_2*se_gamma_hat + gamma_hat
u = z_alpha_2*nsd_gamma_hat + gamma_hat
```

We can also adjust the intervals for inference on many parameters by using Bonferroni correction [7].

Now we know how to estimate the posterior mode. We also know how to estimate the posterior variance after computing the posterior mode. And we have seen how confidence intervals are made based on this posterior variance, mode, and the normal approximation to the posterior. Let's discuss some tools that will enable us to perform these operations.

TensorFlow Probability

Now we will introduce TensorFlow Probability, a Python library that we can use to perform the methods we have been discussing. TensorFlow Probability is library built using TensorFlow, a leading software library for machine learning and artificial intelligence [8].

TensorFlow Probability is a probabilistic programming language. This lets us build powerful models in a modular way and estimate them automatically. At the heart of TensorFlow Probability is the **Distribution** class. In theory, a probability distribution is the set of rules that govern the likelihood of how a random variable (vector, or even general tensor) takes its values.

In TensorFlow Probability, distribution rules for scalars and vectors are parametrized, and these are expanded for higher dimensions as independent samples. A distribution object corresponds to a random variable or vector. The parts of a Bayesian model can be represented using different distribution objects for the parameters and observed data.

Example Distribution

As an example, let's examine a linear regression with a χ^2 prior for the intercept a and a normal prior for the slope β. Our observed outcome variable is y with a normal distribution and the predictor is x.

$$y_i \sim \text{Normal}(x_i\beta + \alpha, 1)$$

We can store the distribution objects in a dictionary for clear organization. The prior distribution of β is Normal with mean 1

and variance 1, $N(1,1)$. We use the **Normal** distribution subclass to encode its information in our dictionary.

```
tfd = tfp.distributions
dist_dict = {}
dist_dict['beta'] = tfd.Normal(1,1)
```

The β parameter can range over the real line, but the intercept, α should be nonnegative. The **Chi2** distribution sublcass has support on only the nonegative reals. However, if we are performing optimization on the α parameter, we may take a step where it became negative. We can avoid any complications like this if we use a **TransformedDistribution**. Transformed distributions can be used together with a **Bijector** object that represents the transforming function.

For α, we will model an unconstrained parameter, $\alpha^u = \log \alpha$. The natural logarithm can take values over the real line.

```
tfb = tfp.bijectors
dist_dict['unconstrained_alpha'] = \
tfd.TransformedDistribution(tfd.Chi2(4),tfb.Log())
```

We can use the **sample** method on the distribution objects we created to see random realizations. Before we do that we should set the seed, so that we can replicate our work.

```
tf.random.set_seed(132)
sample_ex=dist_dict['unconstrained_alpha'].sample(10)
sample_ex
```

```
<tf.Tensor: shape=(10,), dtype=float32, numpy=
array([ 2.050956  , 0.56120026,  1.8559402,
       -0.05669071, ... ], dtype=float32)>
```

We see that the results are stored in a **tf.Tensor** object. This has an easy interface with NumPy, as you can see by the **numpy** component. We see that the unconstrained α, α^u takes positive and negative values.

We can evaluate the density, or it's natural logarithm using class methods as well. Here is the log density for the sample we just drew.

```
dist_dict['unconstrained_alpha'].log_prob(sample_ex)
```

```
<tf.Tensor: shape=(10,), dtype=float32, numpy=
array([-1.1720479 , -1.1402813 , -0.8732692 ,
       -1.9721189 , ...], dtype=float32)>
```

Now we can get α from α^u by using a callable and the **Deterministic** distribution.

```
dist_dict['alpha'] = \
    lambda unconstrained_alpha: \
        tfd.Deterministic(\
            loc= tfb.Log().inverse( \
                unconstrained_alpha))
```

Now we've added all of the parameters to **dist_dict**. We just need to handle the observed variables y and x. In this example x is **exogenous**, which means it can be treated as fixed and nonrandom in estimating α and β in the model for y. y is **endogenous**, which means it is a response variable in the model, the outcome we are trying to estimate.

We will define x separately from our dictionary of distributions. For the example we have to generate values of x, but once this is done we will treat it as fixed and exogenous

The observed variable x will have a standard normal distribution. We will start by giving it a sample size of 100.

```
n = 100
x_dist = tfd.Normal(tf.zeros(n),1)
x = x_dist.sample()
```

The distribution of y, which would give us the likelihood, can be formulated using a callable function of the parameters and the fixed value of x we just obtained.

```
dist_dict['y'] = \
    lambda alpha, beta: \
        tfd.Normal(loc = alpha + beta*x,scale=1)
```

With a dictionary of distributions and callables indicating their dependencies, we can work with the joint density. This will correspond to the posterior distribution of the model, augmenting the priors with the likelihood.

The **JointDistributionNamed** class takes a dictionary as input and behaves similarly to a regular univariate distribution object. We can take samples, which are returned as dictionaries keyed by the parameter and observed variable names. We can also compute log probabilities, which gives us the posterior density.

```
posterior = tfd.JointDistributionNamed(dist_dict)
```

Now we have a feel for how TensorFlow Probability can be used to store a Bayesian model. We have what we need to start performing optimization and variance estimation.

Maximum A Posteriori (MAP) with SciPy

We can use SciPy's implementation of the Limited memory Broyden–Fletcher–Goldfarb–Shanno (L-BFGS) [9] algorithm to estimate the posterior mode. This is a Quasi-Newton optimization method that does not need to store the entire Hessian matrix during the algorithm, so it can be very fast. If the Hessian was fully stored we could just use it directly in variance estimation, but it would be slower. We do to take advantage of automatic differentiation to calculate the score function, the first derivative of the posterior. TensorFlow Probability provides this through the **value_and_gradient** function of its **math** library.

We will use **minimize** from the **optimize** SciPy library, which operates on a loss function that takes a vector of parameters as input. We will optimize on **unconstrained_alpha** and **beta**, the unconstrained space parameters of the model. In the joint distribution representation, they are separate tensors. But in optimization, we will need to evaluate a single tensor.

We will use the first utility function from the **bayes_mapvar** library, which will be available with this paper, to accomplish this. The **par_vec_from_dict** function unpacks the tensors in a dictionary into a vector.

Within our loss function, we must move the vector of input parameters back to a dictionary of tensors to be evaluated by TensorFlow probability. The **par_dict_from_vec** function moves the unconstrained parameters back into a dictionary, and the constrained parameters are generated by the **get_constrained** function. Then the posterior density is evaluated by augmenting this dictionary of constrained parameters with the observed endogenoous variables. The get_constrained function is also used to get the final posterior model estimates from the SciPy optimization.

Variance Estimation with SciPy

Once the posterior mode is estimated we can estimate the variance. The first step is calculating the bandwidths. The **get_bandwidths** function handles this.

```
def get_bandwidths(unconstrained_par_vec):
    abspars = abs(unconstrained_par_vec)
    epsdouble = np.finfo(float).eps
```

Statistic	Mean	S.D.
α_{MAP} mean	4.141	2.475
α_{MCMC} mean	3.989	2.765
α_{MAP} S.E.	0.037	0.004
α_{MCMC} S.E.	0.041	0.001
α A.D. Reject	0.042	0.201
β_{MAP} mode	1.013	0.504
β_{MCMC} mean	1.022	1.003
β_{MAP} S.E.	0.029	0.001
β_{MCMC} S.E.	0.041	0.002
β A.D. Reject	0.045	0.207

TABLE 1
Simulation Results, $n_{pre} = 1000$, $n_{post} = 600$, $n_{MCMC} = 500$.

```
epsdouble = epsdouble**(1 / 3)
scale = epsdouble * (abspars + epsdouble)
scaleparmstable = scale + abspars
return scaleparmstable - abspars
```

With the bandwidths calculated, we step through the parameters and create the Hessian and Delta matrices that we need for variance estimation. The **get_hessian_delta_variance** function use numeric differentation to calculate the Hessian, based on numeric derivaties of the automatic derivatives computed by TensorFlow probability for the score function. The Delta matrix is calculated using numeric differentation of the constrained parameter functions.

Simulation

We evaluated our methodology with a simulation based on the α and β parameter setting discussed earlier. This was an investigation into how well we estimated the posterior mode, variance, and distribution using the methods of TensorFlow Probability, SciPy, and bayes_mapvar.

To evaluate the posterior distributions of the parameters we used the MCMC capabilities of TensorFlow Probability. Particulary the the No-U-Turn Sampler [10]. We were careful to thin the sample based on effective sample size so that autocorrelation would not be a problem. This was accomplished using TensorFlow Probability's **effective_sample_size** function from its **mcmc** library.

We drew $n_{pre} = 1000$ observations from the unconstrained prior parameter distribution for α_i and β_i. For each of these prior draws, we drew a posterior sample of $\mathbf{y}_i$ and $\mathbf{x}_i$. $\mathbf{y}_i$ and $\mathbf{x}_i$ were $n_{post} = 600$ samples based on each `\alpha_i` and β_i. The posterior mode and variance were estimated, and $n_{MCMC} = 500$ posterior draws from MCMC were made. The mean was used in the MCMC draws since it sould coincide with the mode if our assumptions are correct.

To check the distributional results, we used the Anderson-Darling test [11]. This is given by **anderson** in **scipy.stats**. We stored a record of whether the test rejects normality at the .05 significance level for each of the n_{pre} draws. This test actually checks the mean and variance assumptions as well, since it compares to a standard normal and we are standardizing based on the MAP and **get_hessian_delta_variance** estimates.

The results of the simulation are shown in 1.We use Standard Error (S.E.) to refer to the 1000 estimates of posterior standard deviations from **get_hessian_delta_variance** and the MCMC sample standard deviations. The Standard Deviation (S.D.) column represents the statistics calculated over the 1000 estimates. The standard errors are not far from each other, and neither are the modes and means. The rejection rates for the Anderson Darling test are not far from .05 either.

Statistics	Lower	Upper
α AD Reject	0.030	0.056
β A.D. Reject	0.033	0.060

TABLE 2
A.D. Confidence Intervals, $n_{pre} = 1000$, $n_{post} = 600$, $n_{MCMC} = 500$.

We can perform a hypothesis test of whether the rejection rate is .05 by checking whether .05 is in the confidence interval for the proportion. We will use the **proportion_confint** function from **statsmodels** [12]. In 2, we see that .05 is comfortably within intervals for both parameters. Our simulation successfully corroborated our assumptions about the model and the consistency of our method for estimating the posterior mode, variance, and distribution.

Conclusion

We have explored how Bayesian analysis can be performed without resampling and still obtain full inference. With adequate amounts of the data, the posterior mode can be estimated with numeric optimization and the posterior variance can be estimated with numeric or automatic differentation. The asymptotic normality of the posterior distribution enables simple calculation of posterior probabilities and confidence (credible) intervals as well.

Bayesian methods let us use data from past experience, subject matter expertise, and different levels of certainty to solve data sparsity problems and provide a probabilistic basis for inference. Retail Price Optimization benefits from historical data and different granularities of information. Other fields may also take advantage of access to large amounts of data and be able to use these approximation techniques. These techniques and the tools implementing them can be used by practicioners to make their analysis more efficient and less intimidating.

References

[1] A. Gelman, J. Carlin, H. Stern, D. Dunson, A. Vehtari, and D. Rubin, *Bayesian Data Analysis, Third Edition*, ser. Chapman & Hall/CRC Texts in Statistical Science. Taylor & Francis, 2013. [Online]. Available: https://books.google.com/books?id=ZXL6AQAAQBAJ

[2] G. W. Oehlert, "A note on the delta method," *The American Statistician*, vol. 46, no. 1, pp. 27–29, 1992, https://doi.org/10.1080/00031305.1992.10475842. [Online]. Available: https://www.tandfonline.com/doi/abs/10.1080/00031305.1992.10475842

[3] A. G. Baydin, B. A. Pearlmutter, A. A. Radul, and J. M. Siskind, "Automatic differentiation in machine learning: a survey," 2018.

[4] W. Press and S. Teukolsky, *Numerical Recipes 3rd Edition: The Art of Scientific Computing*, ser. Numerical Recipes: The Art of Scientific Computing. Cambridge University Press, 2007. [Online]. Available: https://books.google.com/books?id=1aAOdzK3FegC

[5] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant, "Array programming with NumPy," vol. 585, no. 7825, pp. 357–362, https://doi.org/10.1038/s41586-020-2649-2. [Online]. Available: https://doi.org/10.1038/s41586-020-2649-2

[6] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. J. van der Walt, M. Brett, J. Wilson, K. J. Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. J. Carey, İ. Polat, Y. Feng, E. W. Moore, J. VanderPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, P. van Mulbregt, and SciPy 1.0 Contributors, "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python," *Nature Methods*, vol. 17, pp. 261–272, 2020, https://doi.org/10.1038/s41592-019-0686-2.

[7] C. Bonferroni, "Teoria statistica delle classi e calcolo delle probabilita," *Pubblicazioni del R Istituto Superiore di Scienze Economiche e Commericiali di Firenze*, vol. 8, pp. 3–62, 1936.

[8] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: https://www.tensorflow.org/

[9] R. Fletcher, *Practical Methods of Optimization*, 2nd ed. New York, NY, USA: John Wiley & Sons, 1987, https://doi.org/10.1002/9781118723203.

[10] M. D. Hoffman and A. Gelman, "The no-u-turn sampler: Adaptively setting path lengths in hamiltonian monte carlo," 2011.

[11] M. A. Stephens, "Edf statistics for goodness of fit and some comparisons," *Journal of the American Statistical Association*, vol. 69, no. 347, pp. 730–737, Sep. 1974, https://doi.org/10.2307/2286009.

[12] S. Seabold and J. Perktold, "statsmodels: Econometric and statistical modeling with python," in *9th Python in Science Conference*, 2010, https://doi.org/10.25080/majora-92bf1922-011.

Using Numba for GPU acceleration of Neutron Beamline Digital Twins

Coleman J. Kendrick‡*, Jiao Y. Y. Lin‡, Garrett E. Granroth‡

Abstract—Digital twins of neutron instruments using Monte Carlo ray tracing have proven to be useful in neutron data analysis and verifying instrument and sample designs. However, these simulations can become quite complex and computationally demanding with tens of billions of neutrons. In this paper, we present a GPU accelerated version of MCViNE using Python and Numba to balance user extensibility with performance. Numba is an open-source just-in-time (JIT) compiler for Python using LLVM to generate efficient machine code for CPUs and GPUs with NVIDIA CUDA. The JIT nature of Numba allowed complex instrument kernels to be generated easily. Initial simulations have shown a speedup between 200-1000x over the original CPU implementation. The performance gain with Numba enables more sophisticated data analysis and impacts neutron scattering science and instrument design.

Index Terms—Monte Carlo, numba, digital twin, Python, neutron, GPU, ray tracing, CUDA

Introduction

MCViNE [1], [2] is a software package for creating digital twins of neutron scattering experiments using a Monte Carlo ray-tracing approach. In this method, randomly generated probability packets (representing neutrons) are propagated through a series of components. Each component changes the probability packets according to the physics of the component. As an example of a component, consider a neutron mirror with less than perfect reflectivity. The interaction between the probability packet and the mirror would cause the velocity component perpendicular to the mirror to reverse and would reduce the probability of the packet to take into account that there is a finite probability that the neutron would not be reflected. The physics of each MCViNE component is documented in the code. An extensive description of components for a similar package, McStas is provided at [3]. There is no correlation between packets, so the system is embarrassingly parallel. These simulations are useful in performing advanced neutron data analysis [4], [5], [6], [7], [8], [9] as well as in designing novel neutron instruments [10], [11], [12], [13], [14], [15], [16], [17], [18], [19], [20] and sample environments [21], [22]. Specifically, it has been used in the initial designs for instruments in the Second Target Station at the Spallation Neutron Source (SNS) [23] at Oak Ridge National Laboratory. Currently, MCViNE only runs on CPUs which is a bottleneck in large simulations with tens of billions of neutrons, and in modelling complex multiple scattering, with some simulations taking months to complete. Due to the massively parallel nature of Monte Carlo methods, bringing GPU acceleration to these simulations would offer superior performance and scalability. MCViNE was originally implemented in C++ and parallelized using MPI, with bindings to Python for user interaction. However, extensibility for the end user can be very difficult.

To further improve performance and to create an easily extensible code base, Python and Numba [24] were chosen to create a new package of MCViNE providing GPU acceleration, `mcvine.acc` [25]. Numba is an open-source JIT (just-in-time) compiler for Python using LLVM to generate efficient machine code and supports GPUs using NVIDIA CUDA. Numba is designed for scientific computing and can support NumPy arrays and functions. Currently, we are only using Numba for its GPU capabilities as the original version of MCViNE is used to run on CPUs. This accelerated MCViNE package is compatible with existing MCViNE scripts, and using a mixture of CPU and GPU components is supported.

This paper will first describe how MCViNE works at a high-level, how components and instruments are created using Numba to generate CUDA kernels, and how Numba is also used to generate kernels for complex sample geometries and scatterers. Next, we will compare performance of the CPU version of MCViNE to the Numba GPU version. After that we will describe how the MCViNE GPU acceleration will be used in the larger context of a workflow for data analysis. Finally, we will discuss our experience using Numba for this application.

Method

MCViNE Overview

MCViNE simulations are run from a script that defines an instrument, where an instrument is composed of multiple components. A simple 4 component instrument is shown in Figure 1. Full instruments may have several hundred components. Each instrument script is run with a specified number of probability packets, where each packet has several state variables: position, velocity, spin, probability, and time.

At a high-level, a component takes a neutron as input and performs some action on the neutron. Components can be attached to the instrument at a specified position and orientation. Some of the main component types are sources, guides, monitors, samples,

* Corresponding author: kendrickcj@ornl.gov
‡ Oak Ridge National Laboratory

and detectors. In a full instrument simulation, neutrons are generated from a source component and are propagated through each component in the instrument. Sample components are a special type of component with additional input files to specify geometry and material composition.

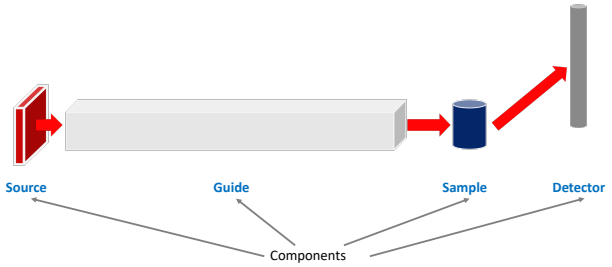

Fig. 1: *Example instrument with four components: a source, guide, sample, and detector.*

Component Hierarchy

One of the major benefits of using Python for this application is the ease of utilizing an object-oriented design and polymorphism. Each component inherits from a base `AbstractComponent` class. This `AbstractComponent` class requires a "propagate" method to be defined which takes in a neutron for the first parameter. The "propagate" method is responsible for defining the action of the component and is decorated with `@cudajit` to indicate that it is a CUDA kernel. An example of creating a simple component can be seen below. Additional parameters can be specified, but must also be defined in the `propagate_params` list. There are several major types of components that have their own subclasses: `SampleBase`, `SourceBase`, and `MonitorBase`.

```
from mcvine.acc.ComponentBase import ComponentBase
class Arm(ComponentBase):

    def __init__(self, name, **kwargs):
        self.name = name
        self.propagate_params = ()

        # Aim a neutron at this arm to
        # cause JIT compilation.
        import mcni
        neutrons = mcni.neutron_buffer(1)
        neutrons[0] = \
            mcni.neutron(r=(0, 0, -1),
                         v=(0, 0, 1),
                         prob=1, time=0)
        self.process(neutrons)

    @cuda.jit(void(NB_FLOAT[:]),
              device=True)
    def propagate(in_neutron):
        pass
```

CUDA Kernel Generation for an Instrument

In order to run a simulation from an input instrument script, `mcvine.acc` first generates a GPU kernel from the instrument specification. The input script will be parsed and then "compiled" to generate a Python script representing an instrument kernel. This compiled version is then imported and executed to run the simulation. An example of a simple MCViNE instrument script containing a source, guide, and monitor can be seen below.

```
def instrument():
    instrument = mcvine.instrument()
    source = Source_simple(
        'src',
        radius = 0., width = 0.03,
        height = 0.03, dist = 1.,
        xw = 0.035, yh = 0.035
    )
    instrument.append(source,
                      position=(0,0,0.))

    guide1 = Guide(
        'guide',
        w1=0.035, h1=0.035,
        w2=0.035, h2=0.035,
        l=10
    )
    instrument.append(guide1,
                      position=(0,0,1.))

    mon = PosDiv_monitor(
        'mon', xwidth=0.08, yheight=0.08,
        maxdiv=2.,
        npos=250, ndiv=251,
    )
    instrument.append(mon,
                      position=(0,0,12.))
    return instrument
```

When `mcvine.acc` is called on the above instrument script, a new Python file is generated which contains a Numba CUDA kernel for the entire instrument. Each of the instrument components' `process` kernels are collected and inserted to form a generic kernel in this process. This generated kernel effectively models a neutron travelling through the entire instrument.

Depending on the kernel launch configuration, each GPU thread might be responsible for more than one neutron. An example of a compiled instrument script can be seen in the code listing below. As seen in the script, a CUDA kernel is defined using Numba. Inside the kernel, each GPU thread will loop over the number of neutrons it is processing. Each `propagate` function has a number appended to it. These propagate functions correspond to the Component's propagate method. For this case, `propagate0` matches the Source component propagate kernel, `propagate1` matches the Guide component propagate kernel, and so on. The `applyTransformation` function is inserted in-between instrument components and is responsible for translating the position/velocity of a neutron in the current component's coordinate system to that of the next component by applying a transformation matrix.

```
@cuda.jit
def process_kernel_no_buffer(
    rng_states, N, n_neutrons_per_thread,
    args
):
    args0, args1, args2, offsets, rotmats, neutron_counter = args
    thread_index = cuda.grid(1)
    start_index = thread_index*n_neutrons_per_thread
    end_index = min(start_index+n_neutrons_per_thread, N)
    neutron = cuda.local.array(shape=10, dtype=NB_FLOAT)
    r = cuda.local.array(3, dtype=NB_FLOAT)
    v = cuda.local.array(3, dtype=NB_FLOAT)
    for neutron_index in range(start_index, end_index):
        cuda.atomic.add(neutron_counter, 0, 1)
        propagate0(thread_index, rng_states,  neutron, *args0)
        applyTransformation(neutron[:3], neutron[3:6],
                            rotmats[0], offsets[0], r, v)
        propagate1(neutron, *args1)
        applyTransformation(neutron[:3], neutron[3:6],
                            rotmats[1], offsets[1], r, v)
        propagate2(neutron, *args2)

from mcvine.acc.components.sources.SourceBase import SourceBase
class _Base(SourceBase): # has to be named Base in definition
    def __init__(self, instrument):
        offsets, rotmats = calcTransformations(instrument)
        self.neutron_counter = neutron_counter = np.zeros(1, dtype=int)
        self.propagate_params = tuple(
            c.propagate_params for c in instrument.components)
        self.propagate_params += (offsets, rotmats, neutron_counter)
```

```
        return
    def propagate(self): return
InstrumentWrapper = _Base
InstrumentWrapper.process_kernel_no_buffer = process_kernel_no_buffer

def run(ncount, ntotalthreads=None, threads_per_block=None, **kwds):
    instrument = loadInstrument(script, **kwds)
    iw = InstrumentWrapper(instrument)
    iw.process_no_buffer(ncount, ntotalthreads=ntotalthreads,
                         threads_per_block=threads_per_block)
    processed = iw.neutron_counter[0]
    saveMonitorOutputs(instrument, scale_factor=1.0/ncount)
```

Sample Kernels and CSG

One unique feature of MCViNE is its sample component, which allows for simulation of complex sample/sample environment and detector systems with flexible, sophisticated geometry and scattering physics. This feature has enabled simulations that produce virtual experiment data that closely resemble real experimental data and make MCViNE a useful tool for both instrument design and advanced data analysis.

A Sample is made up of a Sample Assembly which has `Shape` and `Phase` tags. The `Shape` tag contains the geometry specification. The `Phase` tag, together with additional XML file(s) named `{name}-scatterer.xml`, contain the scattering physics specification where `{name}` is the name of a scatterer included in the sample assembly. A simple example for an aluminum (Al) sphere is shown below. Such an Al sphere is an idealized sample, where the sphere matches the scattering condition and Al is the material most transparent to neutron and is used for mounts and sample containers. Therefore it is well studied and well understood.

```
<?xml version="1.0"?>

<!DOCTYPE SampleAssembly>

<SampleAssembly name="isotropic_sphere">
  <PowderSample name="sample" type="sample">
    <Shape>
      <sphere radius="1*cm" />
    </Shape>
    <Phase type="crystal">
      <ChemicalFormula>Al</ChemicalFormula>
      <xyzfile>Al.xyz</xyzfile>
    </Phase>
  </PowderSample>
  <LocalGeometer registry-coordinate-system="InstrumentScientist">
    <Register name="sample" position="(0,0,0)" orientation="(0,0,0)"/>
  </LocalGeometer>
</SampleAssembly>
```

Simple shapes can be created easily, but more complex shapes can be created too. To create the sample geometry, MCViNE uses Constructive Solid Geometry (CSG). CSG can create complex geometries from operations such as intersection and union, on primitive shapes, such as cubes, spheres, and cylinders. For example, Figure 2 shows how CSG is used to create an annular sample can from cylindrical primitives. This can is similar to those used in experiment design [26]. First a ring for the outside of the can is made from two cylinders by subtraction, then a ring for the inside of the can is also made from two cylinders. These two rings are then combined in union with another cylinder that is the bottom of the can.

Ray tracing routines are implemented as CUDA kernels for these primitive shapes and operations. To support complex geometries that might involve many operations and shapes, the visitor pattern is used which constructs a single CUDA kernel to handle the ray intersection of that shape. This highlights one of the major advantages of using Python and Numba, as the ability to generate a CUDA kernel dynamically at run-time would be much more difficult to implement in other languages.

The specification of the scattering physics of a particular neutron scatterer is described in a dedicated "scatterer" XML file,

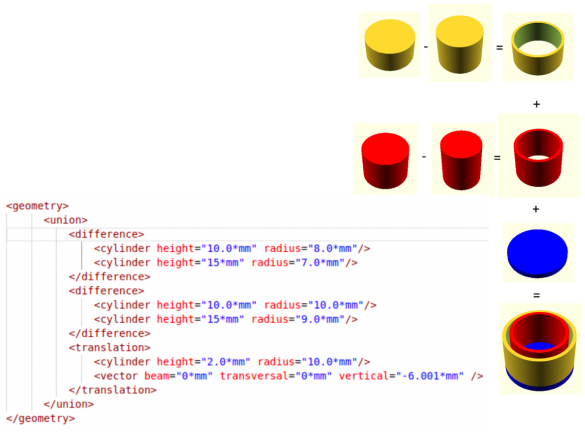

Fig. 2: *An example of an annular sample can created using CSG (right) and how the primitives and operations are specified in XML as input to MCViNE.*

where one or more sample kernels can be specified. An example of a scatterer XML file for specifying scattering physics can be seen below.

```
<?xml version="1.0"?>

<!DOCTYPE scatterer>

<!-- weights: absorption, scattering, transmission -->
<homogeneous_scatterer mcweights="0, 1, 0">
  <IsotropicKernel absorption_coefficient="10./m" scattering_coefficient="10./m">
  </IsotropicKernel>
</homogeneous_scatterer>
```

Note the format is extensible enough to allow a composite scatterer with multiple scatters, though at the time of writing this paper the Numba version of this functionality is still under development.

Results

Two types of comparisons were performed to show the usefulness of `mcvine.acc`. First, simulations comparing the CPU and GPU shows a significant performance gain by using a GPU (Figures 3 – 5). Second, simulations from a more complete instrument solution showing equivalent outcomes from a CPU and GPU simulation were performed (Figure 6 and 7).

For the first study, we focus on the performance gain achieved by the GPU accelerated version of MCViNE. We used a simple instrument consisting of a source, sample, and detector to focus on the sample assembly component. We performed tests with two different samples: a simple spherical sample with an isotropic scattering kernel, and a second with a more complex Uranium Nitride (UN) sample. The UN sample was chosen as it has been experimentally studied and is well modeled by single and multiple scattering of a Quantum Harmonic oscillator model [27], [9]. The UN sample is treated as a 1 cm polycrystalline cube to match the experimental configuration [27]. The UN structure is the same as rock salt structure with the light N atoms located between the much heavier U atoms. The N vibrates as a harmonic oscillator which provides equally spaced lines in energy transfer E. The lines are modeled in MCViNE with a sample scattering kernel containing a composite of $E(Q)$ kernels with constant E values of a 50 meV spacing from 0 to 350 meV.

Figure 3 shows the performance of the isotropic sphere sample for the CPU version of MCViNE with one and 16 cores (blue and

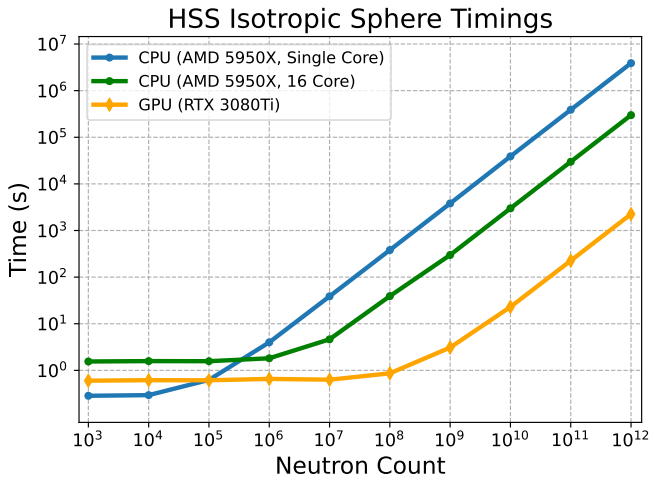

Fig. 3: *Time versus neutron counts for a single core CPU (blue line), multi-core CPU (green line), and GPU (orange line) for a simple instrument using a homogeneous single scatterer (HSS) with a aluminum sphere sample.*

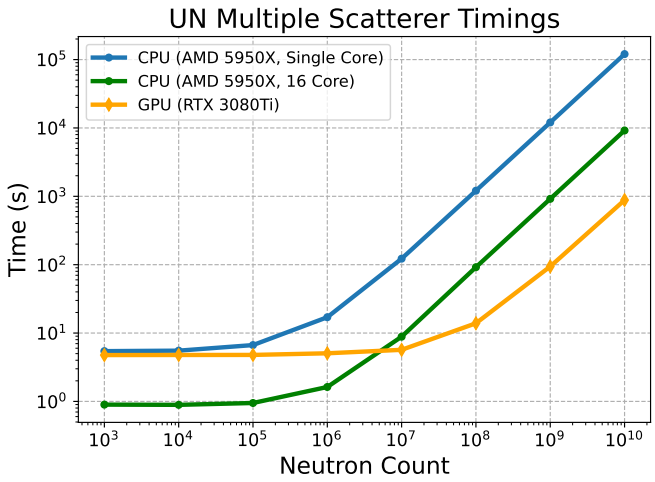

Fig. 5: *Time versus neutron counts for a single core CPU (blue line), multi-core CPU (green line), and GPU (orange line) for the UN instrument with multiple scattering.*

green curves) compared to the GPU version of MCViNE (orange curve). At 10^{12} neutrons, `mcvine.acc` achieves a speedup of 1725x over a single core and 133x over 16 cores.

Two versions of this simulation were run for the UN sample: the first with only single scattering events (Figure 4), and the second with single and multiple scattering events (Figure 5). Single scattering was implemented first to verify the overall workflow and kernel generation of `mcvine.acc`. Multiple scattering was then added to fully capture the realistic scattering physics. Multiple scattering is much more computationally intensive since each neutron can scatter more than once.

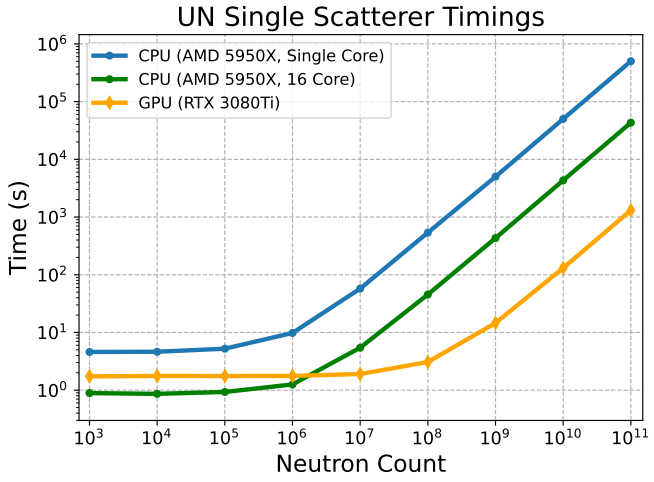

Fig. 4: *Time versus neutron counts for a single core CPU (blue line), multi-core CPU (green line), and GPU (orange line) and GPU (orange line) for the UN instrument with single scattering.*

For the UN single scattering case, Figure 4 shows that for 10^{11} neutrons, the GPU version obtained a speedup of 383x over a single core, and 33x over 16 cores. For the UN multiple scattering case, Figure 5 shows that for 10^{10} neutrons, the GPU version obtained a speedup of 137x over a single core, and 10x over 16 cores. Comparing the speedup achieved for the simple isotropic sphere to the UN with single and multiple scattering shows the additional complexity required for the UN sample. A speedup of 10x over 16-core CPU for the UN multiple scattering case is still significant as some simulations can take on the order of days to months to complete.

Simulation Type	Speedup over 1 Core	Speedup over 16 Cores
HSS Isotropic Sphere	1725	133
UN Single Scatterer	383	33
UN Multiple Scatterer	137	10

TABLE 1: *Speedup achieved by `mcvine.acc` over the CPU with one core and 16 cores for each type of simulation.*

While the speedup over the CPU version of MCViNE is significant, further optimization is possible. Currently, each GPU thread executes a single large kernel that models the instrument. For large instruments that contain many components, the instrument kernel can use too many registers which limits device occupancy. Additionally, a lot of components involve many conditional statements which do not perform well on the GPU. This can be seen by comparing the performance of complex sample components, such as the UN sample, to the simple isotropic sphere.

Next we run a more complete test with the Uranium Nitride sample to verify the same result between the CPU and GPU. A study on UN was the first time the multiple-scattering as well as the multiple-scattering physics in the CPU version of MCViNE was used to explain experimental results [9]. Specifically, one of the puzzles from the measured data was that the equally spaced lines extend over all Q. It was determined that this was due to multiple scattering. To more conclusively check this, a CPU simulation using MCViNE was performed [9]. At the time this CPU simulation was run, it took days to do such calculations. Therefore, this is a good test case to check the speed increase gained from using GPUs with `mcvine.acc`.

In this case, the incident beamline simulation (the simulation up to the sample containing the SNS source, guide choppers and slits) for the Wide Angular-Range Chopper Spectrometer (ARCS) instrument [28], [29] was run using McStas [30], [31] inside a workflow tool [32]. To configure the incident beam simulation in this workflow a user simply provides an existing experimental data

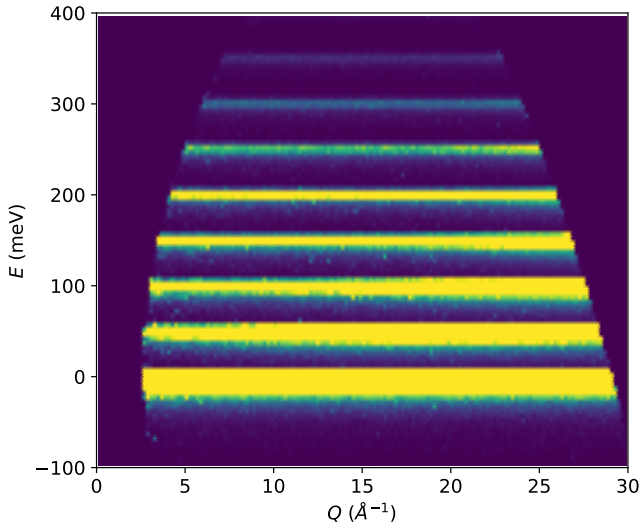

Fig. 6: *The results from the UN simulation on ARCS with multiple scattering turned on. Color indicates the scattering intensity where brighter regions represent higher intensity. Note that the equally spaced lines are visible even at low Q.*

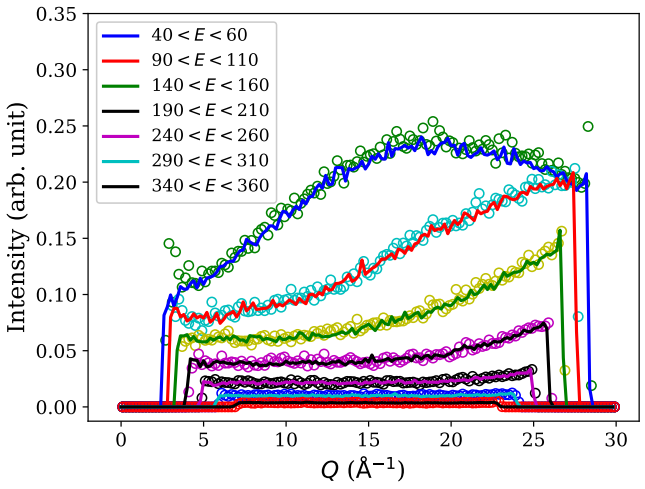

Fig. 7: *Constant-E cuts around the individual energy levels in the $I(Q,E)$ displayed in Figure 6. Note the functional dependence of each level and the background increase are consistent with the physics and the multiple scattering. The comparisons between the CPU and GPU calculations are shown with the solid line and circles respectively.*

file that provides the necessary parameters. Specifically, an ARCS data file with an $E_i = 500$ meV was fed into the workflow which then generates an mcpl [33] file [34] for use in MCViNE. The rest of the virtual instrument uses `mcvine.acc` to leverage the GPU acceleration. It consists of a source component that reads the mcpl file to generate neutrons for the neutron source component, a sample assembly component, and a powder $S(Q,E)$ monitor component for direct-geometry inelastic neutron spectrometers. The results are shown in Figure 6 and 7.

First, note the equally spaced lines in E shown in Figure 6. This the expected quantum oscillator behavior. Furthermore, Figure 7 shows the Q dependence of the scattering intensity along each of the E lines in Figure 6. The expected functional dependence for each successive transition and the overall increase in background expected from multiple scattering are both observed.

As this paper focuses on the GPU implementation, Figure 6 and 7 also show the agreement between the CPU and GPU versions of MCViNE. The majority of the speed increase for this particular simulation is in the incident beam line simulation leveraging the McStas GPU implementation, and is now under an hour rather than days. The MCViNE part of the simulation has a speed up similar to the simpler test from $\sim 10^3$s to $\sim 10^2$s. For a virtual neutron experiment the incident beam simulation can often be reused in a series of source-sample-detector simulations with various sample and detector configurations. For example, a researcher may run the case with and without multiple scattering or a series of related samples. Thus fast sample simulations are critical to the overall speed of experimental analysis which highlights the need for using `mcvine.acc`.

Conclusions

Python and Numba were used successfully to create `mcvine.acc`, a new GPU accelerated version of MCViNE, which has so far achieved significant performance gains over the original CPU implementation. Using Python for this application has helped increase the usability, extensibility, and maintainability of the codebase, while gaining performance benefits of GPUs by using Numba. Additionally, the JIT nature of Numba allowed complex combinations of CUDA kernels to be generated at runtime, which would have been significantly harder to implement in other languages.

The performance gains from using Numba have shown to be beneficial. For a simple isotropic sphere sample, a speedup of 133x was achieved over a 16-core CPU using a consumer-grade GPU. For the more complex UN sample with multiple scattering, a speedup of 10x was achieved over a 16-core CPU. These performance gains are crucial for current simulations that take on the order of days to weeks to complete. However, there are still opportunities to further optimize these simulations to better leverage the full capability of the GPU.

Using Numba for GPU acceleration has enabled more sophisticated data analysis for neutron scattering and instrument design, while overall lowering the development cost needed to obtain significant performance improvements. The techniques used in this project could also be applied to other scientific computing applications.

Acknowledgements

For initial Framework development and instrument component development, this research used resources of the Spallation Neutron Source Second Target Station Project at Oak Ridge National Laboratory (ORNL). Sample development was sponsored by ORNL's Laboratory Director's Research and Development Fund. ORNL is managed by UT-Battelle LLC for DOE's Office of Science, under Contract No. DE-AC05-00OR22725.

The US government retains and the publisher, by accepting the article for publication, acknowledges that the US government retains a nonexclusive, paid-up, irrevocable, worldwide license to publish or reproduce the published form of this manuscript, or allow others to do so, for US government purposes. DOE will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan (http://energy.gov/downloads/doe-public-access-plan).

REFERENCES

[1] J. Y. Y. Lin, H. L. Smith, G. E. Granroth, D. L. Abernathy, M. D. Lumsden, B. Winn, A. A. Aczel, M. Aivazis, and B. Fultz, "MCViNE–an object oriented monte carlo neutron ray tracing simulation package," *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, vol. 810, pp. 86–99, 2016, https://doi.org/10.1016/j.nima.2015.11.118.

[2] J. Y. Y. Lin, F. Islam, G. Sala, I. Lumsden, H. Smith, M. Doucet, M. B. Stone, D. L. Abernathy, G. Ehlers, J. F. Ankner, and G. E. Granroth, "Recent developments of MCViNE and its applications at SNS," *Journal of Physics Communications*, vol. 3, no. 8, p. 085005, Aug. 2019, https://doi.org/10.1088/2399-6528/ab3622. [Online]. Available: https://doi.org/10.1088%2F2399-6528%2Fab3622

[3] P. Willendrup, E. Farhi, E. Knudsen, U. Filges, and K. Lefmann, *Component Manual for the Neutron Ray-Tracing Package McStas*. Danish Techincal University, 2022.

[4] J. Y. Y. Lin, G. Sala, and M. B. Stone, "A super-resolution technique to analyze single-crystal inelastic neutron scattering measurements using direct-geometry chopper spectrometers," *Review of Scientific Instruments*, vol. 93, no. 2, p. 025101, 2022, https://doi.org/10.1063/5.0079031.

[5] F. Islam, J. Y. Y. Lin, R. Archibald, D. L. Abernathy, I. Al-Qasir, A. A. Campbell, M. B. Stone, and G. E. Granroth, "Super-resolution energy spectra from neutron direct-geometry spectrometers," *Review of Scientific Instruments*, vol. 90, no. 10, p. 105109, 2019, https://doi.org/10.1063/1.5116147.

[6] G. Sala, J. Y. Y. Lin, A. M. Samarakoon, D. S. Parker, A. F. May, and M. B. Stone, "Ferrimagnetic spin waves in honeycomb and triangular layers of $Mn_3Si_2Te_6$," *Phys. Rev. B*, vol. 105, p. 214405, Jun 2022, https://doi.org/10.1103/PhysRevB.105.214405. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevB.105.214405

[7] S.-H. Do, K. Kaneko, R. Kajimoto, K. Kamazawa, M. B. Stone, J. Y. Y. Lin, S. Itoh, T. Masuda, G. D. Samolyuk, E. Dagotto *et al.*, "Damped dirac magnon in the metallic kagome antiferromagnet FeSn," *Physical Review B*, vol. 105, no. 18, p. L180403, 2022, https://doi.org/10.1103/physrevb.105.l180403.

[8] J. C. Leiner, H. O. Jeschke, R. Valentí, S. Zhang, A. T. Savici, J. Y. Y. Lin, M. B. Stone, M. D. Lumsden, J. Hong, O. Delaire, W. Bao, and C. L. Broholm, "Frustrated magnetism in mott insulating $(v_{1-x}cr_x)_2o_3$," *Phys. Rev. X*, vol. 9, p. 011035, Feb 2019, https://doi.org/10.1103/PhysRevX.9.011035. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevX.9.011035

[9] J. Y. Y. Lin, A. A. Aczel, D. L. Abernathy, S. E. Nagler, W. Buyers, and G. E. Granroth, "Using monte carlo ray tracing simulations to model the quantum harmonic oscillator modes observed in uranium nitride," *Physical Review B*, vol. 89, no. 14, p. 144302, 2014, https://doi.org/10.1103/physrevb.89.144302.

[10] E. Mamontov, C. Boone, M. Frost, K. Herwig, T. Huegle, J. Y. Y. Lin, B. McCormick, W. McHargue, A. Stoica, P. Torres *et al.*, "A concept of a broadband inverted geometry spectrometer for the Second Target Station at the Spallation Neutron Source," *Review of Scientific Instruments*, vol. 93, no. 4, p. 045101, 2022, https://doi.org/10.1063/5.0086451.

[11] K. An, A. D. Stoica, T. Huegle, J. Y. Y. Lin, and V. Graves, "MENUS—materials engineering by neutron scattering," *Review of Scientific Instruments*, vol. 93, no. 5, p. 053911, 2022, https://doi.org/10.1063/5.0089783.

[12] G. Sala, M. Mourigal, C. Boone, N. P. Butch, A. Christianson, O. Delaire, A. DeSantis, C. Hart, R. P. Hermann, T. Huegle *et al.*, "CHESS: The future direct geometry spectrometer at the Second Target Station," *Review of Scientific Instruments*, vol. 93, no. 6, p. 065109, 2022, https://doi.org/10.1063/5.0089740.

[13] V. O. Garlea, S. Calder, T. Huegle, J. Y. Y. Lin, F. Islam, A. Stoica, V. B. Graves, B. Frandsen, and S. D. Wilson, "VERDI: Versatile diffractometer with wide-angle polarization analysis for magnetic structure studies in powders and single crystals," *Review of Scientific Instruments*, vol. 93, no. 6, p. 065103, 2022, https://doi.org/10.1063/5.0090919.

[14] G. E. Borgstahl, W. B. O'Dell, M. Egli, J. F. Kern, A. Kovalevsky, J. Y. Y. Lin, D. Myles, M. A. Wilson, W. Zhang, P. Zwart *et al.*, "EWALD: A macromolecular diffractometer for the Second Target Station," *Review of Scientific Instruments*, vol. 93, no. 6, p. 064103, 2022, https://doi.org/10.1063/5.0090810.

[15] Y. Liu, H. Cao, S. Rosenkranz, M. Frost, T. Huegle, J. Y. Y. Lin, P. Torres, A. Stoica, and B. C. Chakoumakos, "PIONEER, a high-resolution single-crystal polarized neutron diffractometer," *Review of Scientific Instruments*, vol. 93, no. 7, p. 073901, 2022, https://doi.org/10.1063/5.0089524.

[16] S. Qian, W. Heller, W.-R. Chen, A. Christianson, C. Do, Y. Wang, J. Y. Y. Lin, T. Huegle, C. Jiang, C. Boone *et al.*, "CENTAUR—the small-and wide-angle neutron scattering diffractometer/spectrometer for the Second Target Station of the Spallation Neutron Source," *Review of Scientific Instruments*, vol. 93, no. 7, p. 075104, 2022, https://doi.org/10.1063/5.0090527.

[17] C. Do, R. Ashkar, C. Boone, W.-R. Chen, G. Ehlers, P. Falus, A. Faraone, J. S. Gardner, V. Graves, T. Huegle *et al.*, "EXPANSE: A time-of-flight expanded angle neutron spin echo spectrometer at the Second Target Station of the Spallation Neutron Source," *Review of Scientific Instruments*, vol. 93, no. 7, p. 075107, 2022, https://doi.org/10.1063/5.0089349.

[18] J. Ankner, R. Ashkar, J. Browning, T. Charlton, M. Doucet, C. Halbert, F. Islam, A. Karim, E. Kharlampieva, S. Kilbey *et al.*, "Cinematic reflectometry using QIKR, the quite intense kinetics reflectometer," *Review of Scientific Instruments*, vol. 94, no. 1, p. 013302, 2023, https://doi.org/10.1063/5.0122279.

[19] A. Brugger, H. Z. Bilheux, J. Y. Y. Lin *et al.*, "The Complex, Unique, and Powerful Imaging instrument for Dynamics (CUPI2D) at the Spallation Neutron Source," *Review of Scientific Instruments*, vol. 94, no. 5, p. 051301, 2023, https://doi.org/10.1063/5.0131778. [Online]. Available: https://doi.org/10.1063/5.0131778

[20] J. Y. Y. Lin, T. Huegle, L. Coates, and A. D. Stoica, "A realistic guide misalignment model for the Second Target Station instruments at the Spallation Neutron Source," *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, vol. 1047, p. 167881, 2023, https://doi.org/10.1016/j.nima.2022.167881.

[21] M. B. Stone, G. Sala, and J. Y. Y. Lin, "Design of a radial collimator for the SEQUOIA direct geometry chopper spectrometer," *Physica B: Condensed Matter*, vol. 564, pp. 17–21, 2019, https://doi.org/10.1016/j.physb.2018.11.042.

[22] J. L. Niedziela, R. Mills, M. J. Loguillo, H. D. Skorpenske, D. Armitage, H. L. Smith, J. Y. Y. Lin, M. S. Lucas, M. B. Stone, and D. L. Abernathy, "Design and operating characteristic of a vacuum furnace for time-of-flight inelastic neutron scattering measurements," *Review of Scientific Instruments*, vol. 88, no. 10, p. 105116, 2017, https://doi.org/10.1063/1.5007089.

[23] T. E. Mason, D. Abernathy, I. Anderson, J. Ankner, T. Egami, G. Ehlers, A. Ekkebus, G. Granroth, M. Hagen, K. Herwig, J. Hodges, C. Hoffmann, C. Horak, L. Horton, F. Klose, J. Larese, A. Mesecar, D. Myles, J. Neuefeind, M. Ohl, C. Tulk, X.-L. Wang, and J. Zhao, "The Spallation Neutron Source in Oak Ridge: A powerful tool for materials research," *Physica B: Condensed Matter*, vol. 385, pp. 955–960, 2006, https://doi.org/10.1016/j.physb.2006.05.281.

[24] S. K. 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*, 2015, pp. 1–6, https://doi.org/10.1145/2833157.2833162.

[25] J. Lin and C. Kendrick, "mcvine.acc," https://github.com/mcvine/acc. [Online]. Available: https://github.com/mcvine/acc

[26] T. R. Prisk, R. T. Azuah, D. L. Abernathy, G. E. Granroth, T. E. Sherline, P. E. Sokol, J. Hu, and M. Boninsegni, "Zero-point motion of liquid and solid hydrogen," *Phys. Rev. B*, vol. 107, p. 094511, Mar 2023, https://doi.org/10.1103/PhysRevB.107.094511. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevB.107.094511

[27] A. A. Aczel, G. E. Granroth, G. J. MacDougall, W. Buyers, D. L. Abernathy, G. D. Samolyuk, G. M. Stocks, and S. E. Nagler, "Quantum oscillations of nitrogen atoms in uranium nitride," *Nature Communications*, vol. 3, p. 1124, 2012, https://doi.org/10.1038/ncomms2117.

[28] D. L. Abernathy, M. B. Stone, M. J. Loguillo, M. S. Lucas, O. Delaire, X. Tang, J. Y. Y. Lin, and B. Fultz, "Design and operation of the wide angular-range chopper spectrometer ARCS at the Spallation Neutron Source," *Review of Scientific Instruments*, vol. 83, no. 1, 2012, https://doi.org/10.1063/1.3680104.

[29] M. B. Stone, J. L. Niedziela, D. L. Abernathy, L. DeBeer-Schmitt, G. Ehlers, O. Garlea, G. E. Granroth, M. Graves-Brook, A. I. Kolesnikov, A. Podlesnyak, and B. Winn, "A comparison of four direct geometry time-of-flight spectrometers at the Spallation Neutron Source," *Review of Scientific Instruments*, vol. 85, no. 4, p. 045113, 2014, https://doi.org/10.1063/1.4870050.

[30] P. K. Willendrup and K. Lefmann, "McStas (i): Introduction, use, and basic principles for ray-tracing simulations," *Journal of Neutron Research*, vol. 22, no. 1, pp. 1–16, 2020, https://doi.org/10.3233/JNR-190108.

[31] ——, "McStas (ii): An overview of components, their use, and advice for user contributions," *Journal of Neutron Research*, vol. 23, no. 1, pp. 7–27, 2021, https://doi.org/10.3233/JNR-200186.

[32] G. R. Watson, G. Cage, J. Fortney, G. E. Granroth, H. Hughes, T. Maier, M. McDonnell, A. Ramirez-Cuesta, R. Smith, S. Yakubov, and W. Zhou, "Calvera: A platform for the interpretation and analysis of neutron scattering data," in *Accelerating Science and Engineering Discoveries*

Through Integrated Research Infrastructure for Experiment, Big Data, Modeling and Simulation, K. Doug, G. Al, S. Pophale, H. Liu, and S. Parete-Koon, Eds. Cham: Springer Nature Switzerland, 2022, pp. 137–154.

[33] T. Kittelmann, E. Klinkby, E. Knudsen, P. Willendrup, X. Cai, and K. Kanaki, "Monte Carlo Particle Lists: MCPL," *Computer Physics Communications*, vol. 218, pp. 17–42, 2017, https://doi.org/10.1016/j.cpc.2017.04.012. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0010465517301261

[34] G. Granroth, D. L. Abernathy, J. Lin, W. Zhou, and P. K. Wilendrup, "Incident beamline simulation for ei= 510 mev on the ARCS spectrometer at the Spallation Neutron Source," https://doi.org/10.13139/ORNLNCCS/1975747, 6 2023, https://doi.org/10.13139/ORNLNCCS/1975747. [Online]. Available: https://doi.ccs.ornl.gov/ui/doi/438

EEG-to-fMRI Neuroimaging Cross Modal Synthesis in Python

David Calhas[‡§*]

Abstract—Electroencephalography (EEG) and functional magnetic resonance imaging (fMRI) are two ways of recording brain activity; the former provides good time resolution but poor spatial resolution, while the converse is true for the latter. Recently, deep neural network models have been developed that can synthesize fMRI activity from EEG signals, and vice versa. Because these generative models simulate data, they make it easier for neuroscientists to test ideas about how EEG and fMRI signals relate to each other, and what both signals tell us about how the brain controls behavior. To make it easier for researchers to access these models, and to standardize how they are used, we developed a Python package, EEG-to-fMRI, which provides cross modal neuroimaging synthesis functionalities. This is the first open source software enabling neuroimaging synthesis. Our main focus is for this package to help neuroscience, machine learning, and health care communities. This study gives an in-depth description of this package, along with the theoretical foundations and respective results.

Index Terms—Electroencephalography, Functional Magnetic Resonance Imaging, Synthesis, Deep Learning, Learning, Machine Learning, Computer Vision

Introduction

Neuronal activity, usually measured through electroencephalography (EEG), is related to haemodynamical activity, measured through functional magnetic resonance imaging (fMRI). The first captures the dynamics of the electrical field, whose source is located from the firing neurons' action potentials. In its turn, the second measures the blood supply dynamics. These two while being studied simultaneously [1], [2], [3], [4], [5], [6], [7], [8] differ in many aspects such as: temporal and spatial resolution, brain functions captured, recording and hardware cost. Recently we have seen several studies that use deep neural network models [9] to learn a mapping from EEG data to and from fMRI data [10], [11]. These are a type of generative models [12], that sample/synthesize instances from a different data source (instead of a distribution). Such a model could allow health care cost reductions and discoveries of new neuroscience insights on the relationship between these two modalities. Indeed, pathologies that require MRI scans diagnostics benefit from a lower cost EEG assessment, since availability of MRI hardware is very scarce [13]. As Python [14] becomes a hub for scientific development [15], [16], [17] we find the need to provide open source software that provides solutions for *EEG to fMRI synthesis*, urgent, in order for third party scientific contributions coming from other laboratories to coexist and health care software integration to develop for diagnostic settings. To that end, we provide a description of the open source software EEG-to-fMRI, which originated from an academic scientific project funded by Fundação para a Ciência e Tecnologia, and make publicly available a github repository.

* *Corresponding author: david.calhas@tecnico.ulisboa.pt*
‡ *INESC-ID*
§ *Instituto Superior Tecnico*

Methods

The mapping function provided in this software is the one proposed by [11]. It consists on transforming the EEG from a channel by time representation to a channel by time-frequency one, achieved using the short time Fourier transform [18] by means of the fast Fourier transform (scipy.fft.fft) available in the SciPy package. The latter corresponds to the second step of the diagram illustrated in Figure 1. In the second step, this representation is then forward through a deep neural network (implemented as a tf.keras.Model), with contributions ranging from Resnet blocks [19], an automated machine learning framework [20] and Fourier features [21]. Ultimately, this enables the prediction of an fMRI volume associated with a 26 second segment of an EEG recording.

Description

The dependencies of each component are described in an UML diagram. Overall to install the package, the user is required to install the following dependencies: `tensorflow 2.9.0`, `matplotlib 3.5.3`, `mne 0.23.4`, `nilearn 0.7.0`, `tensorflow_probability 0.12.2`, `tensorflow_determinism 0.3.0`, and `tensorflow_addons 0.19.0`. As seen there is a high dependency in tensorflow related packages. This is due to the whole system provided being built in tensorflow, a library that enables automatic differentiation and is widely used for deep learning model development.

Package modules

This package, as it is provided, has eight main modules:

- *models*: here you will find the code that implements the models for synthesis and classification. The synthesis models are located in the *synthesizers.py*, where the class *EEG_to_fMRI* is implemented. This class defines a `tf.keras.Model`, composed of two encoders, one for the EEG and another for the fMRI. Additionally, there is

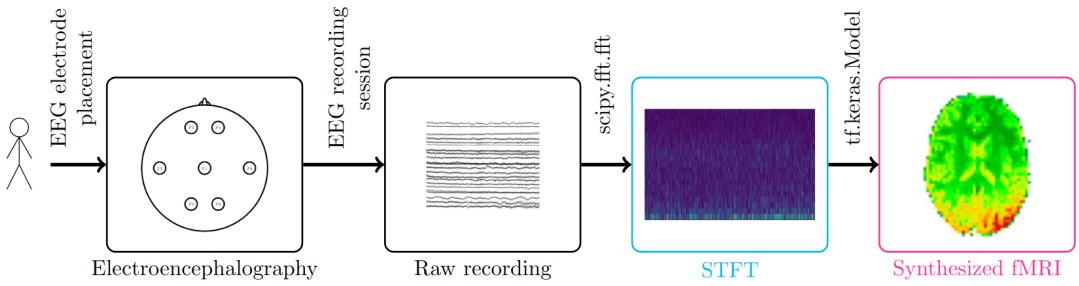

Fig. 1: *The pipeline of the EEG to fMRI synthesis project consists of processing EEG recordings, that are sourced from a human subject that goes through an EEG recording sesssion, then processing the channel by time signal and taking the short time Fourier transform. The latter, gives us the time frequency representation of the EEG signal. The novelty of this software is that it provides a model that given as input the EEG signal it predicts the corresponding fMRI volume associated with that segment. A video demonstration of the whole pipeline is available on Youtube.*

a decoder that maps the latent EEG representation, that is the output of the EEG encoder, to the estimate of the fMRI volume associated. The fMRI encoder is defined in the *fmri_ae.py* file. The task would not be complete without the extrapolation of the synthesis model to a classification task. To that end, two linear classifiers are provided, the *ViewLatentContrastiveClassifier* and the *ViewLatentLikelihoodClassifier*, corresponding to a contrastive latent loss and a cross entropy error driven classification, respectively. The extrapolation is made by taking the output of the neural flow that comes from the EEG, that is the EEG encoder and the decoder.

- *layers*: this module contains the layers which compose the synthesizer models. It provides five main types of layers. First, the *TopographicalAttention* computes a self attention mechanism in the channels dimension of the EEG [11] that allows the representation to be correctly processed by the convolutional blocks. Second, the *ResnetBlock* [19] implements a residual block composed of convolutional layers. This block allows efficient gradient propagation, by tackling the vanishing gradient phenomena. Third, the *FourierFeatures* layer projects *cosines* functions of different shifts and biases to build a latent spectral basis for the prediction of the fMRI volume. Fourth, the *DenseVariational* is an implementation of the `tf.keras.layers.Dense` layer for two-dimensional inputs, whose weights are drawn from gaussian distributions. Last, but not least, we provide the *DCT* based layers, which implement the discrete cosine transform [22]. These layers are useful for alternative ways of decoding the latent representation of the EEG to produce the desired fMRI volume.
- *regularizers*: here the implementation of different regularizers for the synthesizer model is provided. Most importantly in the *activity_regularizers.py* file is found the *OrganizeChannels* implementation which can be added in the *TopographicalAttention* layer, so that the layer does not surpress any channel;
- *learning*: here are implemented the routines for the optimization of the models, namely the *losses* and the *train* procedures. The *train.py* is a generalizable training routine that fits any type of `tf.keras.Model` instance. Following, the *losses.py* contain a set of losses that are implemented to train the models provided in the *models* module. Most importantly, here you will find the *mae_cosine* for the deterministic versions of the *EEG_to_fMRI* model, the *LaplacianLoss* for variational versions, and the *ContrastiveClassificationLoss* which serves as the cost function for the *ViewLatentContrastiveClassifier*;
- *explainability*: this module contains explainability methods that were employed for the models developed. The implementation for the layer wise relevance propagation [23] can be found in *lrp.py*. In particular, since this type of algorithm is not model agnostic, we have the implementation for all the permutations' of operations that the *EEG_to_fMRI* model can have;
- *data*: this is maybe the most important module for researchers wanting to try out their collected data with this software. Here all of the functions, that read data and manipulate it as such to allow the efficient training, are implemented. Starting with the *eeg_utils.py*, where the *get_eeg_instance_<DS>* function is implemented for a limited set of datasets that participated in the experiments of the associated studies. The <DS> stands for the identifier of the dataset. Currently there are a set of functions implemented for publicly available datasets. Please check the description of the code in the github repository for more details. In the *fmri_utils.py* file one finds functions of the form *get_individuals_paths_<DS>*, that have the exact same function as the functions to read EEG recordings, but in this case they read nifti file format. These type of files are the standard format for fMRI recordings. Finally, the *data_utils.py* and *preprocessed_data.py* are responsible for concatenating the EEG and fMRI instance pairs, with all the alignments and event synchronization events taken into account, as well as the processing to build `tf.data.Dataset` classes.
- *metrics*: in this module are implemented the metrics usually used to evaluate synthesis of generated images, such as the root mean squared error, mean absolute error, structural similarity index measure [24], among others;
- *utils*: Last but not least, is the utilities module, which provides the user with print functions, configuration of the tensorflow environments and several visualizations that were used in the original studies that developed the package.

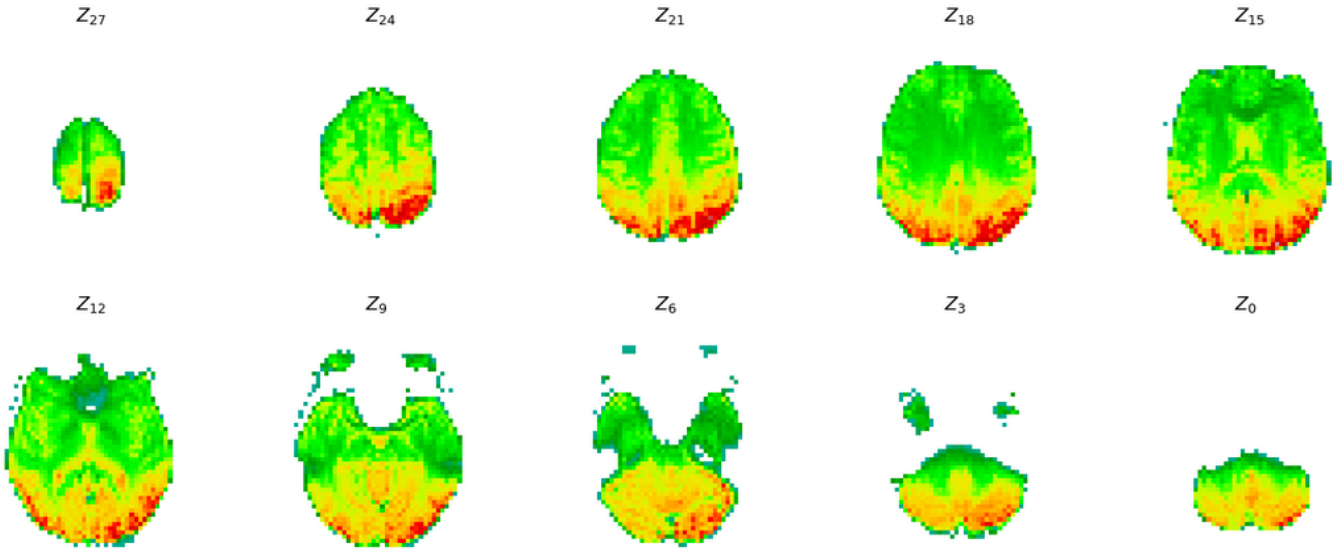

Fig. 2: *The synthesized signal of fMRI. This is the output visualization when running the code for the classification notebook.*

New data integration

In order to use the software provided for new data, we recommend that the dataset is structured as shown in Figure 3. If the data is

```
NEW
|       .
|       ..
|       README.md
└──────EEG
|       └──────sub-001
|       |       |       FILE.eeg
|       |       |       FILE.vhdr
|       |       |       FILE.vmrk
|       └──────sub-002
|       ...
└──────BOLD
        └──────sub-001
        |       |       FILE.anat
        |       |       FILE.nii.gz
        └──────sub-002
        ...
```

Fig. 3: *Recommended structure of the dataset directory.*

provided as illustrated then the user only has to name the directory of the data as *01*. This should suffice for the correct loading of the data. Any type of issue that is encountered for this package should be published as an issue in the official github repository.

Building an EEG to fMRI model

For the user to build an EEG to fMRI model, they have to first import the correct library module and *tensorflow*.

```
from tensorflow as tf
from eeg_to_fmri.models.synthesizers
        import EEG_to_fMRI
from eeg_to_fmri.models.synthesizers
        import parameters
from eeg_to_fmri.models.synthesizers
        import na_specification_eeg
from eeg_to_fmri.models.fmri_ae
        import na_specification_fmri
```

Let us define the size of the EEG representation $\vec{x} \in \mathbb{R}^{64\times134\times10\times1}$, the fMRI representation $\vec{y} \in \mathbb{R}^{64\times64\times30\times1}$, and the latent representation, for both the EEG and fMRI, that is $\vec{z}_x, \vec{z}_y \in \mathbb{R}^{7\times7\times7}$.

```
fmri_dim=(64,134,10,1)
fmri_dim=(64,64,30,1)
latent_dim=(7,7,7)
```

Then, we have to define the parameters for the model, these are provided as a variable in the *synthesizers.py*.

```
learning_rate,weight_decay,kernel_size,
stride_size,batch_size,latent_dimension,
n_channels,max_pool,batch_norm,
skip_connections,dropout,
n_stacks,outfilter,local=parameters
```

Some of these parameters will also be used to define the fMRI encoder. Since the fMRI encoder is a different class, we need to define the initialization parameters.

```
fmri_parameters=(parameters, latent_dim,
    fmri_dim, kernel_size, stride_size,
    n_channels, max_pool, batch_norm,
    weight_decay, skip_connections,
    n_stacks, True, False,
    outfilter, dropout, None,
    False, na_specification_fmri)
```

Next, we have all of the parameters necessary to build a simple deterministic version of the EEG to fMRI model.

```
with tf.device('/CPU:0'):
    model = EEG_to_fMRI(latent_dim, eeg_dim,
        na_specification_eeg, n_channels,
        weight_decay=weight_decay,
        skip_connections=True, batch_norm=True,
        fourier_features=True,
        random_fourier=True,
        topographical_attention=True,
        conditional_attention_style=True,
        conditional_attention_style_prior=False,
        local=True, seed=None,
        fmri_args = fmri_parameters)
```

The model is built once it is specified the input dimension, this is done through the *tf.keras.Layer#build*. This will initialize all of the weights of the network.

```
model.build((None,)+eeg_dim, (None,)+fmri_dim)
```

Cost function and optimization

Regarding the cost function, which is provided in the *learning* module at the *losses.py* file, we can specify different metrics that are provided. Take, for instance, the example of using an approximation of the mean absolute error at the output for the fMRI volume prediction $\hat{\vec{y}}$, and approximation of the latent representations of the fMRI as proposed by [11]. This is reduced to the mathematical formula

$$\mathscr{L}(\vec{x},\vec{y},\vec{z}_x,\vec{z}_y) = ||\vec{y}-\hat{\vec{y}}||_1^1 + 1 - \frac{\vec{z}_x \cdot \vec{z}_y}{||\vec{z}_x||_2^2 ||\vec{z}_y||_2^2}. \quad (1)$$

In terms of code, this is already implemented and can be loaded directly.

```
from eeg_to_fmri.learning import losses
loss_fn=losses.mae_cosine
```

The optimizer is already provided in the `tensorflow` library and its corresponding learning rate is given in the parameters variable.

```
optimizer=
        tf.keras.optimizers.Adam(learning_rate)
```

Training the model requires an input, the EEG $\vec{x}$, and an output, the fMRI $\vec{y}$. The architecture processes both the EEG and fMRI, producing the latent representation for both. Proceeding the latent EEG representation, $\vec{z}_x$, is fed to the decoder which estimates the fMRI $\hat{\vec{y}}$.

```
def apply_gradient(model, optimizer, loss_fn,
    x, y, return_logits=False, call_fn=None):
    with tf.GradientTape(persistent=True) as
                                          tape:
        logits=model(x, y, training=True)
        regularization=0.
        if(len(model.losses)):
            regularization=
                tf.math.add_n(model.losses)
        loss=loss_fn(y, logits)+regularization
    gradients=tape.gradient(loss,
            model.trainable_variables)
    optimizer.apply_gradients(zip(gradients,
        model.trainable_variables))
    return tf.reduce_mean(loss)
```

The training routine functions are found in the *learning* module. The function *apply_gradient* computes the forward and backward pass of the neural network. Note that the input of the *EEG_to_fMRI* model is composed of two tensors, the EEG and fMRI, which is the reason why one gives the model both x and y. The loss computes both the estimation of the fMRI according to the mean absolute error (MAE) and the cosine distance between the latent representations. Regularization terms, such as weight decay and other activity regularizers that may or may not participate in the model, are also added to the loss. The loss function used for the synthesis task, the one used in [11], is the MAE and the cosine distance. This loss function is defined in the *learning* module, found at the *losses.py* file. This loss receives a *tf.Tensor* object, *y_true*, and a list of *tf.Tensor*. The list of tensors contains the outputs of the neural network that should be approximate the ground truth. The first element of the list is the estimated fMRI volume and the second and third items are the latent EEG and fMRI representations, respectively.

```
def mae_cosine(y_true, y_pred):
    return tf.reduce_mean(
        tf.math.abs(y_pred[0] - y_true),
        axis=(1,2,3)) +
        cosine(y_pred[1], y_pred[2])
```

At test time, the *EEG_to_fMRI* model can discard the fMRI input. To that end, only the decoder attribute is called, which is composed of the EEG encoder and the decoder.

```
def call(self, x1, x2):
    if(self.training):
        return [self.decoder(x1),
            self.eeg_encoder(x1),
            self.fmri_encoder(x2)]

    return self.decoder(x1)
```

Note that, the *pretrained_EEG_to_fMRI* class processes a pretrained model of the type *EEG_to_fMRI* and builds a new model that only processes EEG and outputs an fMRI prediction given the representation learned. This class is built to be then appended with a classifier, that can be either a *ViewLatentContrastiveClassifier* or a *ViewLatentLikelihoodClassifier*.

Examples

In this section, we walk through the examples given in jupyter notebooks.

Synthesis

We provide a compressed version of the dataset of [25]. Users can directly execute the code and have both the python package, as well as the dataset, setup in a google colab environment. The flow of execution has been already described. In the end a synthesized fMRI is shown, as illustrated in Figure 2. This image is built using the *viz_utils.py*. The user can find metrics for synthesis evaluation in *eeg_to_fmri.metrics.quantitative_metrics*. We report results from the [11] study on the NODDI dataset [25]. An example with a reduced dataset is available in this synthesis notebook. The best model, which used the configuration of the *eeg_to_fmri.models.synthesizers.EEG_to_fMRI* achieved 0.3972 RMSE and 0.4613 SSIM. This constitutes the state-of-the-art for this task and provides a view that can be applied in EEG only datasets for classification task.

Classification

We also provide a compressed version of the dataset of [26]. This example, available in this classification notebook, is based on a publicly available dataset that contains individuals diagnosed with schizophrenia and healthy controls. The whole goal of the project is to be applied in an health care setting and to this end we employ an end to end software solution. The whole software package is able to synthesize fMRI and adapt to a classification setting, that given EEG recordings outputs a set of probabilities for each group of people considered in the dataset.

Collaboration

Ultimately, the goal of this package is to collect, in one package, methods for EEG to fMRI synthesis. We welcome contributions from authors of related work such as [10]. In the future, we plan to add a module or example folder with implementations of these approaches, so that other research groups can easily access them and reproduce key results.

On a higher level, this software is encouraged for testing its applicability in health care settings. The impact, that such mappings from EEG to fMRI, would have on society is enormous, given that

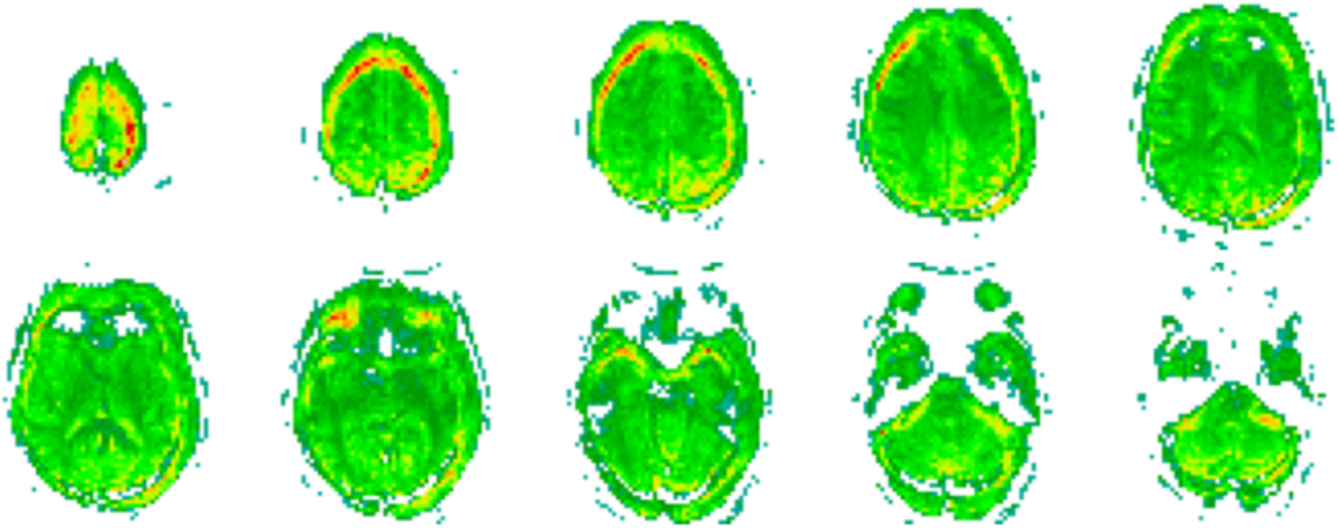

Fig. 4: *Output of the predicted fMRI when given an EEG representation. Note that, due to the EEG encoder being optimized towards classifying the data according to the groups of individuals defined, e.g. schizophrenic and healthy controls, the decoder (that has the parameters frozen) gives a slightly altered representation. This change is seen in the produced fMRI, where activity beyond the limit of the human scalp is reported. Please recall Figures 1 and 2 to directly compare with an fMRI representation without these flaws.*

the diagnostic is faithful. Take for instance the example of the MRI machine density across the African continent. In the worst case scenario, Nigeria has a density of 0.33 MRI machines per million people, according to [13]. To let that sink in, imagine having to wait in a line of 3 million people to get a diagnostic exam. This type of waiting bottleneck impacts greatly the development of diseases for the worse. Countries in such conditions would greatly benefit from contributions that further advance this scientific field. Even fortunate countries, whose economy thrives, that are able to provide their populations with a good ratio of MRI machines, they still have small portions of the population who live in remote areas. These people find it hard to get quality health care, without having to travel significant distances.

Conclusion

This is the first package, to the best of our knowledge, that provides a machine learning oriented synthesis between functional neuroimaging modalities (EEG and fMRI). It is targeted to help the neuroscience community, in tasks such as modality augmentation, resolution enhancement, neuroimaging explainability techniques, among others. We hope to motivate researchers, scientists, and software developers to contribute to this package which we have been so passionate about throughout the last years.

Acknowledgments

This work was supported by national funds through Fundação para a Ciência e Tecnologia under the PhD Grant SFRH/BD/5762/2020 to David Calhas.

References

[1] H. Shibasaki, "Human brain mapping: hemodynamic response and electrophysiology," *Clinical Neurophysiology*, vol. 119, no. 4, pp. 731–743, 2008, https://doi.org/10.1016/j.clinph.2007.10.026.

[2] Q. Yu, L. Wu, D. A. Bridwell, E. B. Erhardt, Y. Du, H. He, J. Chen, P. Liu, J. Sui, G. Pearlson *et al.*, "Building an eeg-fmri multi-modal brain graph: a concurrent eeg-fmri study," *Frontiers in human neuroscience*, vol. 10, p. 476, 2016, https://doi.org/10.3389/fnhum.2016.00476.

[3] Y. He, M. Steines, J. Sommer, H. Gebhardt, A. Nagels, G. Sammer, T. T. J. Kircher, and B. Straube, "Spatial–temporal dynamics of gesture–speech integration: a simultaneous eeg-fmri study," *Brain Structure and Function*, vol. 223, pp. 3073–3089, 2018, https://doi.org/10.1007/s00429-018-1674-5.

[4] G. M. Rojas, C. Alvarez, C. E. Montoya, M. de la Iglesia-Vayá, J. E. Cisternas, and M. Gálvez, "Study of resting-state functional connectivity networks using eeg electrodes position as seed," *Frontiers in neuroscience*, vol. 12, p. 235, 2018, https://doi.org/10.3389/fnins.2018.00235.

[5] L. Bréchet, D. Brunet, G. Birot, R. Gruetter, C. M. Michel, and J. Jorge, "Capturing the spatiotemporal dynamics of self-generated, task-initiated thoughts with eeg and fmri," *Neuroimage*, vol. 194, pp. 82–92, 2019.

[6] I. Daly, D. Williams, F. Hwang, A. Kirke, E. R. Miranda, and S. J. Nasuto, "Electroencephalography reflects the activity of sub-cortical brain regions during approach-withdrawal behaviour while listening to music," *Scientific reports*, vol. 9, no. 1, pp. 1–22, 2019, https://doi.org/10.1038/s41598-019-45105-2.

[7] C. Cury, P. Maurel, R. Gribonval, and C. Barillot, "A sparse eeg-informed fmri model for hybrid eeg-fmri neurofeedback prediction," *Frontiers in neuroscience*, vol. 13, p. 1451, 2020, https://doi.org/10.3389/fnins.2019.01451.

[8] R. Abreu, J. Jorge, A. Leal, T. Koenig, and P. Figueiredo, "Eeg microstates predict concurrent fmri dynamic functional connectivity states," *Brain topography*, vol. 34, no. 1, pp. 41–55, 2021, https://doi.org/10.1007/s10548-020-00805-1.

[9] I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning*. MIT press, 2016.

[10] X. Liu *et al.*, "A convolutional neural network for transcoding simultaneously acquired eeg-fmri data," in *NER*. IEEE, 2019, https://doi.org/10.1109/ner.2019.8716994.

[11] D. Calhas and R. Henriques, "Eeg to fmri synthesis benefits from attentional graphs of electrode relationships," *arXiv preprint arXiv:2203.03481*, 2022.

[12] K. P. Murphy, *Machine learning: a probabilistic perspective*. MIT press, 2012.

[13] G. I. Ogbole, A. O. Adeyomoye, A. Badu-Peprah, Y. Mensah, and D. A. Nzeh, "Survey of magnetic resonance imaging availability in west africa," *Pan African Medical Journal*, vol. 30, no. 1, 2018, https://doi.org/10.11604/pamj.2018.30.240.14000.

[14] G. Van Rossum and F. L. Drake Jr, *Python tutorial*. Centrum voor Wiskunde en Informatica Amsterdam, The Netherlands, 1995.

[15] C. R. Harris, K. J. Millman, S. J. Van Der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith *et al.*, "Array programming with numpy," *Nature*, vol. 585, no. 7825, pp. 357–362, 2020.

[16] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright *et al.*,

"Scipy 1.0: fundamental algorithms for scientific computing in python," *Nature methods*, vol. 17, no. 3, pp. 261–272, 2020.

[17] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: a system for large-scale machine learning." in *Osdi*, vol. 16. Savannah, GA, USA, 2016, pp. 265–283.

[18] J. Allen, "Short term spectral analysis, synthesis, and modification by discrete fourier transform," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 25, no. 3, pp. 235–238, 1977, https://doi.org/10.1109/tassp.1977.1162950.

[19] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778, https://doi.org/10.1109/cvpr.2016.90.

[20] D. Calhas, V. M. Manquinho, and I. Lynce, "Automatic generation of neural architecture search spaces," in *Combining Learning and Reasoning: Programming Languages, Formalisms, and Representations*, 2022.

[21] M. Tancik, P. P. Srinivasan, B. Mildenhall, S. Fridovich-Keil, N. Raghavan, U. Singhal, R. Ramamoorthi, J. T. Barron, and R. Ng, "Fourier features let networks learn high frequency functions in low dimensional domains," *arXiv preprint arXiv:2006.10739*, 2020.

[22] N. Ahmed, T. Natarajan, and K. R. Rao, "Discrete cosine transform," *IEEE transactions on Computers*, vol. 100, no. 1, pp. 90–93, 1974, https://doi.org/10.1016/b978-0-08-092534-9.50007-2.

[23] S. Bach, A. Binder, G. Montavon, F. Klauschen, K.-R. Müller, and W. Samek, "On pixel-wise explanations for non-linear classifier decisions by layer-wise relevance propagation," *PloS one*, vol. 10, no. 7, p. e0130140, 2015, https://doi.org/10.1371/journal.pone.0130140.

[24] Z. Wang, A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli, "Image quality assessment: from error visibility to structural similarity," *IEEE transactions on image processing*, vol. 13, no. 4, pp. 600–612, 2004, https://doi.org/10.1109/tip.2003.819861.

[25] F. Deligianni, M. Centeno, D. W. Carmichael, and J. D. Clayden, "Relating resting-state fmri and eeg whole-brain connectomes across frequency bands," *Frontiers in Neuroscience*, vol. 8, p. 258, 2014, https://doi.org/10.3389/fnins.2014.00258. [Online]. Available: https://www.frontiersin.org/article/10.3389/fnins.2014.00258

[26] A. Padée *et al.*, ""fribourg ultimatum game in schizophrenia study"," 2022, https://doi.org/10.18112/openneuro.ds004000.v1.0.0.

vak: a neural network framework for researchers studying animal acoustic communication

David Nicholson[‡*], Yarden Cohen[§]

Abstract—How is speech like birdsong? What do we mean when we say an animal learns their vocalizations? Questions like these are answered by studying how animals communicate with sound. As in many other fields, the study of acoustic communication is being revolutionized by deep neural network models. These models enable answering questions that were previously impossible to address, in part because the models automate analysis of very large datasets. Acoustic communication researchers have developed multiple models for similar tasks, often implemented as research code with one of several libraries, such as Keras and Pytorch. This situation has created a real need for a framework that allows researchers to easily benchmark multiple models, and test new models, with their own data. To address this need, we developed vak (https://github.com/vocalpy/vak), a neural network framework designed for acoustic communication researchers. ("vak" is pronounced like "talk" or "squawk" and was chosen for its similarity to the Latin root *voc*, as in "vocal".) Here we describe the design of the vak, and explain how the framework makes it easy for researchers to apply neural network models to their own data. We highlight enhancements made in version 1.0 that significantly improve user experience with the library. To provide researchers without expertise in deep learning access to these models, vak can be run via a command-line interface that uses configuration files. Vak can also be used directly in scripts by scientist-coders. To achieve this, vak adapts design patterns and an API from other domain-specific PyTorch libraries such as torchvision, with modules representing neural network operations, models, datasets, and transformations for pre- and post-processing. vak also leverages the Lightning library as a backend, so that vak developers and users can focus on the domain. We provide proof-of-concept results showing how vak can be used to test new models and compare existing models from multiple model families. In closing we discuss our roadmap for development and vision for the community of users.

Index Terms—animal acoustic communication, bioacoustics, neural networks

Introduction

Are humans unique among animals? We seem to be the only species that speaks languages [1], but is speech somehow like other forms of acoustic communication in other animals, such as birdsong [2]? How should we even understand the ability of some animals to learn their vocalizations [3]? Questions like these are answered by studying how animals communicate with sound [4]. As others have argued, major advances in this research will require cutting edge computational methods and big team science across a wide range of disciplines, including ecology, ethology, bioacoustics, psychology, neuroscience, linguistics, and genomics [5], [6], [3], [1].

Research on animal acoustic communication is being revolutionized by deep learning algorithms [5], [6], [7]. Deep neural network models enable answering questions that were previously impossible to address, in part because these models automate analysis of very large datasets. Within the study of animal acoustic communication, multiple models have been proposed for similar tasks--we review these briefly in the next section. These models have been implemented using a range of frameworks for neural networks, including PyTorch (as in [8] and [9]), Keras and Tensorflow (as in [10] and [11]), and even in programming environments outside Python such as Matlab (as in [12]). Because of this, it is difficult for researchers to directly compare models, and to understand how each performs on their own data. Additionally, many researchers will want to experiment with their own models to better understand the fit between tasks defined by machine learning researchers and their own question of interest. All of these factors have created a real need for a framework that allows researchers to easily benchmark models and apply trained models to their own data.

To address this need, we developed vak [13] (https://github.com/vocalpy/vak), a neural network framework designed for researchers studying animal acoustic communication. vak is already in use in at least 10-20 research groups to our knowledge, and has already been used in several publications, including [8], [9], [14], [15]. Here we describe the design of the vak framework, and explain how vak makes it easy for acoustic communication researchers to work with neural network models. We have also recently published an alpha release of version 1.0 of the library, and throughout this article we highlight enhancements made in this version that we believe will significantly improve user experience.

Related work

First, we briefly review related literature, to further motivate the need for a framework. A very common workflow in studies of acoustic behavior is to take audio recordings of one individual animal and segment them into a sequence of units, after which further analyses can be done, as reviewed in [16]. Some analyses require further annotation of the units to assign them to one of some set of classes, e.g. the unique syllables within an individual songbird's song. An example of segmenting audio of Bengalese finch song into syllables and annotating those syllables is shown in Figure 1.

Several models have been developed to detect and classify a large dataset of vocalizations from an individual animal. These are

* *Corresponding author: nicholdav@gmail.com*
‡ *Independent researcher, Baltimore, Maryland, USA*
§ *Weizmann Institute of Science, Rehovot, Israel*

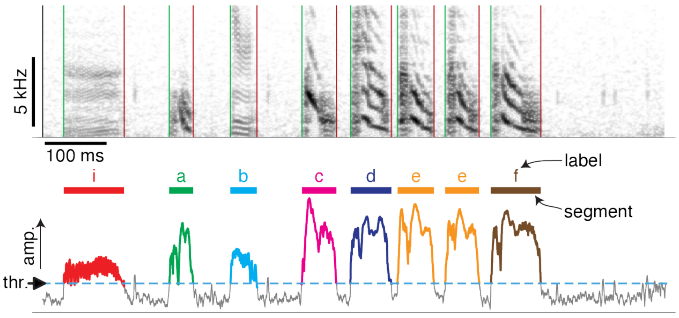

Fig. 1: *Schematic of analyzing acoustic behavior as a sequence of units. Top panel shows a spectrogram of an individual Bengalese finch's song, consisting of units, often called syllables, separated by brief silent gaps. Bottom panel illustrates one method for segmenting audio into syllables that are annotated: a threshold is set on the audio amplitude to segment it into syllables (a continuous period above the threshold), and then a human annotator labels each syllable (e.g., with a GUI application). Adapted from [8] under CC BY 4.0 license.*

all essentially supervised machine learning tasks. Some of these models seek to align a neural network task with the common workflow just described [16], where audio is segmented into a sequence of units with any of several methods [17], that are then labeled by a human annotator. The first family of neural network models reduces this workflow to a frame classification problem [18], [19]. That is, these models classify a series of *frames*, like the columns in a spectrogram. Sequences of units (e.g., syllables of speech or birdsong) are recovered from this series of frame classifications with post-processing. Essentially, the post-processing finds the start and stop times of each continuous run of a single label. Multiple neural network models have been developed for this frame classification approach, including [8] and [20]. A separate approach from frame classification models has been to formulate recognition of individual vocalizations as an object detection problem. To our knowledge this has been mainly applied to mouse ultrasonic vocalizations as in [12].

Another line of research has investigated the use of unsupervised models to learn a latent space of vocalizations. This includes the work of [11] and [9]. These unsupervised neural network models allow for clustering vocalizations in the learned latent space, e.g., to efficiently provide a human annotator with an estimate of the number of classes of vocalizations in an animal's repertoire [11], and/or to measure similarity between vocalizations of two different animals [9], [21]. It is apparent that unsupervised approaches are complementary to supervised models that automate labor-intensive human annotation. This is another reason that a single framework should provide access to both supervised and unsupervised models.

Methods

In this section we describe the design of vak: its application programming interface (API) and its command-line interface (CLI). We begin by introducing the design of vak at the highest level.

Design

vak relies on PyTorch [22] for neural networks, because PyTorch accommodates Pythonic idioms and low-level control flow within networks when needed. In version 1.0, we have additionally adopted the Lightning library [23] as a backend, freeing us up as developers to focus on the research domain while benefiting from the Lightning team's engineering expertise. Of course, vak relies heavily on the core libraries of the scientific Python stack. Many functions make use of numpy [24], [25], scipy [26], and matplotlib [27], [28]. In particular, the built-in workflows for preparing datasets make frequent use of pandas [29] to work with tabular data formats, and dask [30] to enable scalable, distributed processing of very large datasets with mixed file formats, which are common in acoustic communication research. Functionality for preparing datasets is specifically tailored to the needs of acoustic communication researchers in other ways as well. For example, to parse the wide range of annotation formats used by acoustic communication researchers across disciplines, we use the pyOpenSci package crowsetta [31].

In terms of its API, the design of vak is most similar to other domain-specific libraries developed with torch, such as torchvision [32], but here the domain is animal acoustic communication research. (Perhaps surprisingly, many of the models proposed to date in this area are essentially adopted from computer vision.) Thus, similar to the torchvision API, vak provides modules for neural network models, operations, transformations for loading data, and datasets.

In addition to its torchvision-like API, vak provides a simple command-line interface (CLI) that allows researchers to work with neural network models without requiring significant expertise in Python programming or deep learning. We first describe the API, so that key concepts have been introduced when we explain the usage of the CLI.

Models

As its name implies, the `models` module is where implementations of neural network models are found. Our design is focused on a user who wants to benchmark different models within an established task and data processing pipeline as defined by our framework. In version 1.0 of vak, we have introduced abstractions that make it easier for researchers to work with the built-in models and with models they declare in code outside of the library, e.g., in a script or notebook. At a high level, we achieved this by adopting the Lightning library as a backend. By sub-classing the core `lightning.LightningModule` class, we provide users with per-model implementations of methods for training, validation, and even for forwarding a single batch or sample through the model. We briefly describe the abstractions we have developed to make it easier to work with models.

Abstractions for declaring a model in vak

Our goal is to make it so that a scientist-coder is able to use any of the built-in models, and experiment with their own models, without needing to contribute code to vak or to use a developer-focused mechanism like entry points. To achieve this, we provide a decorator, `vak.models.model`, that is applied to a *model definition* to produce a sub-class of a *model family*. The `vak.models.model` decorator additionally adds any class it decorates to a *registry*. In the rest of the section we explain these abstractions and how they make it possible to easily test different models.

A model definition takes the form of a class with four required class variables: `network`, `loss`, `optimizer`, and `metrics`. In other words, our abstraction asserts that the definition of a neural network model consists of the neural network function, the loss function used to optimize the network's parameters, the optimizer, and the metrics used to assess performance.

To relate a model as declared with a definition to the machine learning tasks that we implement within the vak framework, we introduce the concept of model *families*. A model family is represented by a sub-class of the core `lightning.LightningModule` class. Each class representing a family implements family-specific methods: `training_step`, `validation_step`, `prediction_step`, and `forward`. In this way, model families are defined operationally: a model can belong to a family if it accepts the inputs provided by logic within the training, validation, and prediction steps, and the model also produces the appropriate outputs needed within those same steps.

With these two abstractions in hand, we can add models to vak as follows: we start by applying the `model` decorator to create a new subclass of a model family. This new subclass has the same name as the class that it decorates, which is the class representing the model definition. The decorator then adds a single attribute to this sub-class, the `definition`, that is used when initializing a new instance of the specific model. After creating this sub-class and adding this attribute, the `model` decorator finally registers the model within the `vak.models.registry` module. This allows other functions within vak to find the model by its name in the registry. The registry is implemented with its own helper functions and module-level `dict` variables that are updated by those functions. We present a listing that demonstrates usage of the abstractions just described.

```
from vak.models import (
    model,
    FrameClassificationModel
)
from vak.metrics import (
    Accuracy,
    Levenshtein,
    SegmentErrorRate,
)

@model(family=FrameClassificationModel)
class TweetyNoLSTMNet:
    """TweetyNet model without LSTM layer"""
    network = TweetyNetNoLSTM
    loss = torch.nn.CrossEntropyLoss
    optimizer = torch.optim.Adam
    metrics = {
        'acc': Accuracy,
        'levenshtein': Levenshtein,
        'segment_error_rate': SegmentErrorRate,
        'loss': torch.nn.CrossEntropyLoss}
    default_config = {
        'optimizer':
            {'lr': 0.003}
    }
```

This example is used in an experiment accompanying this paper, as described below in Results. That experiment demonstrates how the decorator enables models to be declared and used in a script outside of vak. Here we can notice that we apply the `model` decorator to the class `TweetyNoLSTMNet`, which is the model definition. Notice also that we pass in as an argument to the decorator the name of the model family that we wish to sub-class, `FrameClassificationModel`. When Python's import machinery parses the script, the model class will be created and added to vak's registry, so that it can be found by other functions for training and evaluating models. The models that are built in to vak use the exact same decorator.

Model families

Having introduced the abstraction needed to declare models within the vak framework, we now describe the families we have implemented to date.

Frame classification. As stated in the Related Work section, one way to formulate the problem of segmenting audio into sequences of units so that it can be solved by neural networks is to classify each frame of audio, or a spectrogram produced from that audio, and to then recover segments from this series of labeled frames [18], [19].

This problem formulation works, but an issue arises from the fact that audio signals used by acoustic communication researchers very often vary in length. E.g., a bout of Bengalese finch birdsong can vary from 1-10 seconds, and bouts of canary song can vary roughly from 10 seconds to several minutes. In contrast, the vast majority of neural network models assume a "rectangular" tensor as input and output, in part because they were originally developed for computer vision applications applied to batches. One way to work around this issue is to convert inputs of varying lengths into rectangular batches with a combination of windowing and padding. E.g., pick a window size w, find the minimum number of consecutive non-overlapping strides s of that window that will cover an entire input x of length T, $s * w \geq T$, and then pad x to a new length $T_{padded} = s * w$. This approach then requires a post-processing step where the outputs are stitched back together into a single continuous sequence x_{padded}. The padding is removed by tracking which time bins are padded, e.g., with a separate vector that acts as a "padded" flag for each time bin. Of course there are other ways to address the issue of varying lengths, such as using the `torch.nn.utils.rnn` API to pad and unpad tensors (or using a different family of neural network models).

Because more than one model has been developed that uses this post-processing approach to solve the problem of frame classification, we define this as a family of models within vak, the `FrameClassification` model. Both the TweetyNet model from [8] and the Deep Audio Segmenter (DAS) from [10] are examples of such models. We provide an implementation of TweetyNet now built directly into vak in version 1.0. We also provide a PyTorch implementation of the Encoder Decoder-Temporal Convolutional (ED-TCN) Network, that was previously applied to frames of video features for an action segmentation task [33]. Below in Results we show how vak can be used to benchmark and compare both models on the same dataset.

Parametric UMAP. To minimally demonstrate that our framework is capable of providing researchers with access to multiple families of models, we have added an initial implementation of a Parametric UMAP model family. The original algorithm for UMAP (Uniform Manifold Approximation and Projection) consists of two steps: computing a graph on a dataset, and then optimizing an embedding of that graph in a lower dimensional space that preserves local relationships between points [34]. The parametrized version of UMAP replaces the second step with optimization of a neural network architecture [35]. Because the parametrized version can be used with a wide variety of neural network functions, we declare this as a family. We provide an implementation of a single model, an encoder with a convolutional front-end that can map spectrograms of units extracted from audio to a latent space. Our implementation is adapted from https://github.com/elyxlz/umap_pytorch and https://github.com/lmcinnes/umap/issues/580#issuecomment-1368649550.

Neural network layers and operations

Like PyTorch, vak provides a module for neural network operations and layers named `nn`. This module contains layers used by more than one network. For example, it includes a 2-D convolutional layer with the `'SAME'` padding provided by Tensorflow, that is used both by the TweetyNet model [8] and by our implementation of the ED-TCN model [33]. (PyTorch has added this padding from version 1.10 on, but we maintain our original implementation for purposes of replicability.) Another example of an operation in `vak.nn` is a PyTorch implementation of the normalized ReLu activation used by [33] with their ED-TCN model.

Transformations

Like torchvision, vak provides a module for transformations of data that will become input to a neural network model or will be applied to the outputs of model, i.e., pre- and post-processing.

Standardization of spectrograms. A key transform that we provide for use during training is the `StandardizeSpect` class, that standardizes spectrograms so they are all on the same scale, by subtracting off a mean and dividing by a standard deviation (often called "normalization"). This transform is distinct from the normalization done by computer vision frameworks like torchvision, because it normalizes separately for each frequency bin in the spectrogram, doing so across all time bins. Using a scikit-learn-like API, this `StandardizeSpect` is fit to a set of spectrograms, such as the training set. The fit transform is saved during training as part of the results and then loaded automatically by vak for evaluation or when generating predictions for new data.

Transforms for frame labels. Many of the transforms we provide relate to what we call *frame labels*, that is, vectors where each element represents a label for a time bin from a spectrogram or a sample in an audio signal. These vectors of class labels are used as targets when training models in a supervised setting to perform frame classification.

The `from_segments` transform is used when loading annotations to produce a vector of labeled timebins from the segmented units, which are specified in terms of their onset and offset times along with their label.

Conversely, the `to_segments` takes a vector of labeled timebins and returns segments, by finding each continuous run of labels and then converting the onset and offsets from indices in the timebins vector to times in seconds. This post-processing transformation can be configured to perform additional clean-up steps: removing all segments shorter than a minimum duration, and taking a "majority vote" within each series of labels that are bordered by a "background" or "unlabeled" class.

In version 1.0, we have added the ability to evaluate models with and without the clean-up steps of the `to_segments` transform applied, so that a user can easily understand how the model is performing before and after these steps. This enhancement allows users to replicate a finding from [8], which showed, while the TweetyNet model achieved quite low segment error rates without post-processing, these simple clean-up steps allowed for significant further reduction of error. This finding was originally shown with an ad hoc analysis done with a script, but is now available directly through vak. This makes it easier for users to compare their model to a sort of empirical upper bound on performance, a strong baseline that indicates the "room for improvement" any given model has.

One more transformation worth highlighting here is the `to_labels` transformation, that converts a vector of labeled timebins directly to labels without recovering the onset or offset times. Essentially this transform consists of a `numpy.diff` operation, that we use to find the start of each run of continuous labels, and we then take the label at the start of each run. This transformation can be efficient when evaluating models where we want to measure just the segment error rate. (Of course we preclude the use of other metrics related to onset and offset times when throwing away that information, but for some research questions the main goal is to simply have the correct labels for each segment.)

Metrics

Vak additionally declares a `metrics` module for evaluation metrics that are specific to acoustic communication models. The main metric we have found it necessary to implement at this time is the (Levenshtein) string edit distance, and its normalized form, known in speech recognition as the word error rate. Our results have shown that edit distances such as this are crucial for evaluating frame classification models. We provide a well-tested implementation tailored for use with neural network models. In version 1.0 of vak, we have additionally adopted as a dependency the `torchmetrics` library, that makes it easier to compute a wide array of metrics for models.

Datasets

Lastly, vak provides a `dataset` module, again similar in spirit to the module of the same name in torchvision. Each family of models has its own dataset class or classes. We introduce these below, but first we describe our standardized dataset format.

Dataset directory format. In version 1.0 of vak we have adopted a standard for datasets that includes a directory structure and associated metadata. This addressed several limitations from version 0.x: datasets were not portable because of absolute paths, and certain expensive computations were done by other commands that should really have been done when preparing the dataset, such as validating the timebin size in spectrograms or generating multiple random subsets from a training set for learning curves. A listing that demonstrates the directory structure and some key contents is shown below.

```
dataset/
  train/
      song1.wav.npz
      song1.csv
      song2.wav.npz
      song2.csv
  val/
      song3.wav.npz
      song3.csv
  dataset.csv
  config.toml  # config used to generate dataset
  prep.log  # log from run of prep
  metadata.json  # any metadata
```

We can observe from the listing that, after collating files and separating them into splits as just described, the files are either moved (if we generated them) or copied (if a user supplied them) to directories corresponding to each split. For annotation formats where there is a one-to-one mapping from annotation file to the file that it annotates, we copy the annotation files to the split subdirectories as well. For annotation formats that place all annotations in a single file, we place this file in the root of the dataset directory. After moving these files, we change the paths in the

pandas dataframe representing the entire dataset so that they are written relative to the root of the directory. This makes the dataset portable. In addition to these split sub-directories containing the data itself, we note a few other files. These include a csv file containing the dataset files and the splits they belong to, whose format we describe next. They also include the `metadata.json` file that captures important parameters that do not fit well in the tabular data format of the csv file. For example, the metadata file for a frame classification dataset contains the duration of the timebin in every spectrogram. Finally, we note two other files in a dataset as shown above. The first is the configuration file used to generate it, copied into the dataset as another form of metadata. The second is a log file that captures any other data about choices made during dataset preparation, e.g., what files were omitted because they contained labels that were not specified in the labelset option of the configuration file.

Dataset csv file format. Next we outline the format of the csv file that represents a dataset. This csv (and the dataframe loaded from it) has four essential columns: `'audio_path'`, `'spect_path'`, `'annot_path'`, and `'split'`. These columns serve as provenance for the prepared dataset. Each row represents one sample in the dataset, where the meaning of sample may vary depending on the model family. For example, a sample for a frame classification model is typically an entire bout of vocalizations, whereas a sample for a Parametric UMAP model is typically a single unit from the bout. The csv format allows for tracing the provenance of each sample back to the source files used to generate the dataset. Each row must minimally contain either an `audio_path` or a `spectrogram_path`; if a user provides pre-computed spectrograms, the `audio_path` column is left empty. For models that use these files directly, the files will be copied into a sub-directory for each split, and the paths are written relative to the dataset root. The `'annot_path'` column points to annotation files. These again may be in the split sub-directories with the file that each annotates, or in the case of a single file will be in the root of the dataset directory, meaning that this single path will be repeated for every row in the csv. Logic in vak uses this fact to determine whether annotations can be loaded from a single file or must be loaded separately for each file when working with models.

Frame classification datasets

There are two generalized dataset classes for frame classification models in vak. Both these classes can operate on a single dataset prepared by the `vak prep` command; one class is used for training and the other for evaluation. We describe the workflow for preparing this dataset so that the difference between classes is clearer. The initial step is to pair data that will be the source of inputs x to a neural network model with the annotations that will be the source of training targets y for that model. This is done by collecting audio files or array files containing spectrograms from a "data directory", and then optionally pairing these files with annotation files. For models that take spectrograms as input, vak can use audio files to generate spectrograms that are then saved in array files and paired with any annotations. Alternatively a user can provide pre-computed spectrograms. This dataset can also be prepared without the targets y, for the case where a model is used to predict annotations for previously unseen data.

WindowDataset. This dataset class represents all possible time windows of a fixed width from a set of audio recordings or spectrograms. It is used for training frame classification models. Each call to `WindowDataset.__getitem__` with an `index` returns one window x from an audio signal or a spectrogram loaded into a tensor, along with the annotations that will be the target for the model y. Because this is a frame classification dataset, the annotations are converted during dataset preparation to vectors of frame labels, and y will be the window from this vector that corresponds to the window x. This is achieved by using a set of vectors to represent indices of valid windows from the total dataset, as described in detail in the docstring for the class. This use of a set of vectors to represent valid windows also enables training on a dataset of a specified duration without modifying the underlying data.

FramesDataset. As with the `WindowDataset`, every call to `FramesDataset.__getitem__` returns a single sample from the dataset. Here though, instead of a window, the sample will be the entire audio signal or spectrogram x and a corresponding vector of frame labels y. The default transforms used with this dataset apply additional pre-processing to the sample that facilitate evaluation. Specifically, the frames x and the frame labels y in a single sample are transformed to a batch of consecutive, non-overlapping windows. This is done by padding both x and y so their length is an integer multiple w of the window size used when training the model, and then returning a `view` of the sample as a stack of those w windows. Post-processing the output batch allows us to compute metrics on a per-sample basis, to answer questions such as "what is the average segment error rate per bout of vocalizations?".

Parametric UMAP datasets

For the parametric UMAP model, we provide a single dataset class, `ParametricUMAPDataset`. The underlying dataset consists of single units extracted from audio with a segmenting algorithm. The parameters of the dataset class configure the first step in the UMAP algorithm, that of building a graph on the dataset before embedding.

Command-line interface and configuration file

Having described the API, we now walk through vak's CLI. An example screenshot of a training run started from the command line is shown in Figure 2. A key design choice is to avoid any sub-commands or even options for the CLI, and instead move all such logic to a configuration file. Thus, commands through the CLI all take the form of `vak command configuration-file.toml`, e.g., `vak train gy6or6_train.toml`. This avoids the need for users to understand options and sub-commands, and minimizes the likelihood that important metadata about experiments will be lost because they were specified as options. The configuration file follows the TOML format (Tom's Obvious Minimal Language) that has been adopted by the Python and Rust communities among others.

The few commands available through the CLI correspond to built-in, model-specific workflows. There are five commands: `prep`, `train`, `eval`, `predict`, and `learncurve`. These commands are shown in 3 as part of a chart illustrating the built-in workflows, using as an example a frame classification model as we define them below. As their names suggest, the commands `train`, `eval`, and `predict` are used to train a model, evaluate it, and generate predictions with it once trained. The `prep` and `learncurve` commands require more explanation. A user makes a separate configuration file for each of the other four commands, but `prep` can be used with any configuration file. As can be seen

a

```
(.venv) $ vak train tests/data_for_tests/generated/configs/teenytweetynet_train_audio_wav_annot_birdsongrec.toml
2023-06-02 15:45:41,686 - vak.cli.train - INFO - vak version: 1.0.0a1
2023-06-02 15:45:41,687 - vak.cli.train - INFO - Logging results to tests/data_for_tests/generated/results/train/audio_wav_annot_birdsongrec/teenytweetynet/results_230602_154541
2023-06-02 15:45:41,688 - vak.core.train - INFO - Loading dataset from path: tests/data_for_tests/generated/prep/train/audio_wav_annot_birdsongrec/teenytweetynet/BirdO-vak-dataset-generated-230529_193550
2023-06-02 15:45:41,695 - vak.core.train - INFO - Size of timebin in spectrograms from dataset, in seconds: 0.002
2023-06-02 15:45:41,695 - vak.core.train - INFO - using training dataset from tests/data_for_tests/generated/prep/train/audio_wav_annot_birdsongrec/teenytweetynet/BirdO-vak-dataset-generated-230529_193550
2023-06-02 15:45:41,696 - vak.core.train - INFO - Total duration of training split from dataset (in s): 19.828
2023-06-02 15:45:41,977 - vak.core.train - INFO - number of classes in labelmap: 10
2023-06-02 15:45:41,977 - vak.core.train - INFO - no spect_scaler_path provided, not loading
2023-06-02 15:45:41,977 - vak.core.train - INFO - will normalize spectrograms
2023-06-02 15:45:42,205 - vak.core.train - INFO - Duration of WindowDataset used for training, in seconds: 19.828
```

b

```
Metric val_acc improved by 0.010 >= min_delta = 0.0. New best score: 0.877 3/3 [00:00<00:00, 24.77it/s]
Epoch 0:  33%|                                  | 848/2604 [00:10<00:21, 81.09it/s, loss=0.357, v_num=0E
poch 0, global step 800: 'val_acc' reached 0.87697 (best 0.87697), saving model to '/home/pimienta/Documents/repos/coding/vocalpy/vak-vocalpy/tests/data_for_tests/generated/results/train/audio_wav_annot_birdsongrec/teenytweetynet/results_230602_154419/TeenyTweetyNet/checkpoints/max-val-acc-checkpoint.pt' as top 1
Epoch 0:  35%|                                  | 901/2604 [00:11<00:20, 81.37it/s, loss=0.291, v_num=0]
Metric val_acc improved by 0.001 >= min_delta = 0.0. New best score: 0.878 3/3 [00:00<00:00, 35.16it/s]
Epoch 0:  35%|                                  | 901/2604 [00:11<00:20, 81.35it/s, loss=0.291, v_num=0E
poch 0, global step 850: 'val_acc' reached 0.87776 (best 0.87776), saving model to '/home/pimienta/Documents/repos/coding/vocalpy/vak-vocalpy/tests/data_for_tests/generated/results/train/audio_wav_annot_birdsongrec/teenytweetynet/results_230602_154419/TeenyTweetyNet/checkpoints/max-val-acc-checkpoint.pt' as top 1
Epoch 0:  37%|                                  | 954/2604 [00:11<00:20, 81.51it/s, loss=0.25, v_num=0E
poch 0, global step 900: 'val_acc' was not in top 1
Epoch 0:  39%|                                  | 1007/2604 [00:12<00:19, 81.76it/s, loss=0.25, v_num=0]
Metric val_acc improved by 0.014 >= min_delta = 0.0. New best score: 0.892 3/3 [00:00<00:00, 33.87it/s]
Epoch 0:  39%|                                  | 1007/2604 [00:12<00:19, 81.75it/s, loss=0.25, v_num=0E
poch 0, global step 950: 'val_acc' reached 0.89199 (best 0.89199), saving model to '/home/pimienta/Documents/repos/coding/vocalpy/vak-vocalpy/tests/data_for_tests/generated/results/train/audio_wav_annot_birdsongrec/teenytweetynet/results_230602_154419/TeenyTweetyNet/checkpoints/max-val-acc-checkpoint.pt' as top 1
Epoch 0:  41%|                                  | 1060/2604 [00:12<00:18, 81.88it/s, loss=0.296, v_num=0E
poch 0, global step 1000: 'val_acc' was not in top 1
Epoch 0:  42%|                                  | 1091/2604 [00:13<00:18, 82.72it/s, loss=0.248, v_num=0]
```

Fig. 2: *Screenshots of vak, demonstrating the command-line interface and logging. In top panel (a), an example is shown of using the command-line interface to train a model with a configuration file. In the bottom panel (b) an example is shown of how vak logs progress and reports metrics during training*

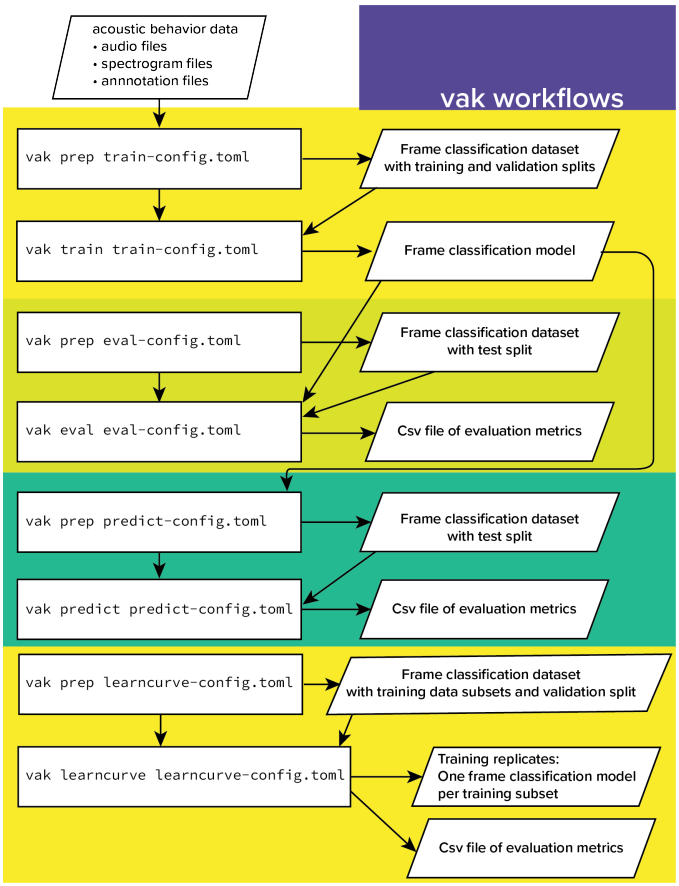

Fig. 3: *A chart showing workflows in vak, using an example a frame classification model as defined below. See text for description of workflows.*

in the figure, the typical workflow starts with a call to `vak prep`, which prepares a canonicalized form of a dataset for the specific machine learning task associated with a model, and then adds the path to that dataset to the configuration file. Thus, there is a `prep_frame_classification_dataset` function that will be called for the example model in the figure. If a dataset has already been prepared and is being re-used for another experiment, this step would not be necessary. Once any needed dataset is prepared, the user can run the command related to the model, using the same configuration file.

The `learncurve` command is used to generate results for a learning curve, that plots model performance as a function of training set size in seconds. Although technically a learning curve, its use is distinct from common uses in machine learning, e.g., looking for evidence of high bias or high variance models. Instead, the learning curve functionality allows vak users to answer important practical questions for their research. Most importantly, what is the optimal performance that can be achieved with the minimum amount of labor-intensive, hand-annotated training data?

Results

In this section we present proof-of-concept results demonstrating the utility of our framework. The project that produced these results can be found at: https://github.com/vocalpy/scipy-proceedings-2023-vak

Ablation experiment

We first show how vak allows researchers to experiment with a model not built into the library. For this purpose, we carry out an "ablation experiment" as the term is used in the artificial neural network literature, where an operation is removed from a neural network function to show that operation plays an important role in the model's performance. Using a script, we define a version of the TweetyNet model in [8] without the recurrent Long Short Term Memory (LSTM) layer (thus "ablating" it). This model without the LSTM makes a prediction for each frame using the output of the convolutional layers, instead of using the hidden state of the recurrent layer at each time step. If the hidden state contains features that are useful for predicting across time steps, we would expect that "ablating" (removing) it would impair performance. To show that removing the LSTM layer impairs performance, we compare with the full TweetyNet model (now built into vak). For all experiments, we prepared a single dataset and then trained both models on that same dataset. We specifically ran learning curves as described above, but here we consider only the performance using 10 minutes of data for training, because as we previously reported [8] this was the minimum amount of training data required to achieve the lowest error rates. As shown in the top row of Figure 4, ablating the recurrent layer increased the frame error rate (left column, right group of bars), and this produced an inflated syllable error rate (right column, right group of bars).

This first result is the average across models trained on datasets prepared from individual birds in the Bengalese finch song repository dataset [36], as we did previously in [8]. (There are four birds, and five training replicates per bird, where each replicate is trained on different subsets from a larger pool of training data.) Other studies using the same benchmark data repository have trained models on datasets prepared from all four birds [10] (so that the model predicts 37 classes, the syllables from all four birds, instead of 5-10 per bird). We provide this result for the TweetyNet model with and without LSTM in the bottom row of Figure 4. It

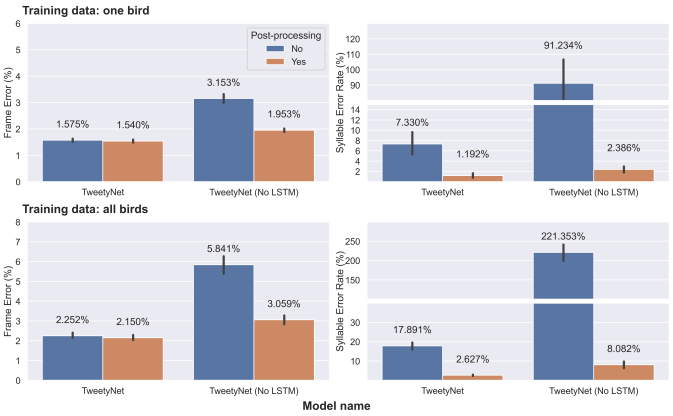

Fig. 4: *Ablation experiment carried out by declaring a model in a script using the vak framework. Bar plots show frame error (left column) and syllable error rate (right column), without post-processing clean-up (blue bars) and with (orange bars). Within each axes, the grouped bars on the left indicate results from the TweetyNet model built into the vak library, and the grouped bars on the right indicate results from a model declared in a script where the recurrent LSTM layer has been removed ("ablated") from the TweetyNet architecture. In the top row, values are the average across models trained on data from four different Bengalese finches, with five training replicates per bird (see text for detail). In the bottom row, single models were trained to classify syllables from all four birds.*

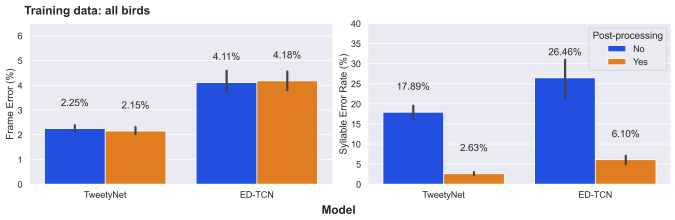

Fig. 5: *Comparison of TweetyNet model [8] with ED-TCN model. Plots are as in 4. Each axes shows results for one individual bird from the Bengalese finch song repository dataset [36]. Bar plots show frame error (left column) and syllable error rate (right column), without post-processing clean-up (blue bars) and with (orange bars).*

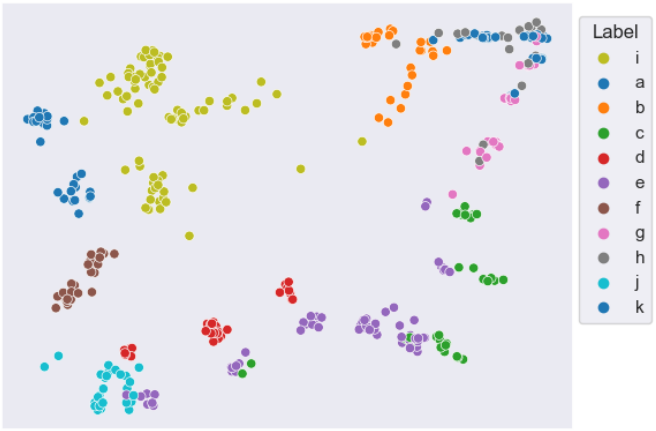

Fig. 6: *Scatter plot showing syllables from the song of one Bengalese finch, embeeded in a 2-D space using a convolutional encoder trained using the Parametric UMAP algorithm. Each marker is a point produced from a spectrograms of a single syllable rendition, mapped down to the 2-D space, from 40 seconds of training data. Colors indicate the label applied to each syllable by an expert human when annotating the spectrograms with a GUI.*

can be seen that asking the models to predict a greater number of classes further magnified the difference between them (as would be expected). TweetyNet without the LSTM layer has a syllable error rate greater than 230%. (Because the syllable error rate is an edit distance, it can be greater than 1.0. It is typically written as a percentage for readability of smaller values.)

Comparison of TweetyNet and ED-TCN

We next show how vak allows researchers to compare models. For this we compare the TweetyNet model in [8] with the ED-TCN model of [33]. As for the ablation experiment, we ran full learning curves, but here just focus on the performance of models trained on 10 minutes of data. Likewise, the grouped box plots are as in Figure 4, with performance of TweetyNet again on the left and in this case the ED-TCN model on the right. Here we only show performance of models trained on data from all four birds (the same dataset we prepared for the ablation experiment above). We observed that on this dataset the ED-TCN had a higher frame error and syllable error rate, as shown in Figure 5. However, there was no clear difference when training models on individual birds (results not shown because of limited space). Our goal here is not to make any strong claim about either model, but simply to show that our framework makes it possible to more easily compare two models on the exact same dataset.

Applying Parametric UMAP to Bengalese finch syllables with a convolutional encoder

Finally we provide a result demonstrating that a researcher can apply multiple families of models to their data with our framework. As stated above, the vak framework includes an implementation of a Parametric UMAP family, and one model in this family, a simple encoder network with convolutional layers on the front end. To demonstrate this model, we train it on the song of an individual bird from the Bengalese finch song repository. We use a training set with a duration of 40 seconds total, containing clips of all syllable classes from the bird's song, taken from songs that were drawn at random from a larger data pool by the vak dataset preparation function. We then embed a separate test set. It can be seen in Figure 6 that points that are close to each other are almost always the same color, indicating that syllables that were given the same label by a human annotator are also nearer to each other after mapping to 2-D space with the trained parametric UMAP model.

Discussion

Researchers studying acoustic behavior need to benchmark multiple neural network models on their data, evaluate training performance for different training set sizes, and use trained models to make predictions on newly acquired data. Here we presented vak, a neural network framework developed to meet these needs. In the Methods we described its design and development. Then in the Results we provide proof-of-concept results demonstrating how researchers can easily use our framework.

Finally, we summarize the roadmap for further development of version 1.0 of vak. In the spirit of taking an open approach, we are tracking issues related to this roadmap on GitHub: https://github.com/vocalpy/vak/issues/614. A key goal will be to add benchmark datasets, generated by running the vak prep command, that a user can download and use to benchmark models

with publicly shared configuration files. Another key goal will be to add models that are pre-trained on these benchmark datasets. Additionally we plan to refactor the prep module to make use of the vocalpy package [37], developed to make acoustic communication research code in Python more concise and readable. Another key step will be inclusion of additional models like those reviewed in the Related Work. Along with this expansion of existing functionality, the final release of version 1.0 will include several quality-of-life improvements, including a revised schema for the configuration file format that better leverages the strengths of TOML, and dataclasses that represent outputs of vak, such as dataset directories and results directories, to make it easier to work with outputs programmatically. It is our hope that these conveniences plus the expanded models and datasets will provide a framework that can be developed collaboratively by the entire research community studying acoustic communication in animals.

References

[1] M. D. Hauser, N. Chomsky, and W. T. Fitch, "The Faculty of Language: What Is It, Who Has It, and How Did It Evolve?" *Science*, vol. 298, no. 5598, pp. 1569–1579, Nov. 2002, https://doi.org/10.1126/science.298.5598.1569.

[2] A. J. Doupe and P. K. Kuhl, "BIRDSONG AND HUMAN SPEECH: Common Themes and Mechanisms," *Annual Review of Neuroscience*, vol. 22, no. 1, pp. 567–631, Mar. 1999, https://doi.org/10.1146/annurev.neuro.22.1.567.

[3] M. Wirthlin, E. F. Chang, M. Knörnschild, L. A. Krubitzer, C. V. Mello, C. T. Miller, A. R. Pfenning, S. C. Vernes, O. Tchernichovski, and M. M. Yartsev, "A Modular Approach to Vocal Learning: Disentangling the Diversity of a Complex Behavioral Trait," *Neuron*, vol. 104, no. 1, pp. 87–99, Oct. 2019, https://doi.org/10.1016/j.neuron.2019.09.036.

[4] S. L. Hopp, M. J. Owren, and C. S. Evans, *Animal Acoustic Communication: Sound Analysis and Research Methods*. Springer Science & Business Media, 2012.

[5] T. Sainburg and T. Q. Gentner, "Toward a Computational Neuroethology of Vocal Communication: From Bioacoustics to Neurophysiology, Emerging Tools and Future Directions," *Frontiers in Behavioral Neuroscience*, vol. 15, p. 811737, Dec. 2021, https://doi.org/10.3389/fnbeh.2021.811737.

[6] D. Stowell, "Computational bioacoustics with deep learning: A review and roadmap," p. 46, 2022.

[7] Y. Cohen, T. A. Engel, C. Langdon, G. W. Lindsay, T. Ott, M. A. Peters, J. M. Shine, V. Breton-Provencher, and S. Ramaswamy, "Recent advances at the interface of neuroscience and artificial neural networks," *Journal of Neuroscience*, vol. 42, no. 45, pp. 8514–8523, 2022.

[8] Y. Cohen, D. A. Nicholson, A. Sanchioni, E. K. Mallaber, V. Skidanova, and T. J. Gardner, "Automated annotation of birdsong with a neural network that segments spectrograms," *Elife*, vol. 11, p. e63853, 2022.

[9] J. Goffinet, S. Brudner, R. Mooney, and J. Pearson, "Low-dimensional learned feature spaces quantify individual and group differences in vocal repertoires," *eLife*, vol. 10, p. e67855, May 2021, https://doi.org/10.7554/eLife.67855.

[10] E. Steinfath, A. Palacios-Muñoz, J. R. Rottschäfer, D. Yuezak, and J. Clemens, "Fast and accurate annotation of acoustic signals with deep neural networks," *eLife*, vol. 10, p. e68837, Nov. 2021, https://doi.org/10.7554/eLife.68837.

[11] T. Sainburg, M. Thielk, and T. Q. Gentner, "Finding, visualizing, and quantifying latent structure across diverse animal vocal repertoires," *PLOS Computational Biology*, vol. 16, no. 10, p. e1008228, Oct. 2020, https://doi.org/10.1371/journal.pcbi.1008228.

[12] K. R. Coffey, R. E. Marx, and J. F. Neumaier, "DeepSqueak: A deep learning-based system for detection and analysis of ultrasonic vocalizations," *Neuropsychopharmacology*, vol. 44, no. 5, pp. 859–868, Apr. 2019, https://doi.org/10.1038/s41386-018-0303-6.

[13] D. Nicholson and Y. Cohen, "Vak," Zenodo, Mar. 2022, https://doi.org/10.5281/zenodo.6808839.

[14] J. N. McGregor, A. L. Grassler, P. I. Jaffe, A. L. Jacob, M. S. Brainard, and S. J. Sober, "Shared mechanisms of auditory and non-auditory vocal learning in the songbird brain," *eLife*, vol. 11, p. e75691, Sep. 2022, https://doi.org/10.7554/eLife.75691.

[15] K. L. Provost, J. Yang, and B. C. Carstens, "The impacts of fine-tuning, phylogenetic distance, and sample size on big-data bioacoustics," *PLOS ONE*, vol. 17, no. 12, p. e0278522, Dec. 2022, https://doi.org/10.1371/journal.pone.0278522.

[16] A. Kershenbaum, D. T. Blumstein, M. A. Roch, Ç. Akçay, G. Backus, M. A. Bee, K. Bohn, Y. Cao, G. Carter, C. Cäsar, M. Coen, S. L. DeRuiter, L. Doyle, S. Edelman, R. Ferrer-i-Cancho, T. M. Freeberg, E. C. Garland, M. Gustison, H. E. Harley, C. Huetz, M. Hughes, J. Hyland Bruno, A. Ilany, D. Z. Jin, M. Johnson, C. Ju, J. Karnowski, B. Lohr, M. B. Manser, B. McCowan, E. Mercado, P. M. Narins, A. Piel, M. Rice, R. Salmi, K. Sasahara, L. Sayigh, Y. Shiu, C. Taylor, E. E. Vallejo, S. Waller, and V. Zamora-Gutierrez, "Acoustic sequences in non-human animals: A tutorial review and prospectus: Acoustic sequences in animals," *Biological Reviews*, vol. 91, no. 1, pp. 13–52, Feb. 2016, https://doi.org/10.1111/brv.12160.

[17] Y. Fukuzawa, "Computational methods for a generalised acoustics analysis workflow: A thesis presented in partial fulfilment of the requirements for the degree of Master of Science in Computer Science at Massey University, Auckland, New Zealand," Ph.D. dissertation, Massey University, 2022.

[18] A. Graves and J. Schmidhuber, "Framewise phoneme classification with bidirectional LSTM and other neural network architectures," *Neural networks*, vol. 18, no. 5-6, pp. 602–610, 2005, https://doi.org/10.1016/j.neunet.2005.06.042.

[19] A. Graves, "Supervised sequence labelling," in *Supervised Sequence Labelling with Recurrent Neural Networks*. Springer, 2012, pp. 5–13, https://doi.org/10.1007/978-3-642-24797-2_2.

[20] E. Steinfath, A. Palacios, J. Rottschäfer, D. Yuezak, and J. Clemens, "Fast and accurate annotation of acoustic signals with deep neural networks," p. 30.

[21] L. Zandberg, V. Morfi, J. George, D. F. Clayton, D. Stowell, and R. F. Lachlan, "Bird song comparison using deep learning trained from avian perceptual judgments," Animal Behavior and Cognition, Preprint, Dec. 2022, https://doi.org/10.1101/2022.12.23.521425.

[22] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in PyTorch," Oct. 2017.

[23] W. Falcon and T. P. L. team, "PyTorch Lightning," Zenodo, Apr. 2023, https://doi.org/10.5281/zenodo.7859091.

[24] S. van der Walt, S. C. Colbert, and G. Varoquaux, "The NumPy Array: A Structure for Efficient Numerical Computation," *Computing in Science Engineering*, vol. 13, no. 2, pp. 22–30, Mar. 2011, https://doi.org/10.1109/MCSE.2011.37.

[25] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith *et al.*, "Array programming with NumPy," *Nature*, vol. 585, no. 7825, pp. 357–362, 2020.

[26] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. J. van der Walt, M. Brett, J. Wilson, K. J. Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. J. Carey, İ. Polat, Y. Feng, E. W. Moore, J. VanderPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, P. van Mulbregt, and S. . . Contributors, "SciPy 1.0–Fundamental Algorithms for Scientific Computing in Python," *arXiv:1907.10121 [physics]*, Jul. 2019.

[27] J. D. Hunter, "Matplotlib: A 2D graphics environment," *Computing in Science & Engineering*, vol. 9, no. 3, pp. 90–95, 2007, https://doi.org/10.1109/MCSE.2007.55.

[28] T. A. Caswell, M. Droettboom, A. Lee, J. Hunter, E. S. de Andrade, E. Firing, T. Hoffmann, J. Klymak, D. Stansby, N. Varoquaux, J. H. Nielsen, B. Root, R. May, P. Elson, J. K. Seppänen, D. Dale, J.-J. Lee, D. McDougall, A. Straw, P. Hobson, C. Gohlke, T. S. Yu, E. Ma, A. F. Vincent, S. Silvester, C. Moad, N. Kniazev, hannah, E. Ernest, and P. Ivanov, "Matplotlib/matplotlib: REL: V3.3.2," Zenodo, Sep. 2020, https://doi.org/10.5281/zenodo.4030140.

[29] T. pandas development team, "Pandas-dev/pandas: Pandas," Feb. 2020, https://doi.org/10.5281/zenodo.3509134.

[30] Dask Development Team, *Dask: Library for Dynamic Task Scheduling*, 2016.

[31] D. Nicholson, "Crowsetta: A python tool to work with any format for annotating animal vocalizations and bioacoustics data." *Journal of Open Source Software*, vol. 8, no. 84, p. 5338, 2023, https://doi.org/10.21105/joss.05338.

[32] T. maintainers and contributors, "TorchVision: PyTorch's computer vision library," GitHub, 2016.

[33] C. Lea, M. D. Flynn, R. Vidal, A. Reiter, and G. D. Hager, "Temporal convolutional networks for action segmentation and detection," in *proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2017, pp. 156–165, https://doi.org/10.1109/cvpr.2017.113.

[34] L. McInnes, J. Healy, and J. Melville, "Umap: Uniform manifold approximation and projection for dimension reduction," *arXiv preprint arXiv:1802.03426*, 2018.

[35] T. Sainburg, L. McInnes, and T. Q. Gentner, "Parametric umap embeddings for representation and semisupervised learning," *Neural Computation*, vol. 33, no. 11, pp. 2881–2907, 2021, https://doi.org/10.1162/neco_a_01434.

[36] D. Nicholson, J. E. Queen, and S. J. Sober, "Bengalese Finch song repository," Oct. 2017, https://doi.org/10.6084/m9.figshare.4805749.v5.

[37] D. Nicholson, "vocalpy/vocalpy: 0.3.0," May 2023, https://doi.org/10.5281/zenodo.7925888. [Online]. Available: https://zenodo.org/record/7925888

Emukit: A Python toolkit for decision making under uncertainty

Andrei Paleyes[‡*], Maren Mahsereci[§], Neil D. Lawrence[‡]

Abstract—Emukit is a highly flexible Python toolkit for enriching decision making under uncertainty with statistical emulation. It is particularly pertinent to complex processes and simulations where data are scarce or difficult to acquire. Emukit provides a common framework for a range of iterative methods that propagate well-calibrated uncertainty estimates within a design loop, such as Bayesian optimisation, Bayesian quadrature and experimental design. It also provides multi-fidelity modelling capabilities. We describe the software design of the package, illustrate usage of the main APIs, and showcase the breadth of use cases in which the library already has been used by the research community.

Index Terms—statistical emulation, software, Bayesian optimisation, Bayesian quadrature, Bayesian experimental design, multi-fidelity, active learning

Introduction

Data selection is a major challenge in supervised machine learning (ML). Quite often when data availability is not an issue, data collection occurs prior to the training process and results in a static dataset, meaning that the machine learning model has no influence on the data collection process. However, if data points are expensive and scarce the performance of a model trained on a static dataset can be suboptimal or even poor. In those cases, it is beneficial to carefully select the dataset such that, for example, it is maximally informative under the ML model to achieve the task at hand. This branch of ML is generally referred to as *active learning* [1] and has attracted attention in various sub-fields such as *experimental design* (the task of predicting an unknown function value from its input), *global optimisation* (guessing the global minimiser of a function) and *integration* (guessing the integral of a function). The Emukit Python library, at core, augments existing machine learning models with active data selection functionality.

Tasks where data acquisition is hard usually involve a higher degree of expert knowledge on the modeling side, because incorporating prior information, such as mechanical or physical knowledge about the system under study, aims for more physically meaningful and accurate predictions on the task at hand. Often these models are of probabilistic nature and can provide a degree of uncertainty of their prediction to counteract the lack of data [2], [3], [4], [5]. Such a model is often referred to as a *statistical emulator*[1], which is a machine learning model that can replace an expensive computer simulation (a *simulator*) or real world experiment, and is trained on input-output pairs of the latter [6], [7], [8], [9]. Concretely, the simulator could be an involved stochastic weather simulation, and the emulator a predictive machine learning model trained on expensive input-output pairs of the weather simulation. Alternatively, the emulator may model a real world process. The emulator now may replace the original data source to obtain fast predictions when needed, or to compute auxiliary quantities that cannot be obtained from the data source.

Once trained, the performance of the emulator (the ML model), as mentioned, depends on the informativity of the data (the simulation results), especially if it is expensive and scarce, and hence active data selection is often desirable for such models. A unique feature of Emukit is that it enables the user to wrap custom emulator models into an interface provided by Emukit and, by doing so, use them in Emukit's decision loop. As such, Emukit 'actifies' (makes active) the data-acquisition of custom models written in custom backends that only connect via an interface to Emukit. This may i) save users time and money to write their own active learning loop, ii) or to rewrite their custom model in existing decision loop packages with a fixed backend, and iii) improve performance of the model with more informative training data.

Hence, the most prominent features of Emukit can be summarized as follows.

- Emukit *augments existing models with active learning* capability, in particular models used in Bayesian optimisation, Bayesian quadrature and experimental design.
- Emukit can use existing, potentially specialized, custom models provided by the user and wrap them into a provided interface. As such Emukit is *model backend agnostic*.
- Emukit is highly abstracted and mimics the components of an active decision loop. This composability allows users to provide custom implementations of subroutines and classes that seamlessly integrate with the rest of the package. Hence Emukit is *highly flexible* and allows *fast and easy prototyping*.
- In contrast to other packages, Emukit provides several active learning methods via *subpackages that share a core*

* Corresponding author: ap2169@cam.ac.uk
‡ Department of Computer Science and Technology, University of Cambridge
§ University of Tübingen

1. The reader might be familiar with the term 'emulator' in computing context, where it refers to a hardware or software that makes one system behave like another. The '(statistical) emulator' we use throughout this paper is an unrelated, albeit similar, term from the machine learning literature.

implementation of the active learning loop. This enables the user to potentially use the same model backend and even the same model instance across tasks. This increases consistency between results, may reduce implementation overhead and allow resource sharing between tasks.

- Emukit provides basic functionality for *multi-fidelity modeling* which allows the user to incorporate data sources of different fidelities. Further, Emukit contains a limited number of model wrappers to illustrate their usage and some example applications.

The remainder of the paper corroborates the points above in greater detail. The following section briefly introduces the supported machine learning methods before sketching Emukit's workflow and library structure. Throughout the text, we refer to 'tasks' in an abstract sense, without a specific application in mind. In the remainder of text, we will use the terms 'ML model' and 'emulator' interchangeably. The 'simulator' or 'data source' will later also be referred to as 'user function', 'black-box function' or 'objective function'.

Background on probabilistic active methods

This section gives an overview of the machine learning methods provided by Emukit. Emukit mainly contains three high-level methods: Bayesian optimisation (BO), Bayesian quadrature (BQ) and experimental design (ED).

Bayesian optimisation [10], [11] is a numerical method that aims to guess the global minimiser of a black-box function by querying function values at nodes and returning the minimiser of the collected set. Corresponding algorithms are inherently sequential and, at every iteration, decide on where to query the objective function next. The decision solves the so called 'exploration-exploitation trade-off' between exploring unknown regions of the function's domain, or exploiting rather promising regions of a potential minimiser. This trade-off is encoded in a heuristic called 'acquisition function' that quantifies the usefulness of evaluating the function at a certain node. Hence, BO is generally sample-efficient and thus especially useful when the function is expensive to evaluate and the number of allowed evaluations is limited. There exists a large range of heuristics and methods that all fall under the umbrella of BO, out of which Emukit supports several. Bayesian optimisation has been successfully applied in various fields [12], [13], but most notably in the automation of hyperparameter tuning tasks of neural networks [14], [15].

Bayesian quadrature [16], [17], [18] is a numerical method that aims to infer the integral of a black-box function (called the 'integrand') given some integration measure and queries of the integrand at nodes. In contrast to Monte Carlo (MC) methods, BQ generally accept any kind of node design and is especially sample efficient which makes it superior to MC in certain circumstances [19]. A sub-group of BQ methods are active and follow a similar decision loop as BO; the most notable difference being that acquisition functions are specific to BQ and the class of models is somewhat more restricted. Generally, active BQ methods are algorithmically similar on a high level to BO methods and can use similar models.

Experimental design [6], [7], [8], [9], also known as Bayesian active learning, is a method that aims to collect data about a black-box function such that the resulting probabilistic model predicts unseen function values well. Unlike BO and BQ discussed above, ED aims to learn the objective function as well as possible across the entire input space. It traditionally has been applied to statistical emulation of complex computer models but also has found applications in healthcare [20], computational biology [21] and engineering [22]. Some ED methods also obey the structure of an active decision loop similar to BO and BQ.

Emukit embodies the realization that all three methods (BO, BQ, ED), albeit having different numerical aims, share the same algorithmic structure: a decision loop that computes the next node based on the current model, evaluates the user function, and then updates the model accordingly. This decision loop is contained in Emukit's core package and shared by all three high-level methods. Furthermore, especially BO and ED may use similar models while BQ is somewhat more restricted. Potential benefits of sharing implementation, model and compute between tasks are discussed in later sections.

Finally, Emukit provides basic support of *multi-fidelity models* [23] which can combine query results of the black-box function of different quality (from low fidelity to high fidelity). This yields multi-fidelity BO, BQ and ED methods that may be made active again with Emukit's decision loop.

Emukit workflow

Decision making with statistical emulation consists of three parts. All starts with a *task*, a high level goal that we are interested in achieving. It usually involves a complex process that we aim to study to answer a question. Some examples include finding the best operation mode of a drone, measuring the quality of a weather simulation, explaining behavior of a complex system. In order to solve the task we choose a *method*, a relatively low-level technique that guides our exploration of the target process and provides the quantifiable way to answer the task's question. Examples include Bayesian optimisation, Bayesian quadrature and experimental design. And finally there is a *model*, a probabilistic data-driven representation of the process under study. Examples of such models are a Gaussian process, a random forest or a Bayesian network. Consequently, the typical workflow for users working with Emukit consists of three steps (see Figure 1 for a graphical description).

Build the model. Instead of constraining the user to certain model classes, Emukit provides the flexibility of using user-specified models. Generally speaking, Emukit does not provide modeling capabilities. Instead users are expected to define their own models. Because of the variety of modeling frameworks available, Emukit does not mandate or make any assumptions about a particular modeling technique or a library, and suggests implementing a subset of defined model interfaces that are required to use a particular method.

Run the method. This is the main focus of Emukit. Emukit defines a general structure of a decision making method and offers implementations of several such methods: Bayesian optimisation, Bayesian quadrature, experimental design. All methods are model-agnostic and only rely on model interfaces.

Solve the task. For the end users, Emukit is a way to solve a certain task, which may have research or business value. Emukit provides a set of examples of how tasks such as hyperparameter tuning, sensitivity analysis, multi-fidelity modeling or benchmarking are accomplished using the library.

Structure of the library

At a conceptual level the methods supported in Emukit – such as Bayesian optimisation, experimental design and Bayesian quadra-

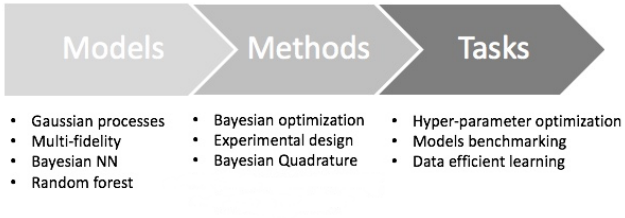

Fig. 1: *Summary of workflow for the users of Emukit. The user chooses a a modeling framework and defines a model. The model is wrapped using a pre-defined interface and connected to the core components of several methods such as Bayesian optimisation, experimental design etc. Specific tasks are then solved using these methods.*

ture – are all iterative decision making processes that follow a similar pattern. Algorithmically they can be thought of as instances of a common abstract loop, which we now describe (also see Algorithm 1).

The common goal of all of these methods is to learn a behavior of an *objective function* - a black-box expensive process that has certain *parameters*. The knowledge about the objective function (initially available as well as that collected during the learning process) is represented with a *probabilistic model*. New data points are proposed by optimising an *acquisition function* constructed using the model. Finally, the decision making process is done in a *loop* until a certain *stopping condition* is met.

Algorithm 1 Decision making loop in Emukit.

1: **while** stopping condition is not met **do**
2: collect next point(s) for evaluation
3: evaluate objective function
4: update model with new observation(s)
5: **end while**

The internal structure of Emukit reflects these abstractions to enable swapping and replacement of fundamental components of the decision making loop. While some of the basic components in Emukit correspond to the parts of the decision making loop exactly, others are more fine-grained to allow for greater flexibility and plug-and-play experience for the researchers using the package. We will now give an overview of these components.

Outer Loop. The `OuterLoop` class is the abstract loop where the different components come together. Loops for specific methods, such as Bayesian optimisation and experiment design, should subclass it. The library provides several concrete implementations of the loop, and also contains examples how the users may build their own.

Parameter space. Represents the parameter space of the objective function, also referred to as input space. Emukit supports continuous, categorical, discrete, and bandit parameters.

Model. All Emukit loops need a probabilistic model. Emukit does not provide functionality to build models as there are already good modeling frameworks available in Python. Instead, it provides a way of interfacing third-party modeling libraries. The interfacing mechanism consists of two parts: interfaces and wrappers. *Interfaces* define functionality required from a model. Different models and modeling frameworks will provide different functionality. For instance a Gaussian process will usually have derivatives of the predictions available but random forests will not. A model implements a set of interfaces that represent these different functionalities. The basic interface that all models must implement is `IModel`, which implements functionality to make predictions and update the model but a model may implement any number of other interfaces such as `IDifferentiable` which indicates a model has prediction derivatives available. Other components of the decision making loop may also be define interfaces to indicate that they require a certain functionality from the model. For example, `ICalculateVarianceReduction` defines methods the user needs to implement with their model to use it with the variance reduction technique. `Model wrappers` adapt third-party models and implement one or more of the interfaces using specific modeling framework. Emukit provides a wrapper for using a model created with `GPy` [24].

Candidate Point Calculator. This entity drives the decision on which point(s) to evaluate next. The simplest implementation provided out of the box, `SequentialPointCalculator`, collects one point at a time by finding where the acquisition is at a maximum by applying the acquisition optimiser to the acquisition function. More complex implementations are possible, for example to enable batches of points to be collected so that the user function may be evaluated in parallel.

Acquisition. The acquisition is a function defined on the parameter space that produces continuous values. It represents a heuristic quantification of how valuable collecting a future point might be, and produces continuous values. It is used by the candidate point calculator to decide which point(s) to collect next. Acquisition functions balance exploration and exploitation of the decision making process.

Acquisition Optimiser. The `AcquisitionOptimizer` optimises the acquisition function to find the point at which the acquisition is at a maximum. If available, the optimiser can use the acquisition function gradients. Otherwise, it will either estimate the gradients numerically or use a gradient free optimisation.

User Function. This is the component that represents the objective function. It can be evaluated by the user or it can be passed into the loop and evaluated by Emukit.

Model Updater. The `ModelUpdater` class updates the model with new training data after a new point is observed and optimises any hyperparameters of the model. It can decide whether hyperparameters need updating based on some internal logic.

Stopping Condition. The `StoppingCondition` class chooses when the decision making loop should stop collecting points. The most commonly used approach is to stop when a set number of iterations has been reached.

These are the core components Emukit defines. Specific methods may also define additional concepts of their own, e.g. integration measures or costs. Table 1 shows the mapping between decision making abstractions and Emukit components.

USAGE OVERVIEW

This section describes Emukit's high level APIs for all main functions of the package: Bayesian optimisation, Bayesian quadrature, experimental design and multi-fidelity emulation. Unless stated otherwise, we assume that some initial data (an initial design of reasonable size with corresponding evaluations of the user function) are already defined and stored in the variables `X` (inputs) and `Y` (values). We use GPy [24] in the code snippets below for modeling, and exclude import lines for brevity.

Decision making abstractions	Emukit components
Loop	Outer loop
Parameters	Parameter space
Probabilistic model	Model interface
	Model wrapper
Acquisition function	Candidate point calculator
	Acquisition
	Acquisition optimiser
Objective function	User function
	Model updater
Stopping Condition	Stopping condition

TABLE 1: *The mapping between abstractions of the decision making process and the components defined in Emukit.*

Standard methods and model wrapping

Interfaces for Bayesian optimisation and experimental design are the most straighforward ways to use the library. Both methods require the user to define a model and wrap it in the Emukit's model wrapper. An input space also has to be defined using Emukit's classes. The choice of acquisition function is optional, as reasonable defaults are provided. High level loop objects allow the user to execute the decision making loop and access its properties.

```
model_gpy = GPy.models.GPRegression(X,Y)
model_emukit = GPyModelWrapper(model_gpy)

parameter_space = ParameterSpace([
    ContinuousParameter('x1', -5, 10),
    ContinuousParameter('x2', 0, 15)
])

expected_improvement_acquisition =
    ExpectedImprovement(model = model_emukit)
bayesopt_loop = BayesianOptimizationLoop(
    model = model_emukit,
    space = parameter_space,
    acquisition = expected_improvement_acquisition
)

model_variance_acquisition =
    ModelVariance(model = model_emukit)
experimental_design_loop =
    ExperimentalDesignLoop(
        model = model_emukit,
        space = parameter_space,
        acquisition = model_variance_acquisition
    )
```

Usage of Bayesian quadrature (BQ) API is more involved, as even in its most basic form it requires more choices from the user. First the objective function, also referred to as an integrand, is modeled with a Gaussian process (GP). Since BQ integrates the kernel function, the kernel is then wrapped in a separate Emukit object. Bundled together, wrappers around the kernel and the model itself represent a base model in the BQ package. This model may be used with several BQ methods, the code below illustrates vanilla Bayesian quadrature where the GP model is directly placed over the integrand function and then integrated analytically.

```
lb = -3.0 # lower integral bound
ub = 3.0 # upper integral bound
gpy_model = GPy.models.GPRegression(X=X, Y=Y)

emukit_rbf = RBFGPy(gpy_model.kern)
emukit_measure = LebesgueMeasure.from_bounds(
    bounds=[(lb, ub)]
)
emukit_qrbf = QuadratureRBFLebesgueMeasure(
    emukit_rbf, emukit_measure
)
gp_model = BaseGaussianProcessGPy(
    kern=emukit_qrbf, gpy_model=gpy_model
)

emukit_model = VanillaBayesianQuadrature(
    base_gp=gp_model, X=X, Y=Y
)
bq_loop = VanillaBayesianQuadratureLoop(
    model=emukit_model
)
```

Once the loop object is created, either for optimisation, quadrature or experiment design, it may be evaluated in one of two modes. If the user has access to the objective function via Python, Emukit can manage the loop with the `run_loop` method that accepts two arguments: the objective function and the stopping criterion. If the objective has to be called externally (e.g. a lab experiment has to be done), Emukit provides `get_next_points` method that produces the next evaluation point(s) based on the data observed so far. In that latter case user has to manage the decision making loop themselves.

Interfaces for fast prototyping

Emukit gives researchers a lot of flexibility in swapping individual pieces in and out of the decision making loop. This is made possible by clearly defined interfaces. We illustrate how this is accomplished in the package with an example of `IntegratedHyperParameterAcquisition`. This class provides an ability to integrate any acquisition function over hyperparameters of the model. To do that, the model needs to support two operations: generate hyperparameter samples and fix hyperparameters to a certain sample value. Consequently, Emukit defines an interface `IPriorHyperparameters` that declares these operations, and `IntegratedHyperParameterAcquisition` requires input model to implement this interfaces, as is shown in the following code snippet:

```
class IPriorHyperparameters:
    def generate_hyperparameters_samples(...

    def fix_model_hyperparameters(...


class IntegratedHyperParameterAcquisition(Acquisition):
    def __init__(
        self,
        model: Union[IModel, IPriorHyperparameters],
        ...
```

Model reuse across tasks

Emukit's composability allows to reuse components between methods. For example, we use the quadrature model defined above to perform an optimisation loop, and then integrate it using the quadrature API. The ability to reuse components in this way lowers implementation overhead, optimises utilisation of compute resources, and increases consistency.

```
# see BQ snippet for complete
# definition of the model
emukit_bq_model = VanillaBayesianQuadrature(
    base_gp=gp_model, X=X, Y=Y
)
```

```
bayesopt_loop = BayesianOptimizationLoop(
    model = emukit_bq_model, space = parameter_space
)
n_iterations = 20
bayesopt_loop.run_loop(
    user_function,
    stopping_condition=n_iterations
)

emukit_bq_model.integrate()
```

Multi-fidelity emulation

To support research on multi-fidelity emulation methods, Emukit implements both linear and non-linear multi-fidelity models. The user needs to provide data for each of the fidelities and make the choice of appropriate Gaussian process kernel. Emukit can then be used to define a combined multi-fidelity model. In the example below we define a linear multi-fidelity model, where the relationship between fidelities is linear.

```
# This utility method allows conversion
# of data from different fidelities
# to arrays where fidelity is represented
# as an input variable
X, Y = convert_xy_lists_to_arrays(
    [x_low, x_high],
    [y_low, y_high]
)

kernels = [
    GPy.kern.RBF(dim=1),
    GPy.kern.RBF(dim=1)
]
linear_mf_kernel =
    LinearMultiFidelityKernel(kernels)
gpy_linear_mf_model =
    GPyLinearMultiFidelityModel(
        X, Y,
        linear_mf_kernel,
        n_fidelities = 2
)
```

Other methods and features

In addition to the APIs discussed above, Emukit also provides basic support for sensitivity analysis and benchmarking. Further information about Emukit's functionality, including available implementations of acquisition functions, multi-output models, support for constraints and cost functions, and custom events in the outer loop may be found in library's website[2], documentation[3] and tutorial notebooks[4].

EMUKIT IN ACTION

Since its announcement in 2019 [25], Emukit was used in a wide range of research projects. In this section we review a selection of these projects to showcase the breadth of situations in which the library may be useful.

Methodological research

Because of its flexibility Emukit allows researchers to rapidly experiment with decision making methods in its suite. In this section we discuss several research papers that leverage this advantage to advance the field of decision making under uncertainty.

Optimisation of parameters in high dimensional structured data spaces is an increasingly important and challenging task. A common pattern is to use unsupervised learning methods to project parameters into low dimensional continuous representations, also known as latent spaces. There are multiple ways to approach the design of the Bayesian optimisation procedure on such latent spaces. Siivola et al. [26] studied the effects of various design choices. Namely, the effects of the dimensionality of the latent space, the optimisation bounds, and the choice of acquisition function were analysed. Emukit's plug-and-play approach allowed the researchers to facilitate measurement of these effects in isolation.

Emukit's composability was also leveraged for the implementation of BOSH, a sampling approach for Bayesian optimisation of functions with stochastic evaluations [27]. Authors used hierarchical Gaussian process as a surrogate and designed a novel BOSH acquisition function using the information-theoretic framework, incorporating both pieces in Emukit's Bayesian optimisation loop. Emukit was also used to assess BOSH performance against a variety of baselines.

Naslidnyk et al. [28] implemented a custom Bayesian quadrature model and used Emukit's existing BQ wrapper and decision loop in order to learn integrals of functions that are input invariant under some transformations. They tested their method on a problem from Fourier optics where the integral over a point spread functions of symmetric lenses was computed. Further, Gessner et al. [29] applied Emukit in the context of active multi-source Bayesian quadrature. The authors implemented a custom multi-source BQ model, a corresponding wrapper and even a custom multi-source acquisition function and point calculator which was possible due to Emukit's abstraction and plug-and-play capability.

Example applications

In this section we describe several cases where Emukit was used to solve applied research problems.

Bell et al. used Emukit to show how to conduct multiverse analysis for machine learning experiments [30]. Multiverse analysis was originally introduced in psychology, and allows researchers to explore the robustness and generality of claims by systematically examining the impact of different choices and variations in the experimental setup. The authors argue that the same concept can be applied to the machine learning: if a new technique, e.g. batch normalization, is proposed for an ML model, it should remain effective regardless of the model architecture, optimisation method, dataset, evaluation metric, and so on. The set of these variations comprises a multiverse, and needs to be explored effectively. The authors use surrogate modeling and Bayesian experimental design to systematically explore the effect of each choice. Emukit was chosen as an implementation tool because of the experimental design API it provides.

Uhrenholt and Jensen used Emukit's Bayesian optimisation module to solve the problem of finding settings of a musical synthesizer to produce a given sound [31]. A musical synthesizer produces sound by generating waveforms via oscillators. Created audio streams are then routed through a pipeline that consists (not necessary all) of mixing of separate streams, filtering, adding of noise, and saturation. Musicians can control the output sound by changing the configuration of the pipeline. In order to estimate the discrepancy between the produced sound and the target, the

2. https://emukit.github.io/
3. https://emukit.readthedocs.io/en/latest/
4. https://nbviewer.org/github/emukit/emukit/blob/main/notebooks/index.ipynb

authors designed a novel modeling approach, in which Gaussian process is used to model the distribution of the output's L2 norm. The flexibility of Emukit allowed to implement this customization directly, without necessary effort duplication. Emukit's API also facilitated a fair comparison to the standard Bayesian optimisation used as a baseline.

Liyanage et al. faced the problem of combining data from multiple particle accelerators, including Large Hadron Collider and Relativistic Heavy Ion Collider, to study the properties of quark-gluon plasma [32]. Nuclear collision experiments generate a large body of measurements with varying levels of uncertainty that would be expensive to quantify with simulations. Instead the authors proposed to use inexpensive statistical emulators and use transfer learning to leverage similarities between different heavy ion collisions systems. This new technique is based on multi-fidelity emulation, making Emukit an obvious implementation choice.

Related work

The Python ecosystem is rich with powerful scientific packages, including those for decision making methods.

In particular, *Bayesian optimisation* enjoys a wide selection of tools and frameworks. Spearmint [14] and GPyOpt [33] are among the first Python packages for Bayesian optimisation, the latter being an inspiration for the first release of Emukit. BoTorch [34] is a popular library for Bayesian optimisation based on PyTorch. Similarly, Trieste [35] also focuses on Bayesian optimisation but uses Tensorflow as a backend. More options, such as pyGPGO [36], scikit-optimize [37], RoBO [38], are also available.

For *Bayesian quadrature* and *Bayesian experimental design* the choice of frameworks is more scarce. Namely, bayesquad [39] appears to be another Python package for Bayesian quadrature. The Python library ProbNum [40] supports a variety of Bayesian quadrature methods, but it's lack of hyperparameter tuning capability reduces its practical relevance significantly in its current form. Optbayesexpt [41] and NEXTorch [42] provide Bayesian experimental design functionality adopted for their respective fields. Elements of experimental design can also be found in Trieste [35].

The key difference between Emukit and the mentioned libraries is the fact that Emukit does not dictate a particular modeling framework, allowing for flexibility in the choice of computational backends. In addition, Emukit does not focus on a single method and provides a common core set of abstractions for optimisation, quadrature and experimental design. Emukit provides a unique way of using the same model backend for all tasks, which increases consistency, reduces implementation and computing overheads.

Likewise, we were not able to locate a Python library other than Emukit that provides multi-fidelity emulation functionality. A notable package for research on multi-fidelity methods is MF2 [43] that implements a variety of multi-fidelity benchmark functions, but does not have modeling capabilities.

Looking at a wider family of optimisation libraries in Python, Optuna [44] is a popular choice for hyperparameter optimisation. Similarly to Emukit, Optuna is framework agnostic, however it provides a different set of optimisation methods, focusing on evolutionary, genetic and Monte Carlo based approaches. Finally, Ray Tune [45] is a well known scalable platform in Python on which other model optimisation frameworks can be executed. Emukit can potentially be integrated with Ray Tune as an optimisation library. This work was not carried out yet, and may be a future development direction.

Limitations

The design choices made in Emukit have proven to be highly beneficial for rapid prototyping and experimentation. However they also led to some of the key limitations of the library.

Emukit does not provide modeling capabilities, and instead requires users to provide their own surrogate models. This requires certain level of proficiency with probabilistic modeling, and can prevent some people from using the library. While we aim to mitigate this with extensive collection of examples and tutorials, and Emukit is successfully used in teaching university-level courses and scientific summer schools, the library still cannot be recommended for absolute beginners.

Emukit puts a strong emphasis on plug-and-play construction of the decision making loop. All components interact via interfaces, and they require a common data format to communicate, which in Emukit is a Numpy array. On one hand Numpy is a defacto standard in scientific Python which means it is reasonable to expect all Emukit users to be able to use Numpy. On the other hand, linear algebra operations in Numpy cannot be GPU-accelerated. This means that while individual components of the outer loop (e.g. a model) can run on GPU, the entire end-to-end process in Emukit is CPU-bound. This issue severely limits Emukit's performance comparing to the libraries that have chosen to rely on a fixed computational backend (BoTorch/PyTorch or Trieste/Tensorflow). This can be potentially mitigated by using specialized libraries that allow Numpy to be run on GPU, such as Numba [46] and CuPy [47].

Conclusions

Emukit is built on a realisation that common methods for decision making under uncertainty – such as Bayesian optimisation, Bayesian quadrature and experimental design – follow the same iterative pattern, and therefore can be seen as instances of a unified high level framework. Emukit provides high level interfaces for these methods that are built on the core set of common abstractions. To enable researchers and practitioners to iterate and experiment quickly Emukit follows plug-and-play design, allowing users to swap out a single part of the decision making loop without affecting other components. Since its initial release in 2019, Emukit has been successfully used in academic research, industry, and teaching.

Emukit has multiple potential growth directions. Mitigation of limitations discussed earlier may improve user experience and overall quality of the library. Integration with other tools in scientific Python ecosystem (e.g. Ray Tune) may increase Emukit's visibility within the community. New functionality, such as multi-objective Bayesian optimisation, would expand library's capabilities and give users new ways to do research with Emukit.

Researchers and enthusiasts from any scientific or industrial domain are welcome to explore the potential of using Emukit for their applications, to contibute new functionality, and to take part in the discussions around the library. The authors are always open to feedback and comments about improvements to the library. The Emukit repository is available on GitHub: https://github.com/EmuKit/emukit.

ACKNOWLEDGMENTS

AP and NL acknowledge the support from the Engineering and Physical Sciences Research Council (EPSRC) and the Alan Turing Institute under grant EP/V030302/1. MM gratefully acknowledges financial support by the European Research Council through ERC StG Action 757275 / PANAMA; the DFG Cluster of Excellence "Machine Learning - New Perspectives for Science", EXC 2064/1, project number 390727645; the German Federal Ministry of Education and Research (BMBF) through the Tübingen AI Center (FKZ: 01IS18039A); and funds from the Ministry of Science, Research and Arts of the State of Baden-Württemberg.

We are thankful to every community member for discussions, comments, bug reports, pull requests, as well as everyone who used the library in their work, study or research. The full list of people who contributed code to Emukit can be found at https://github.com/EmuKit/emukit/graphs/contributors.

REFERENCES

[1] B. Settles, "Active learning literature survey," University of Wisconsin–Madison, Computer Sciences Technical Report 1648, 2009.

[2] C. E. Rasmussen and C. K. I. Williams, *Gaussian Processes for Machine Learning*, ser. Adaptive Computation and Machine Learning. MIT Press, 2006.

[3] T. K. Ho, "Random decision forests," in *Proceedings of 3rd International Conference on Document Analysis and Recognition*, vol. 1, 1995, pp. 278–282 vol.1, https://doi.org/10.1109/ICDAR.1995.598994.

[4] D. J. C. MacKay, "A practical Bayesian framework for backprop networks," *Neural Computation*, 1991.

[5] A. O'Hagan, M. C. Kennedy, and J. E. Oakley, "Uncertainty analysis and other inference tools for complex computer codes," in *Bayesian Statistics 6*, 1998.

[6] M. C. Kennedy and A. O'Hagan, "Predicting the output from a complex computer code when fast approximations are available," *Biometrika*, vol. 87, no. 1, pp. 1–13, 2000, https://doi.org/10.1093/biomet/87.1.1.

[7] ——, "Bayesian calibration of computer models," *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, vol. 63, no. 3, pp. 425–464, 2001, https://doi.org/10.1111/1467-9868.00294.

[8] S. Conti, J. P. Gosling, J. E. Oakley, and A. O'Hagan, "Gaussian process emulation of dynamic computer codes," *Biometrika*, vol. 96, no. 3, pp. 663–676, 06 2009, https://doi.org/10.1093/biomet/asp028.

[9] S. Conti and A. O'Hagan, "Bayesian emulation of complex multi-output and dynamic computer models," *Journal of Statistical Planning and Inference*, vol. 140, no. 3, pp. 640–651, 2010, https://doi.org/10.1016/j.jspi.2009.08.006.

[10] J. Mockus, V. Tiesis, and A. Zilinskas, "The application of Bayesian methods for seeking the extremum," *Towards Global Optimization*, vol. 2, no. 117-129, p. 2, 1978.

[11] R. Garnett, *Bayesian Optimization*. Cambridge University Press, 2023, to appear.

[12] A. Baheri and C. Vermillion, "Altitude optimization of airborne wind energy systems: A Bayesian optimization approach," in *2017 American Control Conference (ACC)*. IEEE, 2017, pp. 1365–1370, https://doi.org/10.23919/acc.2017.7963143.

[13] D. E. Graff, E. I. Shakhnovich, and C. W. Coley, "Accelerating high-throughput virtual screening through molecular pool-based active learning," *Chemical science*, vol. 12, no. 22, pp. 7866–7881, 2021, https://doi.org/10.1039/d0sc06805e.

[14] J. Snoek, H. Larochelle, and R. P. Adams, "Practical Bayesian optimization of machine learning algorithms," *Advances in neural information processing systems*, vol. 25, 2012.

[15] B. Avent, J. González, T. Diethe, A. Paleyes, and B. Balle, "Automatic discovery of privacy–utility Pareto fronts," *Proceedings on Privacy Enhancing Technologies*, vol. 4, pp. 5–23, 2020, https://doi.org/10.2478/popets-2020-0060.

[16] P. Diaconis, "Bayesian numerical analysis," in *Statistical decision theory and related topics IV*. Springer-Verlag New York, 1988, vol. 1, pp. 163–175.

[17] A. O'Hagan, "Some Bayesian numerical analysis," *Bayesian Statistics*, vol. 4, pp. 345–363, 1992.

[18] P. Hennig, M. A. Osborne, and H. P. Kersting, *Probabilistic Numerics: Computation as Machine Learning*. Cambridge University Press, 2022, https://doi.org/10.1017/9781316681411.

[19] C. E. Rasmussen and Z. Ghahramani, "Bayesian Monte Carlo," in *Advances in Neural Information Processing Systems*, vol. 15, 2002, pp. 505–512.

[20] A. Giovagnoli, "The Bayesian design of adaptive clinical trials," *International Journal of Environmental Research and Public Health*, vol. 18, no. 2, 2021, https://doi.org/10.3390/ijerph18020530.

[21] E. Pauwels, C. Lajaunie, and J.-P. Vert, "A Bayesian active learning strategy for sequential experimental design in systems biology," *BMC Systems Biology*, vol. 8, no. 1, pp. 1–11, 2014, https://doi.org/10.1186/s12918-014-0102-6.

[22] A. E. Gongora, B. Xu, W. Perry, C. Okoye, P. Riley, K. G. Reyes, E. F. Morgan, and K. A. Brown, "A Bayesian experimental autonomous researcher for mechanical design," *Science advances*, vol. 6, no. 15, p. eaaz1708, 2020, https://doi.org/10.1126/sciadv.aaz1708.

[23] B. Peherstorfer, K. Willcox, and M. Gunzburger, "Survey of multifidelity methods in uncertainty propagation, inference, and optimization," *SIAM Review*, vol. 60, no. 3, pp. 550–591, 2018, https://doi.org/10.1137/16M1082469.

[24] The GPy authors, "GPy: A Gaussian process framework in Python," http://github.com/SheffieldML/GPy, 2012.

[25] A. Paleyes, M. Pullin, M. Mahsereci, C. McCollum, N. D. Lawrence, and J. González, "Emulation of physical processes with Emukit," *Second workshop on machine learning and the physical sciences, NeurIPS*, 2019.

[26] E. Siivola, A. Paleyes, J. González, and A. Vehtari, "Good practices for Bayesian optimization of high dimensional structured spaces," *Applied AI Letters*, vol. 2, no. 2, p. e24, 2021, https://doi.org/10.1002/ail2.24.

[27] H. B. Moss, D. S. Leslie, and P. Rayson, "BOSH: Bayesian optimisation by sampling hierarchically," *Workshop on Real World Experimental Design and Active Learning, ICML*, 2020.

[28] M. Naslidnyk, J. Gonzalez, and M. Mahsereci, "Invariant priors for Bayesian quadrature," in *Your Model is Wrong: Robustness and misspecification in probabilistic modeling Workshop, NeurIPS*, 2021.

[29] A. Gessner, J. Gonzalez, and M. Mahsereci, "Active multi-information source Bayesian quadrature," in *Proceedings of The 35th Uncertainty in Artificial Intelligence Conference*, ser. Proceedings of Machine Learning Research, R. P. Adams and V. Gogate, Eds., vol. 115. PMLR, 2020, pp. 712–721.

[30] S. J. Bell, O. Kampman, J. Dodge, and N. Lawrence, "Modeling the machine learning multiverse," *Advances in Neural Information Processing Systems*, vol. 35, pp. 18 416–18 429, 2022.

[31] A. K. Uhrenholt and B. S. Jensen, "Efficient Bayesian optimization for target vector estimation," in *The 22nd International Conference on Artificial Intelligence and Statistics*. PMLR, 2019, pp. 2661–2670.

[32] D. Liyanage, Y. Ji, D. Everett, M. Heffernan, U. Heinz, S. Mak, and J.-F. Paquet, "Efficient emulation of relativistic heavy ion collisions with transfer learning," *Phys. Rev. C*, vol. 105, p. 034910, Mar 2022, https://doi.org/10.1103/PhysRevC.105.034910. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevC.105.034910

[33] The GPyOpt authors, "GPyOpt: A Bayesian optimization framework in Python," http://github.com/SheffieldML/GPyOpt, 2016.

[34] M. Balandat, B. Karrer, D. Jiang, S. Daulton, B. Letham, A. G. Wilson, and E. Bakshy, "BoTorch: A framework for efficient Monte-Carlo Bayesian optimization," *Advances in neural information processing systems*, vol. 33, pp. 21 524–21 538, 2020.

[35] V. Picheny, J. Berkeley, H. B. Moss, H. Stojic, U. Granta, S. W. Ober, A. Artemev, K. Ghani, A. Goodall, A. Paleyes *et al.*, "Trieste: Efficiently exploring the depths of black-box functions with Tensorflow," *arXiv preprint arXiv:2302.08436*, 2023.

[36] J. Jiménez and J. Ginebra, "pyGPGO: Bayesian optimization for Python," *Journal of Open Source Software*, vol. 2, no. 19, p. 431, 2017, https://doi.org/10.21105/joss.00431.

[37] G. Louppe, "Bayesian optimisation with scikit-optimize," in *PyData Amsterdam*, 2017.

[38] A. Klein, S. Falkner, N. Mansur, and F. Hutter, "RoBO: A flexible and robust Bayesian optimization framework in Python," in *NIPS 2017 Bayesian Optimization Workshop*, 2017.

[39] OxfordML, "bayesquad," https://github.com/OxfordML/bayesquad, 2013.

[40] J. Wenger, N. Krämer, M. Pförtner, J. Schmidt, N. Bosch, N. Effenberger, J. Zenn, A. Gessner, T. Karvonen, F.-X. Briol, M. Mahsereci, and P. Hennig, "ProbNum: Probabilistic numerics in Python," 2021.

[41] R. D. McMichael, S. M. Blakley, and S. Dushenko, "Optbayesexpt: Sequential Bayesian experiment design for adaptive measurements," *Journal of Research of the National Institute of Standards and Technology*, vol. 126, pp. 1–5, 2021.

[42] Y. Wang, T.-Y. Chen, and D. G. Vlachos, "NEXTorch: a design and Bayesian optimization toolkit for chemical sciences and engineering,"

Journal of Chemical Information and Modeling, vol. 61, no. 11, pp. 5312–5319, 2021, https://doi.org/10.1021/acs.jcim.1c00637.s001.

[43] S. van Rijn and S. Schmitt, "MF2: A collection of multi-fidelity benchmark functions in Python," *Journal of Open Source Software*, vol. 5, no. 52, p. 2049, 2020, https://doi.org/10.21105/joss.02049. [Online]. Available: https://doi.org/10.21105/joss.02049

[44] T. Akiba, S. Sano, T. Yanase, T. Ohta, and M. Koyama, "Optuna: A next-generation hyperparameter optimization framework," in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2019, https://doi.org/10.1145/3292500.3330701.

[45] R. Liaw, E. Liang, R. Nishihara, P. Moritz, J. E. Gonzalez, and I. Stoica, "Tune: A research platform for distributed model selection and training," *arXiv preprint arXiv:1807.05118*, 2018.

[46] S. K. 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*, 2015, pp. 1–6, https://doi.org/10.1145/2833157.2833162.

[47] R. Nishino and S. H. C. Loomis, "CuPy: A numpy-compatible library for Nvidia GPU calculations," *31st conference on neural information processing systems*, vol. 151, no. 7, 2017.

MDAKits: A Framework for FAIR-Compliant Molecular Simulation Analysis

Irfan Alibay[‡†*], Lily Wang[‡†], Fiona Naughton[§†], Ian Kenney[¶†], Jonathan Barnoud[‖], Richard J Gowers[‡], Oliver Beckstein[¶]

Abstract—The reproducibility and transparency of scientific findings are widely recognized as crucial for promoting scientific progress. However, when it comes to scientific software, researchers face many barriers and few incentives to ensure that their software is open to the community, thoroughly tested, and easily accessible. To address this issue, the `MDAKits` framework has been developed, which simplifies the process of creating toolkits for the MDAnalysis simulation analysis package (https://www.mdanalysis.org/) that follow the basic principles of FAIR (findability, accessibility, interoperability, and reusability). The MDAKit framework provides a cookiecutter template, best practices documentation, and a continually validated registry. Registered kits are continually tested against the latest release and development version of the MDAnalysis library and their code health is indicated with badges. Users can browse the registry frontend (https://mdakits.mdanalysis.org/) to find new packages, learn about associated publications, and assess the package health in order to make informed decisions about using a MDAKit in their own research. The criteria for registering an MDAKit (open source, version control, documentation, tests) are similar to the criteria required for publishing a paper in a software journal, so we encourage and support publication in, e.g., the Journal of Open Source Software, creating further academic incentive for researchers to publish code. Through the MDAKits framework, we aim to foster the creation of a diverse ecosystem of sustainable community-driven downstream tools for MDAnalysis and hope to provide a blueprint for a model for growing communities around other scientific packages.

Index Terms—Molecular Dynamics Simulations, Python, MDAnalysis, ecosystem

Introduction

Software has become increasingly essential to research. In many areas, it underlies fundamental tasks such as generating, processing, analyzing, storing, visualizing, and communicating the key results and insights ultimately published.

Scientific code frequently fails to meet FAIR tenets, impeding scientific progress

Despite the importance of software, it is typically not central to the publication peer review process in many scientific fields. Consequently, scientific code frequently fails to meet the basic tenets of FAIR: findability, accessibility, interoperability, and reusability [1], [2]. With the publication of "The FAIR Guiding Principles for scientific data management and stewardship" in 2016 and the follow-up "FAIR Principles for Research Software (FAIR4RS Principles)" in 2022, it has become increasingly acknowledged that abiding by the principles of FAIR is crucial to promoting robust, reproducible, and efficient scientific discovery and innovation [1], [2]. We believe that extending FAIR principles to include open-source software not only significantly advances that goal, but furthermore is necessary for transparent research. Open sharing of code brings a number of substantial benefits to the scientific community. For example, scientists can accurately replicate a given methodology or re-use previous code, reducing duplication of effort and reducing the risk of implementation errors. Indeed, the molecular simulation community in particular has made a concerted effort over recent years to encourage the open sharing of scientific codes [3]. For example, as of July 2022, over 4700 GitHub repositories containing Python code that makes use of MDAnalysis [4], [5] have been made publicly available[1].

However, simply sharing code is not sufficient to fulfill FAIR guidelines. In fact, making software FAIR compliant requires significant investment and often expert knowledge on the part of the developers, especially if the code was written specifically for a particular research project. For example, the Python ecosystem is so dynamic that it is common for research code to rapidly become obsolete or unusable if a new version of a key library is released. To fulfill the Reusability tenet of FAIR alone, code should include documentation, version control, and dependency management. Ideally, it would also include unit tests, examples, and packaging. Even when code is released in reference to a publication, it often falls short of ideal FAIR standards. A short survey of publications in Scopus [6] and the Journal of Open Source Software [7] over 2017—2021 identified that out of a total 720 papers citing MDAnalysis [4], [5], only 43 linked to code available on a version control platform such as GitHub, GitLab, or Bitbucket. Of these, only 18 met the requirements of best practices: they implemented unit tests, comprehensive documentation, and some means of installation.

Two major factors contribute to the lack of open-source FAIR compliant code. Firstly, code is typically written by scientists with no formal training or support in programming, for whom implementing FAIR principles can pose an intimidating and tedious barrier. Secondly, despite the substantial investment of effort and time required to implement best practices, publishing FAIR

† *These authors contributed equally.*
∗ *Corresponding author: ialibay@mdanalysis.org*
‡ *Open Molecular Software Foundation, Irvine, CA, USA*
§ *Cardiovascular Research Institute, University of California, San Francisco, San Francisco, CA, USA*
¶ *Arizona State University, Tempe, AZ, USA*
‖ *Centro Singular de Investigación en Tecnoloxías Intelixentes, Santiago de Compostela, Spain*

software is not typically appreciated with academic recognition or reward. Fostering a culture of open-source FAIR software requires addressing both.

Centralized open-source packages such as MDAnalysis offer a limited solution

One solution is to consolidate scientific code around a small number of large, central packages. MDAnalysis [4], [5] is a widely-used open-source Python library for molecular simulation data. With over 18 years of development by more than 180 developers, MDAnalysis has refined its code base to offer a mature, robust, flexible API that offers a range of high-performance tools to extract, manipulate, and analyze data from the majority of common simulation formats. MDAnalysis tools have been used for a variety of scientific applications ranging from exploring protein-ligand interactions [8], [9], [10], to understanding lipid behavior [11], [12], to assessing the behavior of novel materials [13], [14].

Initially, MDAnalysis focused on growing the developer and user community by encouraging users to contribute their code directly to the MDAnalysis library. Notable examples of this include the waterdynamics [15] and ENCORE [16] analysis modules. This approach of encouraging code to be contributed to a central package has also been successfully taken by packages such as cpptraj [17] and the GROMACS tools [18]. It has a number of key advantages for users and the original developers:

- MDAnalysis can ensure that the code follows best practices (including documentation and tests).
- Code is promoted and made freely accessible to all MDAnalysis users.
- Maintenance, support, and potential updates are performed by the experienced MDAnalysis developer team, ensuring that the contributed code remains functional even while the other parts of the library change. The original developers can thus focus on other work.

However, the many costs of this approach can, under some conditions, result in unsustainable, untenable disadvantages:

- Ensuring that the code follows best practices often requires long review periods and strict code-style adherence, thus slowing down the availability of the new code in a released version of the package.
- The necessity of keeping the API stable between major releases precludes quick releases of breaking changes. In general, a mature package such as MDAnalysis has a slow release cycle, so new features and bug fixes can take months to become available in new releases.
- As MDAnalysis implicitly agrees to maintain any code that we release, a certain level of understanding and expertise is required from the maintainers. If the core developer team lacks expertise in a specific discipline or subdiscipline, adding new code in these areas introduces a substantial maintenance burden should the original code contributors not be available to help with maintenance. Consequently, it is impractical to include recently released or cutting-edge techniques in the core library.
- Introducing new package dependencies incurs software stack maintenance costs for many users who may not require this additional code.
- Code contributors lose complete control of their code.

The many disadvantages listed above can severely limit the usefulness of centralizing code around one monolithic package. Indeed, encountering these issues when attempting to expand the core MDAnalysis library attests that this approach is not the most suited for the MDAnalysis community.

Implementing an ecosystem of downstream packages for more sustainable progress

We believe that a sustainable alternative solution is for communities such as MDAnalysis to encourage, educate, and foster researchers in their efforts towards developing individual software. We have developed a program of structured technical assistance to help researchers implement best practices and publish their code within a growing ecosystem of toolkits that we have called `MDAKits` (MDAnalysis Toolkits). We have also developed a platform called the "MDAKit registry" (https://mdakits.mdanalysis.org/mdakits.html) where packages that meet certain standards are advertised to the community. The MDAKit ecosystem builds on the success of other community packages such as PLUMED's PLUMED-NEST [19], AiiDA's plugin registry [20], or the napari-hub [21] of plugins for the napari image viewer [22], all of which list available tools that are known to work in their respective user communities.

Our technical assistance begins with cookiecutter templates and example repositories. Here we model best practices, promote the use of helpful tools, e.g., for checking code coverage, and reduce the work required to set up processes such as continuous integration, versioned documentation, packaging and deployment. Developers can also reach out to the MDAnalysis community for feedback, technical assistance, or even make connections with new co-developers and potential users. Decoupled from MDAnalysis's release cycle, developers are able to introduce new changes as required, keeping complete control over their code-base. Joining an MDAnalysis registry allows for frequent and streamlined communication between MDAnalysis and downstream developers, allowing developers to be efficiently forewarned about potential breaking changes.

Although establishing such an ecosystem of MDAnalysis-supported packages requires substantial investment from MDAnalysis developers, this approach is nonetheless likely to be far more sustainable than centralizing around a super-package. Offering technical assistance to individual developers in implementing best practices constitutes a large part of the effort; however, this level has thus far proven much lower than the effort associated with adding additional functionality to the core MDAnalysis library, and we believe that it will continue to remain so. Furthermore, as the ecosystem grows, we hope that an increasing portion of the community will participate in taking care of the packages and registry, and that the culture of following best practices and publishing code will gain momentum in itself.

In part, we hope that this momentum will be driven by users and user expectations. Users of the MDAnalysis ecosystem gain huge benefit from the MDAKit registry. They are able to see new software as it gets added, rather than having to comb through literature or rely on developers advertising the code themselves. They are also able to easily verify the current development status of a package and whether it is being actively maintained and passing tests with both released and in-development versions of MDAnalysis. In the future, the registry could contain information about the health of a given codebase, such as its activity

1. Based on a search for repositories containing `import MDAnalysis`.

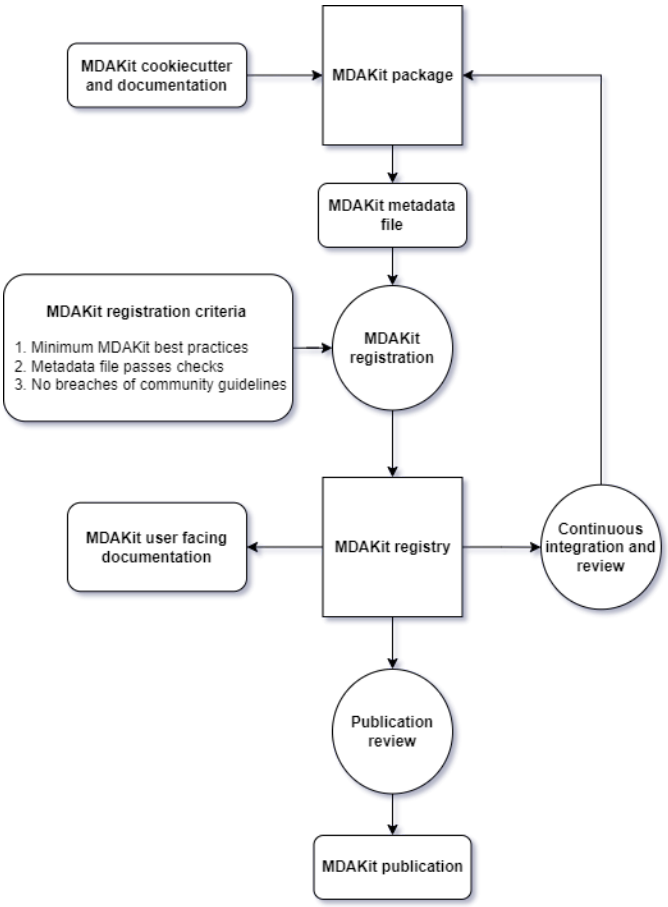

Fig. 1: *Workflow diagram of the MDAKit framework. Starting from the creation of an MDAKit package, with the help of documentation and the MDAKit cookiecutter, the package then goes through the process of being added to the MDAKit registry, undergoing continuous validation and review and eventually reaching the stage of publication.*

status and if it optimally leverages high performance MDAnalysis components (e.g. highly optimised PBC-aware distance routines). Packages on the registry also come with easy-to-find instructions on how to easily install and run a given package, significantly lowering the technical barrier to use and experimentation. As the maintenance remains the burden of the package owners, unfortunately the risk remains that packages on the registry may eventually become out-of-date, which is indeed one of the major disadvantages of this approach. However, the registry significantly increases the likelihood that packages will reach users who will become sufficiently motivated to contribute or take over their maintenance and development.

In the rest of this document we outline our expectations for MDAKits in terms of best practices and how we implement their registration and continuous validation.

The MDAKit framework

The MDAKit framework (Fig. 1) is designed to be a complete workflow to help and incentivize developers to go from the initial stages of package development all the way through to the long-term maintenance of a mature codebase, while adhering to best practices.

Main goals

As such, the main goals of the proposed MDAKit framework are:

1) To help as many packages as possible implement best practices and develop user communities.
2) To ensure that members of the MDAnalysis community can easily identify new packages of interest and know to what extent they are suitable for production use.
3) To improve contacts between MDAnalysis core library developers and those developing packages using MDAnalysis.
4) To encourage participation from the community at all steps of the process.

We wish to state three main points that the framework is *not* designed for:

1) The MDAKit framework is not intended to restrict the packages which can participate. It is our view that all packages at any stage of their development are of value to the community. As such, we aim for framework components to be as non-blocking as possible.
2) It is not the intention of any parts of this framework to take control or ownership of the packages that participate within it. The original code developers retain full ownership, control, and responsibility for their packages and may optionally participate in any part of this framework.
3) We also do not want to block future contributions to the core library. If new code in MDAKits prove particularly popular, and the MDAKit developers are amenable to contributing these back into the core library, the MDAnalysis team will work with them to integrate additional functionality into MDAnalysis itself.

Overview of the framework

The MDAKit framework (Fig. 1) is a multi-step process. In the first step of the MDAKit framework, developers create an initial package which is intended to achieve a set purpose of their choice. To help with this process, MDAnalysis provides a cookiecutter template specifically for MDAKits [23], alongside documentation on best practices and how to optimally use the MDAnalysis API. An overview of what we consider to be best practices for the contents of MDAKit packages is included in Section Defining MDAKits: best practice package features. We note that at this point MDAKits are not expected to fully adhere to best practices, but should at least meet the minimum requirements defined in Section Defining MDAKits: best practice package features before moving to the next step along this process.

Once a package is suitably developed, code owners are encouraged to add the details of their code to the "MDAKit registry" which advertises their package to the MDAnalysis community and offers continual validation and review tools to help with package maintenance. Section The MDAKit registry contains more information about the MDAKit registry, including the registration process (Section Registering MDAKits). Briefly, the registration process involves submitting a metadata file to the registry that contains essential information about the MDAKit, such as where the source code is provided, who the code authors are, and how to install the MDAKit. The contents of this metadata file is reviewed both by automatic code checks and the MDAnalysis developer team before being added to the registry. We want to highlight that this process does not include checks on scientific validity or code health. In fact, none of the processes in this framework account for the scientific validity of the MDAKits. While members of the

community are free to offer help, scientific or technical validity is beyond the scope of what is feasible with the MDAnalysis registry.

Upon registration, the MDAKit is automatically advertised to the MDAnalysis community (see Section Advertising MDAKits). In the first instance this amounts to a set of auto-generated pages that expose the details in the metadata file provided in the registration step. Additional tags and badges are also included that reflect the current status and health of the package. Examples include:

- whether or not it is compatible with the latest versions of MDAnalysis
- what percentage of the codebase is covered by unit tests
- what type or extent of documentation is provided
- what Python versions are currently supported.

This status information is provided as part of checks done during the continual validation and review steps (see Sections Continual validation and Continual review) of the framework. These steps involve a mix of regularly scheduled automatic (e.g., linters and unit test execution) checks and more infrequent manual (e.g., code reviews) processes. It is our intention that code health analysis will help developers maintain and improve their codes, as well as suitably warn potential users about issues they may encounter when using a given codebase.

Where possible, the framework encourages a code review process to be carried out by members of the MDAnalysis community. The aim here is to work with developers in identifying potential areas of improvements for both MDAKits and the core MDAnalysis library (see Sections Continual review and Feeding back into the MDAnalysis library). We aim to tie this process closely to the review processes of journals such as the Journal of Open Source Software [7], which would help lower the barrier towards and encourage an eventual publication (Section Towards publication).

Defining MDAKits: best practice package features

Here we list requirements that we believe MDAKits should strive to fulfill in order to meet best practices in Python package usability and maintenance. To help with implementing these, a cookiecutter is provided which offers a template for potential MDAKits to follow [23]. We want to emphasize again that the aim of the MDAKit project is to encourage best practices whilst also minimizing barriers to sharing code where possible. Therefore, only a minimal set of requirements listed here as *required* are necessary for MDAKits to be included in the MDAKit registry. Similarly, we do not mean to enforce the label of MDAKit on any package; the process is fully optional and the code owners may choose whether to associate themselves with it.

All MDAKits must implement the features on the list of **required features** in order to become registered:

- Code in the package *uses MDAnalysis* (Code using MDAnalysis (required)).
- Open source code is published under an *OSI approved license* (Open source code under an OSI approved license (required)).
- Code is *versioned* and provided in an *accessible version-controlled repository* (Versioning and provision under an accessible version-controlled repository (required)).
- Code *authors and maintainers are clearly designated* (Designated code authors and maintainers (required)).
- *Documentation* is provided (Documentation (required)).
- *Tests and continuous integration* are present (Tests and continuous integration (required)).

The following are **highly recommended features**:

- Code is *installable as a standard package* (Packaging).
- Information on *bug reporting, user discussions, and community guidelines* is made available (Bug reporting, user discussions, and community guidelines).

Code using MDAnalysis (required): This is the base requirement of all MDAKits. The intent of the MDAKit framework is to support packages existing downstream from the MDAnalysis core library. MDAKits should therefore contain code using MDAnalysis components which are intended by the package authors to address the MDAKit's given purpose.

Open source code under an OSI approved license (required): The core aim of MDAKits is to encourage the open sharing of codes to potential users within the MDAnalysis community and beyond. To achieve this, we require that codes under this framework be released as open source. Here we define open source as being under an Open Source Initiative (OSI) approved license [24].

As of writing, the MDAnalysis library is currently licensed under GPLv2+ [25]. Due to limitations with this license type, we cannot currently recommend other licenses than GPLv2+ for codes importing MDAnalysis. However, we hope to relicense to a less restrictive license. In this event, MDAKits will be able to adopt a wider range of OSI approved licenses.

Versioning and provision under an accessible version-controlled repository (required): The ability to clearly identify changes in a codebase is crucial to enabling reproducible science. By referencing a specific release version, it is possible to trace back any bug fixes or major changes which could lead to a difference in results obtained with a later version of the same codebase. Whilst we encourage the use of Semantic Versioning ("semver") [26], any PEP440 [27] compliant versioning specification would be suitable for MDAKits.

Beyond versioning releases, it is also crucial to be able to develop code in a sustainable and collaborative manner. The most popular way of achieving this is through the use of version control through Git [28]. We require all MDAKits to be held in a publicly facing version controlled repository such as GitHub [29], GitLab [30], or Bitbucket [31].

Designated code authors and maintainers (required): In order for users to be able to contact the code owners and maintainers, all MDAKits should clearly list their authors and a means of contacting the persons responsible for maintaining the codebase. To incentivize and recognize contributors throughout the life of a project, we recommend the use of a version controlled "authors" file which lists the authors to a codebase over time.

Documentation (required): Describing what a given code does and how to use it is a key component of open sharing. Ideally a package would include a complete description of the entire codebase, including both API documentation and some kind of user guide with worked examples on how the code could be used in certain scenarios. Whilst this is recommended as best practices for an MDAKit, we recognize that this is not always feasible, especially in the early stages of development. Therefore, the minimum requirement for MDAKits is to have a readme file which details the key aspects of the MDAKit, such as what it is intended to do, how to install it, and a basic usage example.

For best practices, we strongly recommend using docstrings (see PEP 257 [32]) to document code components and using a tool such as ReadTheDocs [33] to build, version and host documentation in a user-friendly manner. We also recommend using duecredit [34] to provide the correct attributions to a given method if it has been published previously.

Tests and continuous integration (required): Testing is a critical component to ensure that code behaves as intended. Not only does it prevent erroneous coding, but it also assures users that the code they rely on is working as intended. We require at least a single regression test for major functionality to qualify for the registry (i.e. if a toolkit implements a new analysis method, at least one test that checks to see if the analysis code yields the expected value on provided data; regression tests can often double as example documentation).

Ideally one should do full unit testing of the contents of a code, ensuring that not only a specific outcome is reached, but also that each smaller component works. As part of best practices, we highly recommend implementing tests using a framework such as pytest [35] for executing tests and codecov [36] to capture which lines are covered by the tests. We strongly encourage that a minimum of at least 80% of the code lines be covered by tests.

To ensure that tests are run regularly, the recommended best practice is to implement a continuous integration pipeline that performs the tests every time new code is introduced. We encourage the use of free pipelines such as GitHub Actions [37] to implement continuous integration.

Packaging: Providing a standard means of installing code as a package is important to ensure that other code can correctly link to (i.e., `import` in the case of Python) and use its contents. Whilst it can be easy to expect users to simply read a Python script, look at its required dependencies, and install them manually, this can quickly become unreasonable should the code grow beyond a single file. Additionally, the lack of clearly defined versions, including the intended Python versions, can lead to inoperable code.

As best practices we heavily encourage the use of setuptools [38] or an alternative such as poetry [39] for package installation. We also encourage that packages be available on common package repositories such as PyPi [40] and conda-forge [41]. The use of such repositories and their respective package managers can significantly lower the barrier to installing a package, enabling new users to rapidly get started using it.

Bug reporting, user discussions, and community guidelines: To help maintain and grow the project, it is important to specify where users can raise any issues they might have about the project or simply ask questions about its operation. To achieve this, we recommend at the very least adding documentation that points users to an issue tracker.

Key to successfully building a user community is ensuring that there are proper guidelines in place for how users will interact with a project [42]. As best practices we recommend making a code of conduct available that defines how users should interact with developers and each other within a project. It is also advised to provide information on how users can contribute to the project as part of its documentation.

The MDAKit registry

As defined in Section The MDAKit framework, once MDAKits are created, we encourage that they be added to the MDAKit registry. The registry not only provides a platform to advertise MDAKits to the MDAnalysis user community at the web page https://mdakits.mdanalysis.org/, but also offers tools and workflows to help packages improve and continue to be maintained. Here we describe the various processes that occur within the registry. We note that we expect the exact details of how these processes are implemented to evolve over time based on feedback from MDAKit developers and other members of the MDAnalysis community.

MDAKit registry contents

The main aim of the registry is to hold information about MDAKits. The contents of the registry therefore center around a list of packages and the metadata associated with each MDAKit. This metadata has the form of two files: one containing user-provided information on the package contents (see Section Registering MDAKits), and the other a set of mostly auto-generated details indicating the code health of the package (see Section Advertising MDAKits).

This metadata is used for two purposes: continuous integration testing and documentation. Continuous testing, helper methods and workflows are used to regularly install MDAKits and run their test suite (if available) to check if they still work as intended. Should the tests fail, package maintainers are automatically contacted and failure information is recorded in the code health metadata to inform users. For the registry documentation, the metadata is used to provide user-facing information about the various MDAKits in the registry, their contents, how to install them, and their current status as highlighted by continuous integration tests. The registry also includes further information such as user guides and tutorials on the MDAKit framework, helping developers to implement their own MDAKits.

Registering MDAKits

A key feature of the MDAKit framework is the process of adding MDAKits to the registry. As previously defined, our intent is to offer a low barrier to entry and have packages be registered early in their development cycles. This allows developers to benefit from the MDAKit registry validation and review processes early on, hopefully lowering the barrier to further improvements and encouraging early user interactions and feedback.

From an MDAKit developer standpoint, the registration process involves opening a pull request against the MDAKit registry that adds a new YAML file with metadata about the project. The metadata, as detailed in Fig. 2, contains information such as the MDAKit description, source code location, installation instructions, how to run tests, and where to find documentation. Complete details about the metadata file specification are provided in the MDAKit registry documentation.

After a pull request is opened, the MDAnalysis developers review the contents of the submission based on the following criteria:

1) If the required features for MDAKits are met (Section Defining MDAKits: best practice package features), that is:
    1. Does the MDAKit contain code using MDAnalysis?
    2. Is the MDAKit license appropriate?
    3. Is the MDAKit code offered through a suitable version-controlled platform?

```yaml
## Required entries
project_name: propkatraj
authors:
  - https://github.com/Becksteinlab/propkatraj/blob/main/AUTHORS
maintainers:
  - ianmkenney
  - IAlibay
  - orbeckst
description:
    pKa estimates for proteins using an ensemble approach
keywords:
  - pKa
  - protein
license: GPL-2.0-or-later
project_home: https://github.com/Becksteinlab/propkatraj/
documentation_home: https://becksteinlab.github.io/propkatraj/
documentation_type: UserGuide + API

## Optional entries
install:
  - pip install propkatraj
src_install:
  - pip install git+https://github.com/Becksteinlab/propkatraj@main
python_requires: ">=3.8"
mdanalysis_requires: ">=2.0.0"
run_tests:
  - pytest --pyargs propkatraj.tests
test_dependencies:
  - mamba install pytest MDAnalysisTests
project_org: Becksteinlab
development_status: Mature
publications:
  - https://zenodo.org/record/7647010
  - https://doi.org/10.1021/ct200133y
  - https://doi.org/10.1085/jgp.201411219
```

Fig. 2: *YAML metadata file for an MDAKit entry of the propkatraj package, stored as* `mdakits/propkatraj/metadata.yaml` *in the registry repository.*

4. Are the MDAKit authors and maintainers clearly designated in the metadata file?
5. Is there at least minimal documentation in place detailing the MDAKit and its functionality?
6. Are there at least minimal regression tests available within the MDAKit code?

2) If the metadata file passes linting and integration checks
3) That there are no potential breaches of community guidelines

Once the criteria are fulfilled the metadata is merged and the MDAKit is considered registered. Updates to the MDAKit metadata can be carried out at any time after registration by opening pull requests to change the metadata file contents.

Advertising MDAKits

Registered MDAKits are automatically added to the registry's public facing documentation at https://mdakits.mdanalysis.org/mdakits.html. This involves an indexable list of entries for all registered MDAKits. Each entry displays available information from the provided metadata, e.g., what the MDAKit does, any relevant keywords, how to obtain the source code, how to install the package, and where to find relevant documentation. Alongside this information is also a set of badges which describe the current health of the codebase, allowing users to rapidly identify which packages are currently active, and their level of code maturity. This includes information such as which MDAnalysis library versions the package is compatible with. We further plan to add more information, such as how much test coverage the package has, and what type of MDAnalysis API extensions are provided (e.g., using base classes such as AnalysisBase or ReaderBase).

Information about MDAKits is continually updated, either through automatic checks or manual additions provided by package owners updating the metadata files. We aim for the MDAKit registry to be immutable (aside from special cases covered by Section Raising issues, concerns, and paths to registry removal). Therefore, should an MDAKit stop being maintained, it will not be removed from the index but instead labeled as abandoned.

Continual validation

The MDAKit registry implements workflows to validate the code health of registered packages. This mostly centers around a test matrix that regularly runs to check if the latest MDAKit release can be installed and if unit tests pass with both the latest release of MDAnalysis and the development version. Should tests fail regularly, an issue is automatically raised on the MDAKit registry issue tracker contacting the package maintainers and letting them know of the failure. The auto-generated code health metadata for the MDAKit is also updated to reflect whether or not the tests are currently failing or passing.

In the future we hope to expand these tests to include more historical releases of the MDAKits and the MDAnalysis library, checks for different architectures (non-x86), and operating systems. We may also expand the checks to consider the cross-compatibility of MDAKits with each other, offering insights on which packages can be safely used together.

Continual review

To help package growth and improvements, it is our goal for the registry to become a platform that allows members of the MDAnalysis community to offer feedback on MDAKits over the lifetime of their inclusion on the registry. Unfortunately, as MDAnalysis developers can only devote limited time towards the registry, offering regularly scheduled comprehensive reviews of packages is too large an undertaking to be practical.

Instead, we aim to use a system of badges and achievements to push packages towards gradual improvements. For example, we may offer an achievement that encourages MDAKits to use high performance PBC-aware distance routines defined in `MDAnalysis.lib.distances` instead of relying on NumPy's `linalg` method to find the distance between two points. Once MDAKit owners believe that they have suitably updated their code to fulfill the relevant badge criteria, they can open a pull request highlighting these changes and have developers review these smaller, more focused updates.

MDAKit users are also encouraged to provide feedback, request improvements, and report bug fixes. However, this should happen outside the scope of the registry; instead, we ask users to use the MDAKit's own issue tracker for these.

Feeding back into the MDAnalysis library

The existence of the MDAKits framework does not preclude the addition of new codes and methods to the core MDAnalysis library. The MDAKit registry, and especially the ongoing review process, provides a platform for MDAnalysis and MDAKit developers to interact and work together to identify common goals and areas of improvements for both upstream and downstream packages. In particular, MDAnalysis developers will work with

MDAKit developers to see if any popular MDAKit methods, components or other means to improve core method performance and lower the barrier to downstream package development can and should be implemented back into the core MDAnalysis library.

Towards publication

We have laid out a number of best practices here that we encourage MDAKits to fulfill. These essentially amount to the majority of the contribution criteria for submissions to software-focused journals such as the Journal Open Source Software (JOSS) [7]. In order to incentivize developers, we heavily encourage MDAKits to consider submission to a journal such as JOSS once they meet the required levels of best practices. To aid in this process, the MDAnalysis developers will in the first instance work with journal editors at JOSS to create a streamlined process to submit MDAKits as JOSS entries [43]. The details of this process are still under development.

Raising issues, concerns, and paths to registry removal

If community members (users, developers or otherwise) have concerns about an MDAKit, we primarily encourage them to raise issues on the MDAKit's own issue tracker. However, in situations where the MDAKit maintainers cannot respond, or if the concern relates to code of conduct breaches, MDAnalysis developers may step in. If an MDAKit has systemic issues with its correctness, the MDAKit may be given special annotations warning users about the issues before using the code. We generally view the MDAKit registry as a permanent record, and avoid removing packages after registration even if they become fully obsolete. However, we reserve the right to remove packages at our discretion in specific cases, notably code of conduct breaches and violation of the GitHub terms of service [44].

Long term registry maintenance and support

As with most MDAnalysis projects, long-term support for the MDAKit framework and especially the registry is expected to be carried out by contributors from the MDAnalysis community. Members of the MDAnalysis core development team lead the maintenance of the registry and are also responsible for passing judgment on serious events such as code of conduct breaches. In the long term, we hope that any gains in popularity of the MDAKits framework are accompanied by an increase in community involvement in reviews and other maintenance tasks.

Examples of MDAKits

The web frontend of the registry (Fig. 3) provides a searchable database of packages. At the moment, seven MDAKits are registered that already showcase the breadth of specialized tools for the analysis of biomolecular simulations. For example, *mdacli* provides a commandline interface to analysis tools in MDAnalysis itself. *openmm-mdanalysis-reporter* enhances the interoperability with the popular OpenMM MD engine [45]. *hole2-mdakit* interfaces with the legacy HOLE2 program for the analysis of pores and tunnels in proteins such as ion channels [46], [47]. The *lipyds* package provides a suite of tools for the analysis of biological membranes in simulations [11]. *ProLIF* quantitatively analyzes the interactions between small molecules such as drugs and biomolecules (protein, nucleic acids) [10].


Show 10 entries Search:

List of registered MDAKits

MDAKit	Keywords	Authors	CI badges
hole2-mdakit	pores, ion channels, transporters, HOLE	hole2-mdakit authors	latest passed develop passed
lipyds	lipids, membranes	lilyminium	latest passed develop unavailable
mdachecker	structure analysis, conformation checks	IAlibay	latest passed develop passed
mdacli	command line interface, molecular-dynamics	mdacli authors	latest passed develop passed
openmm-mdanalysis-reporter	OpenMM, Reporters	openmm-mdanalysis-reporter authors	latest passed develop passed
ProLIF	drug-design, cheminformatics, molecular-dynamics	Cédric Bouysset	latest passed develop passed
propkatraj	pKa, protein	propkatraj authors	latest passed develop passed

Showing 1 to 7 of 7 entries Previous 1 Next

Fig. 3: *Web front end of the searchable MDAKit registry with registered MDAKits. Badges indicate code health based on continuous validation against the latest release and development version of the MDAnalysis library.*

Conclusions

We introduce the MDAnalysis `MDAKits` framework for scientific software packages. This framework is designed to assist and incentivize the creation of FAIR-compliant (findable, accessible, interoperable, and reusable) packages that use and extend MDAnalysis. We describe the current state of scientific code, which is typically published either in independent repositories of varying quality, or as additions to a large, monolithic package. We summarize the limitations of each approach that result in code that falls short of FAIR principles, or may end up impractical to sustain as a long-term strategy. We propose the MDAKits framework as an alternative solution to support developers in creating new packages, guiding them through the process of achieving best practices and FAIR compliance.

In Section The MDAKit framework we lay out the aims and structure of an MDAKit, summarizing the minimal and optimal requirements that we think necessary to build sustainable, reusable software. These include publishing code under a suitable open-source license, the use of version control, comprehensive documentation, thorough unit tests, and packaging the software following modern best practices. In Section Defining MDAKits: best practice package features we outline our vision and implementation of the MDAKit registry, a public facing repository that promotes MDAKits to the MDAnalysis community. The MDAKit registry offers regular checks and reviews in order to help improve and maintain the listed MDAKits. We describe a structured workflow that begins from the initial registration of MDAKits and reaches as far as eventual publication in software-focused journals such as JOSS.

This document is just the first step and broad guide to our vision of developing a rich, diverse software ecosystem, and we are still in the early stages of implementing MDAKits. While we expect that we may need to revisit and refine our strategy to best serve the needs of the community, we believe that the fundamental framework outlined here will bring great benefit to the software written and used by scientists, and thereby empower transparent

and reproducible research.

Acknowledgments

We gratefully acknowledge the 184 developers and countless community members who have contributed to the MDAnalysis project since its inception and NumFOCUS for its support as our fiscal sponsor.

The work on the MDAKits project and this publication have been made possible in part by CZI grant DAF2021-237663 and grant DOI https://doi.org/10.37921/426590wiobus from the Chan Zuckerberg Initiative DAF, an advised fund of Silicon Valley Community Foundation (funder DOI 10.13039/100014989).

Jonathan Barnoud has received financial support from the Agencia Estatal de Investigación (Spain) (REFERENCIA DEL PROYECTO / AEI / CÓDIGO AXUDA), the Xunta de Galicia - Consellería de Cultura, Educación e Universidade (Centro de investigación de Galicia accreditation 2019-2022 ED431G-2019/04 and Reference Competitive Group accreditation 2021-2024, CÓDIGO AXUDA) and the European Union (European Regional Development Fund - ERDF).

REFERENCES

[1] N. P. Chue Hong, D. S. Katz, M. Barker, A.-L. Lamprecht, C. Martinez, F. E. Psomopoulos, J. Harrow, L. J. Castro, M. Gruenpeter, P. A. Martinez, and T. Honeyman, "FAIR Principles for Research Software (FAIR4RS Principles)," *Research Data Alliance*, 2022, https://doi.org/10.15497/RDA00068. [Online]. Available: https://zenodo.org/record/6623556#.YqCJTJNBwlw

[2] M. D. Wilkinson, M. Dumontier, I. J. Aalbersberg, G. Appleton, M. Axton, A. Baak, N. Blomberg, J.-W. Boiten, L. B. da Silva Santos, P. E. Bourne, J. Bouwman, A. J. Brookes, T. Clark, M. Crosas, I. Dillo, O. Dumon, S. Edmunds, C. T. Evelo, R. Finkers, A. Gonzalez-Beltran, A. J. G. Gray, P. Groth, C. Goble, J. S. Grethe, J. Heringa, P. A. C. 't Hoen, R. Hooft, T. Kuhn, R. Kok, J. Kok, S. J. Lusher, M. E. Martone, A. Mons, A. L. Packer, B. Persson, P. Rocca-Serra, M. Roos, R. van Schaik, S.-A. Sansone, E. Schultes, T. Sengstag, T. Slater, G. Strawn, M. A. Swertz, M. Thompson, J. van der Lei, E. van Mulligen, J. Velterop, A. Waagmeester, P. Wittenburg, K. Wolstencroft, J. Zhao, and B. Mons, "The FAIR Guiding Principles for scientific data management and stewardship," *Scientific Data*, vol. 3, no. 1, p. 160018, Mar. 2016, https://doi.org/10.1038/sdata.2016.18. [Online]. Available: https://www.nature.com/articles/sdata201618

[3] W. P. Walters, "Code Sharing in the Open Science Era," *Journal of Chemical Information and Modeling*, vol. 60, no. 10, pp. 4417–4420, Oct. 2020, https://doi.org/10.1021/acs.jcim.0c01000. [Online]. Available: https://doi.org/10.1021/acs.jcim.0c01000

[4] N. Michaud-Agrawal, E. J. Denning, T. B. Woolf, and O. Beckstein, "MDAnalysis: A Toolkit for the Analysis of Molecular Dynamics Simulations," *J Comp Chem*, vol. 32, pp. 2319–2327, 2011, https://doi.org/10.1002/jcc.21787. [Online]. Available: http://www.ncbi.nlm.nih.gov/pmc/articles/PMC3144279/

[5] R. J. Gowers, M. Linke, J. Barnoud, T. J. E. Reddy, M. N. Melo, S. L. Seyler, D. L. Dotson, J. Domanski, S. Buchoux, I. M. Kenney, and O. Beckstein, "MDAnalysis: A Python package for the rapid analysis of molecular dynamics simulations." in *Proceedings of the 15th Python in Science Conference*, S. Benthall and S. Rostrup, Eds., Austin, TX, 2016, pp. 102–109, https://doi.org/10.25080/Majora-629e541a-00e.

[6] "Scopus," https://www.scopus.com/. [Online]. Available: https://www.scopus.com/

[7] "Journal of Open Source Software," https://joss.theoj.org. [Online]. Available: https://joss.theoj.org

[8] I. Alibay, "IAlibay/MDRestraintsGenerator: MDRestraintsGenerator 0.1.0," Mar. 2021, https://doi.org/10.5281/zenodo.4570556. [Online]. Available: https://zenodo.org/record/4570556

[9] D. B. Kokh, B. Doser, S. Richter, F. Ormersbach, X. Cheng, and R. C. Wade, "A workflow for exploring ligand dissociation from a macromolecule: Efficient random acceleration molecular dynamics simulation and interaction fingerprint analysis of ligand trajectories," *The Journal of Chemical Physics*, vol. 153, no. 12, p. 125102, Sep. 2020, https://doi.org/10.1063/5.0019088. [Online]. Available: https://aip.scitation.org/doi/10.1063/5.0019088

[10] C. Bouysset and S. Fiorucci, "ProLIF: a library to encode molecular interactions as fingerprints," *Journal of Cheminformatics*, vol. 13, no. 1, p. 72, Sep. 2021, https://doi.org/10.1186/s13321-021-00548-6. [Online]. Available: https://doi.org/10.1186/s13321-021-00548-6

[11] K. A. Wilson, L. Wang, Y. C. Lin, and M. L. O'Mara, "Investigating the lipid fingerprint of SLC6 neurotransmitter transporters: a comparison of dDAT, hDAT, hSERT, and GlyT2," *BBA Advances*, vol. 1, p. 100010, Jan. 2021, https://doi.org/10.1016/j.bbadva.2021.100010. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2667160321000090

[12] P. Smith and C. D. Lorenz, "LiPyphilic: A Python Toolkit for the Analysis of Lipid Membrane Simulations," *Journal of Chemical Theory and Computation*, vol. 17, no. 9, pp. 5907–5919, Sep. 2021, https://doi.org/10.1021/acs.jctc.1c00447. [Online]. Available: https://doi.org/10.1021/acs.jctc.1c00447

[13] R. Gowers, M. Matta, and H. Nguyen, "kugupu/kugupu: v0.1.2," Feb. 2021, https://doi.org/10.5281/zenodo.4545322. [Online]. Available: https://zenodo.org/record/4545322

[14] P. Loche, H. Jaeger, A. Schlaich, M. Becker, S. Gravelle, P. Stärk, and S. Velpuri, "MAICoS," Feb. 2022. [Online]. Available: https://gitlab.com/maicos-devel/maicos

[15] R. Araya-Secchi, T. Perez-Acle, S.-g. Kang, T. Huynh, A. Bernardin, Y. Escalona, J.-A. Garate, A. D. Martínez, I. E. García, J. C. Sáez, and R. Zhou, "Characterization of a Novel Water Pocket Inside the Human Cx26 Hemichannel Structure," *Biophysical Journal*, vol. 107, no. 3, pp. 599–612, Aug. 2014, https://doi.org/10.1016/j.bpj.2014.05.037. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0006349514006018

[16] M. Tiberti, E. Papaleo, T. Bengtsen, W. Boomsma, and K. Lindorff-Larsen, "ENCORE: Software for Quantitative Ensemble Comparison," *PLOS Computational Biology*, vol. 11, no. 10, p. e1004415, Oct. 2015, https://doi.org/10.1371/journal.pcbi.1004415. [Online]. Available: https://journals.plos.org/ploscompbiol/article?id=10.1371/journal.pcbi.1004415

[17] D. R. Roe and T. E. Cheatham, "PTRAJ and CPPTRAJ: Software for Processing and Analysis of Molecular Dynamics Trajectory Data," *Journal of Chemical Theory and Computation*, vol. 9, no. 7, pp. 3084–3095, Jul. 2013, https://doi.org/10.1021/ct400341p. [Online]. Available: https://doi.org/10.1021/ct400341p

[18] M. J. Abraham, T. Murtola, R. Schulz, S. Páll, J. C. Smith, B. Hess, and E. Lindahl, "GROMACS: High performance molecular simulations through multi-level parallelism from laptops to supercomputers," *SoftwareX*, vol. 1-2, pp. 19–25, Sep. 2015, https://doi.org/10.1016/j.softx.2015.06.001. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2352711015000059

[19] M. Bonomi, G. Bussi, C. Camilloni, G. A. Tribello, P. Banáš, A. Barducci, M. Bernetti, P. G. Bolhuis, S. Bottaro, D. Branduardi, R. Capelli, P. Carloni, M. Ceriotti, A. Cesari, H. Chen, W. Chen, F. Colizzi, S. De, M. De La Pierre, D. Donadio, V. Drobot, B. Ensing, A. L. Ferguson, M. Filizola, J. S. Fraser, H. Fu, P. Gasparotto, F. L. Gervasio, F. Giberti, A. Gil-Ley, T. Giorgino, G. T. Heller, G. M. Hocky, M. Iannuzzi, M. Invernizzi, K. E. Jelfs, A. Jussupow, E. Kirilin, A. Laio, V. Limongelli, K. Lindorff-Larsen, T. Löhr, F. Marinelli, L. Martin-Samos, M. Masetti, R. Meyer, A. Michaelides, C. Molteni, T. Morishita, M. Nava, C. Paissoni, E. Papaleo, M. Parrinello, J. Pfaendtner, P. Piaggi, G. Piccini, A. Pietropaolo, F. Pietrucci, S. Pipolo, D. Provasi, D. Quigley, P. Raiteri, S. Raniolo, J. Rydzewski, M. Salvalaglio, G. C. Sosso, V. Spiwok, J. Šponer, D. W. H. Swenson, P. Tiwary, O. Valsson, M. Vendruscolo, G. A. Voth, A. White, and The PLUMED consortium, "Promoting transparency and reproducibility in enhanced molecular simulations," *Nature Methods*, vol. 16, no. 8, pp. 670–673, Aug. 2019, https://doi.org/10.1038/s41592-019-0506-8. [Online]. Available: https://www.nature.com/articles/s41592-019-0506-8

[20] "AiiDA plugin registry," https://aiidateam.github.io/aiida-registry/. [Online]. Available: https://aiidateam.github.io/aiida-registry/

[21] Chan Zuckerberg Initiative, "napari hub," https://www.napari-hub.org/about, last accessed 2022-08-05. [Online]. Available: https://www.napari-hub.org/about

[22] N. Sofroniew, T. Lambert, K. Evans, J. Nunez-Iglesias, G. Bokota, P. Winston, G. Peña-Castellanos, K. Yamauchi, M. Bussonnier, D. Doncila Pop, A. Can Solak, Z. Liu, P. Wadhwa, A. Burt, G. Buckley, A. Sweet, L. Migas, V. Hilsenstein, L. Gaifas, J. Bragantini, J. Rodríguez-Guerra, H. Muñoz, J. Freeman, P. Boone, A. Lowe, C. Gohlke, L. Royer, A. PIERRÉ, H. Har-Gil, and A. McGovern, "napari: a multi-dimensional image viewer for Python," May 2022, https://doi.org/10.5281/zenodo.3555620. [Online]. Available: https://doi.org/10.5281/zenodo.3555620

[23] L. Wang, I. Alibay, and F. Naughton, "Cookiecutter for MDAnalysis-

based packages," https://github.com/MDAnalysis/cookiecutter-mdakit. [Online]. Available: https://github.com/MDAnalysis/cookiecutter-mdakit
[24] Open Source Initiative, "Licenses and Standards," https://opensource.org/licenses, last accessed 2022-08-05. [Online]. Available: https://opensource.org/licenses
[25] "GNU General Public License v2.0 - GNU Project - Free Software Foundation," https://www.gnu.org/licenses/old-licenses/gpl-2.0.en.html, last accessed 2022-08-04. [Online]. Available: https://www.gnu.org/licenses/old-licenses/gpl-2.0.en.html
[26] T. Preston-Werner, "Semantic Versioning 2.0.0," https://semver.org/, last accessed 2022-08-04. [Online]. Available: https://semver.org/
[27] "PEP 257 – Docstring Conventions | peps.python.org," https://peps.python.org/pep-0257/. [Online]. Available: https://peps.python.org/pep-0257/
[28] "Git," https://git-scm.com/, last accessed 2022-08-04. [Online]. Available: https://git-scm.com/
[29] GitHub, Inc, "GitHub," https://github.com, last accessed 2022-08-04. [Online]. Available: https://github.com
[30] GitLab Inc., "GitLab," https://about.gitlab.com/, last accessed 2022-08-04. [Online]. Available: https://about.gitlab.com/
[31] Atlassian, "Bitbucket," https://bitbucket.org/product, last accessed 2022-08-04. [Online]. Available: https://bitbucket.org/product
[32] "PEP 440 – Version Identification and Dependency Specification | peps.python.org," https://peps.python.org/pep-0440/, last accessed 2022-08-04. [Online]. Available: https://peps.python.org/pep-0440/
[33] Read the Docs, Inc, "Read the Docs," https://readthedocs.org/, 2022. [Online]. Available: https://readthedocs.org/
[34] Y. O. Halchenko, M. Visconti di Oleggio Castello, M. Hanke, J. Gors, M. Szczepanik, C. Barnes, E. Irvine, P. R. Raamana, C. J. Markiewicz, J. Wilk, D. Volgyes, K. Leinweber, L. Estève, O. Beckstein, and O. F. Gulban, "duecredit/duecredit: 0.9.1," Apr. 2021, https://doi.org/10.5281/zenodo.4685131. [Online]. Available: https://zenodo.org/record/4685131
[35] H. Krekel, B. Oliveira, R. Pfannschmidt, F. Bruynooghe, B. Laugher, and F. Bruhin, "pytest-dev/pytest," https://github.com/pytest-dev/pytest, 2004. [Online]. Available: https://github.com/pytest-dev/pytest
[36] Codecov LLC, "Codecov," https://about.codecov.io/, 2022. [Online]. Available: https://about.codecov.io/
[37] GitHub, Inc, "GitHub Terms of Service," https://docs.github.com/en/site-policy/github-terms/github-terms-of-service, last accessed 2022-08-04. [Online]. Available: https://docs.github.com/en/site-policy/github-terms/github-terms-of-service
[38] "pypa/setuptools," https://github.com/pypa/setuptools, Aug. 2022, original-date: 2016-03-29T14:02:33Z. [Online]. Available: https://github.com/pypa/setuptools
[39] "Poetry - Python dependency management and packaging made easy," https://python-poetry.org/. [Online]. Available: https://python-poetry.org/
[40] "PyPI · The Python Package Index," https://pypi.org/. [Online]. Available: https://pypi.org/
[41] Conda-Forge Community, "The conda-forge Project: Community-based Software Distribution Built on the conda Package Format and Ecosystem," *Zenodo*, Jul. 2015, https://doi.org/10.5281/ZENODO.4774216. [Online]. Available: https://zenodo.org/record/4774216
[42] A. Grossfield, "How to be a Good Member of a Scientific Software Community [Article v1.0]," *Living Journal of Computational Molecular Science*, vol. 3, no. 1, pp. 1473–1473, 2021, https://doi.org/10.33011/livecoms.3.1.1473. [Online]. Available: https://livecomsjournal.org/index.php/livecoms/article/view/v3i1e1473
[43] "Submitting a paper to JOSS," https://joss.readthedocs.io/en/latest/submitting.html, 2018, last accessed 2022-08-03. [Online]. Available: https://joss.readthedocs.io/en/latest/submitting.html
[44] GitHub, Inc, "GitHub Actions," https://github.com/features/actions, 2022. [Online]. Available: https://github.com/features/actions
[45] P. Eastman, J. Swails, J. D. Chodera, R. T. McGibbon, Y. Zhao, K. A. Beauchamp, L.-P. Wang, A. C. Simmonett, M. P. Harrigan, C. D. Stern, R. P. Wiewiora, B. R. Brooks, and V. S. Pande, "OpenMM 7: Rapid development of high performance algorithms for molecular dynamics," *PLOS Computational Biology*, vol. 13, no. 7, pp. 1–17, 07 2017, https://doi.org/10.1371/journal.pcbi.1005659.
[46] O. S. Smart, J. G. Neduvelil, X. Wang, B. A. Wallace, and M. S. P. Sansom, "HOLE: A program for the analysis of the pore dimensions of ion channel structural models," *J Molecular Graphics*, vol. 14, pp. 354–360, 1996, https://doi.org/10.1016/s0263-7855(97)00009-x. [Online]. Available: http://www.holeprogram.org/
[47] L. S. Stelzl, P. W. Fowler, M. S. P. Sansom, and O. Beckstein, "Flexible gates generate occluded intermediates in the transport cycle of LacY," *J Mol Biol*, vol. 426, pp. 735–751, 2014, https://doi.org/10.1016/j.jmb.2013.10.024. [Online]. Available: http://doi.org/10.1016/j.jmb.2013.10.024

The Pandata Scalable Open-Source Analysis Stack

James A. Bednar[‡*], Martin Durant[‡]

Abstract—As the scale of scientific data analysis continues to grow, traditional domain-specific tools often struggle with data of increasing size and complexity. These tools also face sustainability challenges due to a relatively narrow user base, a limited pool of contributors, and constrained funding sources. We introduce the Pandata open-source software stack as a solution, emphasizing the use of domain-independent tools at critical stages of the data life cycle, without compromising the depth of domain-specific analyses. This set of interoperable and compositional tools, including Dask, Xarray, Numba, hvPlot, Panel, and Jupyter, provides a versatile and sustainable model for data analysis and scientific computation. Collectively, the Pandata stack covers the landscape of data access, distributed computation, and interactive visualization across any domain or scale. See github.com/panstacks/pandata to get started using this stack or to help contribute to it.

Index Terms—distributed computing, data visualization, workflows

Introduction

Modern science, engineering, and analysis workflows rely on computational tools for data processing, such as the foundational NumPy [1], Pandas [2], Matplotlib [3], and Jupyter [4] libraries for Python. Over the past few decades, different research areas and communities have built their own "stacks", i.e. layered sets of software tools that are combined to solve problems in a particular research area (see [5] for examples for big data, and [6] for the idea of a layered stack). For instance, a Python geoscience stack might combine GDAL and Fiona for geoscience file-format access, Xarray [7] (itself built on NumPy) for multidimensional array processing, cartopy (built on PROJ and NumPy) for handling earth coordinates, Matplotlib for plotting, and Jupyter for user interaction and code execution. A Python financial analysis stack might combine Pandas for file reading and columnar data manipulation, Matplotlib for plotting, TA-Lib for financial mathematics functions, and Jupyter for user interaction and code execution.

Many of these stacks' components date back decades before Python became popular, capturing important functionality but inheriting technical complexity and limitations that may no longer apply. For instance, domain-specific visualization and user interface (UI) tools are often tied to a local desktop operating system and graphical user interface (GUI), limiting the stack to working with data and compute resources available locally, and making it difficult to share work with colleagues at other sites or using other operating systems. Older tools are often either inherently single threaded, with no support for distributed computation, or specifically focused on supercomputing systems rather than flexibly supporting the current diversity of computing platforms (such as GPUs and cloud computing). Software is of course infinitely malleable, and so any such limitations could be addressed *in principle*, but in practice each domain-specific tool has a relatively narrow collection of users, contributors, and funding sources for that domain, limiting the scope of such development.

Could there be a better way? Yes! Today's Python ecosystem includes general data-processing tools that address tasks across all research areas, domains, and industries, with flexible tools for storing, reading, processing, plotting, analyzing, modeling, and exploring data of any kind. This paper introduces a specific collection of such tools called the Pandata stack. Like the "modern data stack" [8], [9], Pandata consists of tools that are engineered and tested to work well with each other to process very large datasets using distributed computation, with independently maintained components adding up to complete, end-to-end solutions for important workflows. Unlike the "modern data stack", the Pandata stack consists only of open-source tools and is also equally usable for small problems that fit onto a single laptop or local workstation, making it a solid basis for addressing data-centric problems of *any* size for any domain.

The Pandata stack's Python libraries are:

- **Domain independent:** Maintained, used, and tested by people from many different backgrounds
- **Efficient:** Run at machine-code speeds using vectorized data and compiled code
- **Scalable:** Run on anything from a single-core laptop to a thousand-node cluster
- **Cloud friendly:** Fully usable for local or remote compute using data on any file storage system
- **Multi-architecture:** Run on Mac/Windows/Linux systems, using CPUs or GPUs
- **Scriptable:** Fully support batch mode for parameter searches and unattended operation
- **Compositional:** Select which tools you need and put them together to solve your problem
- **Visualizable:** Support rendering even the largest datasets without conversion or approximation
- **Interactive:** Support fully interactive exploration, not just rendering static images or text files
- **Shareable:** Deployable as web apps for use by anyone anywhere
- **OSS:** Free, open, and ready for research or commercial use, without restrictive licensing

* Corresponding author: jbednar@anaconda.com
‡ Anaconda, Inc.

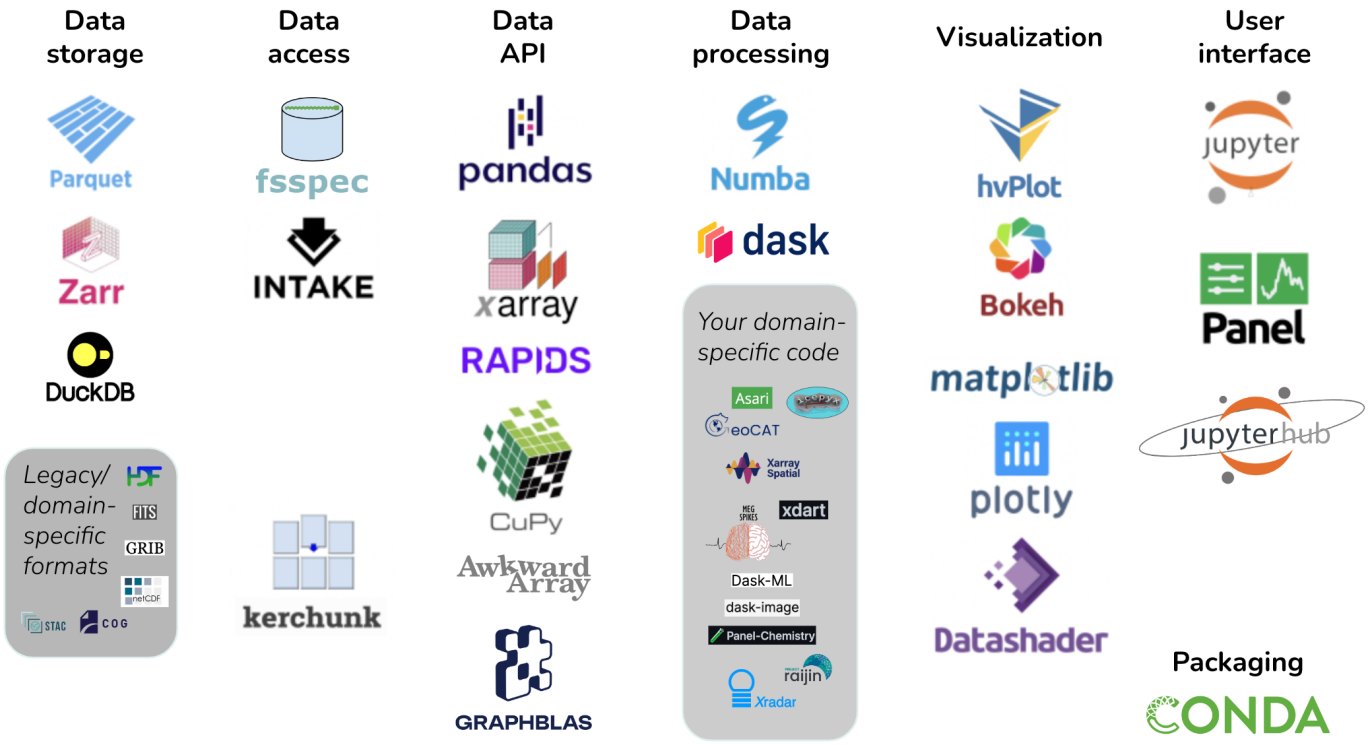

Fig. 1: *Pandata: the scalable open-source analysis stack.*

Figure 1 illustrates the Pandata stack, which includes the Parquet and Zarr file formats along with the fsspec, Kerchunk, Pandas, Xarray, RAPIDS, Dask, Numba, hvPlot, Panel, and Jupyter libraries. The tools are grouped into categories that will be described fully in later sections. Tools shown with a gray background are not part of Pandata itself but are illustrative of domain-specific code that builds on the Pandata stack to complete a solution for problems in a particular domain. We argue that Pandata libraries taken together as a group are a meaningful and well-tested base for general-purpose scientific computation across nearly all research areas and scientific domains. We will first look at the features that governed the selection of these particular libraries and that in many cases drive development priorities in the library itself. We will then describe the specific libraries involved and how they can be put together by users to solve computing tasks and how tool authors can build on them to support scientific computing in their own discipline.

Design considerations

Library authors have to choose between different designs and alternative technologies when deciding which features are implemented and how they are built. Here we will unpack each of the above considerations driving the Pandata collection of libraries and examine how these libraries implement those principles.

Domain independent

Pandata tools are all built to be used in any scientific domain, without making design choices that constrain them to a narrow subfield or topic. At first it may seem that having all your tools be written specifically for your particular research area would be ideal, but we argue that it is neither desirable nor feasible to draw strict boundaries between active scientific research areas. Should you have to switch to an entirely different stack whenever you do collaborative work? Or if you want to do something novel, differing slightly from assumptions previously made in your research area? A deep, difficult-to-change stack that encodes a fixed and unnecessarily brittle model of your domain will make scientific progress difficult to achieve. Of course, not every aspect of research crosses multiple domains or challenges core assumptions, but using Pandata tools ensures that doing so will come at relatively little cost, because the same tools are valid across a very wide range of fields. Pandata tools can also draw from expertise, effort, and funding across all of science and analysis, rather than from a single narrow area.

Efficient

Because Python itself is a relatively slow, interpreted language, native Python code is often suitable only for smaller problems, limiting the audience for any tool that is built purely in Python with Python dependencies. To make efficient use of the computational resources available, libraries in the Pandata stack either wrap fast code written in compiled languages like Fortran and C, or they use a Python compiler like Numba [10] or Cython [11] to compile Python syntax into machine code that runs as fast as compiled C or Rust or Fortran. Custom domain-specific code building on Pandata can easily use a compiler like Numba as well, so that the overall workflows and pipelines are not limited by the speed of Python.

Scalable

Even compiled code is not sufficient to address the largest problems, which require more memory than is available on a single machine, or require long computation that is feasible only when split across many processors working simultaneously. To cover all these cases, Pandata tools support *optional* distributed computation using Dask [12] on hundreds or thousands of processors,

while also fully supporting a single laptop or workstation. That way there is no need to switch to a different stack when you hit a problem larger than your machine, or conversely when you want to work on small problems that do not need extensive infrastructure. Supporting distributed computation efficiently requires tooling support at every level, starting with chunked binary file formats like Parquet and Zarr that let each processor access data independently, and culminating in visualization tools like Datashader that can render plots in separate chunks on each processor and combine them for the final display. Note that Pandata includes libraries like Pandas and Matplotlib that are not necessarily scalable on their own, but become scalable when combined with other Pandata tools (with e.g. Dask providing scalable DataFrames built on Pandas DataFrames, and Matplotlib being scalable when used with Datashader).

Cloud friendly

The enormous datasets now available in many research areas are typically hosted in a remote data center and can be prohibitively expensive (in time and money) to download locally. If you only need a small part of the data, Pandata tools like Pandas support efficient remote access to the relevant chunks of each file without having to download it all locally first, using fsspec to provide uniform filesystem-like access for data on local disk, cloud storage, web servers, and many other locations. If the total size of the chunks that you need to access is still too high to download locally, Pandata tools also support remote computation, with the processing done on a remote cloud-computing server with a high-bandwidth connection to the data server and a user interface using a web browser whether compute is local or remote. A typical scalable cloud-computing configuration for Pandata would be to have a remote JupyterHub system with Dask installed running on the cloud server close to the data. Users then contact the remote system from their local browser, initiating a remote session for computation and data access but with interactivity on the local machine using web-based controls and visuals. Using Datashader on the remote system provides server-side rendering so that only a rendered image of the data ever need be transferred across low-bandwidth connections, providing interactive local visualization without having to transfer large datasets. Pandata tools thus support either efficient access to remote datasets for local computation, or efficient fully remote computing with a local user interface, to make good use of cloud storage or compute resources. Note that "cloud friendly" is not the same as the term "cloud native" as used in the Modern Data Stack; Pandata tools fully support cloud usage but are equally at home on a local machine, and have no requirement for cloud resources, containers, microservices, or other architectures typical of cloud-native approaches.

Multi-architecture

Because researchers typically use Windows or Mac systems locally, while cloud servers typically run Linux, research code needs to be independent of the operating system for it to be equally well supported on local and remote systems. A software stack tied to a particular OS not only shuts out users who are not on that OS, it often implicitly favors either cloud or local usage for that stack, further reducing the community size and range of problems that can be addressed by a particular stack. Pandata tools are all either fully OS independent or support Linux (Intel or ARM), Mac (Intel or M1/M2), and Windows. Similarly, some problems can be addressed orders of magnitude faster on a general-purpose graphics processing unit (GPU) than on a conventional CPU, yet many researchers do not have access to GPUs or are working on problems unsuited for them, making it essential that a general-purpose software stack support both GPU and CPU usage as appropriate. The Pandata stack includes GPU-based equivalents for much of the functionality available on CPUs.

Scriptable

Many of the scientific tools that are common in particular domains (especially commercial tools provided alongside hardware used in that domain) require a GUI. GUI tools can be convenient for exploration, but without an accompanying non-GUI interface it is difficult to capture a reproducible set of steps for publication and dissemination. Additionally, GUIs often funnel users into a few well-supported operations without providing the level of configurability and customization needed to execute less common workflows, long-running jobs, or large parameter searches. Pandata tools that offer a GUI interface never require a local desktop, graphical display, or live interaction, making them fully suitable as a basis for reproducible, large-scale, and long-running research.

Compositional

When approaching a particular task, a researcher can either choose a monolithic tool that addresses all aspects of the task, or they can combine lower-level tools that together accomplish the task. Monolithic tools are attractive when they fully cover a specific use case, but given the dynamic nature of research, it is unreasonable to expect a monolithic tool to cover all the requirements of a particular research area, let alone across different areas of research. Accordingly, Pandata tools are all compositional, intended for independent use or in combination to solve specific problems. Where appropriate, each library has been adapted to work well with components from the other Pandata libraries, allowing a researcher to mix and match and combine projects in novel ways to reach their goals. Pandata projects also take part in larger efforts to improve compositionality and interoperability like `numpy.org/neps` and `scientific-python.org/specs` and by implementing existing APIs that allow a Dask dataframe to be a drop-in replacement for a Pandas dataframe, let hvPlot be a drop-in replacement for Pandas plotting, or allow a Panel app to display ipywidgets (or vice versa). Of course, compositional approaches can take more initial user effort and expertise than an out-of-the box monolithic approach. To address well-established complex tasks, a domain-specific monolithic tool can be built out of Pandata tools, while other tasks that need to remain flexible can be built compositionally as needed for that task.

Visualizable

For computing tasks that operate in the background without any visual output, it is easy to ensure that they are scriptable and cloud friendly. However, doing good science requires understanding each of the processing steps in complete detail, and if there are any unobservable black boxes in your workflows, that is surely where bugs will hide. To make sure that the work is being done correctly, it is crucial to be able to represent each of the computing steps involved in a way that a human can easily grasp what is happening, with a minimum of extra effort that discourages exploration. Often, a remote computing job will export data that is then subsampled and downloaded locally for analysis, but any step that adds friction and covers up the raw data makes it more likely that important issues and insights will be missed. Accordingly,

the HoloViz tools included in the Pandata stack are designed to make the full set of data available at any point in the computation, by supporting efficient in-place interactive visualization of even distributed or GPU-based data of any size using Datashader and hvPlot, assembled from separately computed chunks for display on any device.

Interactive

Supporting unattended batch-mode computation is important for doing comprehensive parameter evaluation, but batch runs tend to keep research focused on specific well-trodden paths, changing only a few options at any one time and thus limiting the search space that gets explored for a model or dataset. Using only a batch approach makes it easy to miss important opportunities or to fail to understand major limitations, by simply re-running the same code paths "blind" every time. Pandata tools like hvPlot and Panel running in Jupyter make it easy even for remote cloud workflows to have interactive widgets and controls in a web browser, to more easily explore parameter combinations with immediate feedback to help understand how the system behaves.

Shareable

Creating easy interactive visualizations is great, but if they are limited to your own desktop or installation, the impact of your ideas and approaches will be limited. With enough HTML/JavaScript/CSS web-technology experience, any computation can be wrapped up as an interactive website, but many scientists lack front-end web-development skills. The Panel tool in Pandata makes it simple to convert any Jupyter notebook into a web app that can be shared with collaborators or the public to disseminate the results of a project. Panel apps can be shared as static JavaScript-based HTML files (for small datasets), WASM-based HTML files (running Python in the browser), or via a live Python server (for the largest computations and datasets). (Also see "Environments and reproducibility" below for other aspects of ensuring that your work is shareable.)

OSS

For software to be fully accessible across years, labs, collaborators, and research problems it is crucial that there be no licensing restrictions that prevent it from being used across the entire discipline and on all relevant hardware and software platforms. Pandata tools are all permissively licensed so that they are freely usable in academia, industry, government, and by private individuals, easily scaled up to the largest problems or in new contexts without having to obtain permission or pay additional fees. Beyond the licensing, it is also crucial that the underlying source code is accessible, so that every processing step involved can be examined and justified. That way, if the research hits any fundamental limitations, it is always possible (though not always easy!) to extend or adapt the software for the new requirements. The OSS developer community is also typically accessible through issues and online forums, so that even though the software is not officially "supported" in the sense of commercial software, it is often easier to discuss details of the software's internal operations with the maintainers than with commercial software where developers are inaccessible.

The Pandata stack

The above considerations determined which libraries are considered to be a part of the Pandata stack. As illustrated in Figure 1, the stack consists of options for each of the major steps in a data-processing task. A finance task might involve files from efficient Parquet-format columnar data storage, accessed from Amazon S3 storage using fsspec file readers, into a Pandas data API, for data processing using Pandas calls plus some custom Numba-optimized analysis code, visualized using an interactive Bokeh plot returned from hvPlot, using Jupyter as a UI (figure 2). When moving to larger financial datasets or more complex analysis and processing, a Dask dataframe can be substituted for the Pandas dataframe, while keeping the rest of the tools the same, thus supporting distributed computation without requiring it. Similarly, a geoscience task typical for the Pangeo community might involve files stored in cloud-friendly Zarr multidimensional array storage, accessed using fsspec based on a specification in an Intake data catalog, into a lazy Dask-based xarray multidimensional data structure, with raster processing done by the Xarray-spatial library (not part of Pandata since it is domain specific, but built on the Pandata stack and using otherwise similar principles), with visualization using xarray's interface to Matplotlib for Dask data structures, and with computation and a UI provided by JupyterHub running in the cloud and providing access to Jupyter (figure 3).

As you can see from these examples, any particular problem solved using the Pandata stack can involve completely different underlying libraries, and because each of the Pandata libraries are designed to interoperate freely, users can select the appropriate library or libraries for their needs at each stage. We'll now look into each of the stages in a bit more detail to explain the options available to a researcher using Pandata tools to solve their particular research problems.

Data storage

Pandata tools are designed for working with data, whether it comes from a file on disk, a hardware device, remote cloud storage, or a database query. Many existing scientific file formats were designed before cloud computing and are *terrible* choices for efficient distributed computation. The ubiquitous comma-separated-value CSV columnar file format, for instance, can be orders of magnitude larger and slower than Parquet (`github.com/holoviz/datashader/issues/129`). CSV and other legacy formats also often require serial access to the whole file to reach any value in it, thereby preventing parallel reads by processors working on different parts of the task. Even relatively efficient older binary formats are typically not "chunked" in a way that makes it simple to access the data needed by any particular run or any particular processor in a large compute job.

Pandata tools can provide very efficient, scalable, end-to-end computations only if the data itself is stored in an efficient, chunk-addressable way. Parquet is a suitable well-supported chunked columnar format (for data arranged in rows consisting of differently typed columns), while zarr is a suitable chunked multidimensional array format (for n-dimensional arrays of typed values indexed by row, column, and other dimensions). DuckDB is a relatively new addition to Pandata that makes an underlying file type like CSV or Parquet act like a queryable database, which can be very efficient when accessing small parts or aggregations of large datasets, so other Pandata tools are starting to add support for working with DuckDB as well.

What if you need to access very large collections of data *not* already stored in an efficient chunked format like Parquet or Zarr? Thanks to kerchunk (see "Data access" below), it is possible to

Fig. 2: *Pandata stack applied to a finance problem*

scan such files to record the locations of each chunk of data stored in them, and from then on act as if the underlying file is in Zarr (or potentially Parquet) format. In this way, the Pandata stack can support access to a wide array of legacy binary file formats such as HDF/NetCDF, GRIB, FITS, and GeoTIFF, which with the appropriate Kerchunk driver can now support efficient scalable computation without servers having to maintain multiple copies of the underlying data.

Data access

Once you have data in a suitable supported file format, it needs to be located somewhere that your Python code can access it. The nearly invisible but powerful fsspec library provides flexible and efficient access to files wherever they might be located, whether that is on your local hard drive, on an FTP file server, on cloud storage like S3, on a website, in a zip file, or in any number of other possible locations. fsspec is now integrated into Pandas and Xarray, transparently providing access from within those libraries (below). As mentioned above, Kerchunk can optionally be used for data access to make an older file format be efficiently addressable for Pandata usage, once there are indexing plugins available for that particular (usually domain-specific) format. Kerchunk and fsspec are also supported by the Intake data-catalog system, which allows a research group or community to capture metadata about collections of datasets so that they can be accessed conveniently without having research code tied to the location or file type of the underlying data.

Data API

At a Python level, most users actually start here, by selecting an appropriate data application programming interface (API) for their work. Python libraries provide many possible APIs that are suitable for different types of data. Pandas (for columnar data) and Xarray (for multidimensional data) are both supported throughout the Pandata stack, with or without Dask (see below) to handle distributed and out-of-core computation. Pandata's other options provide access to other data structures (ragged arrays, for Awkward, and graphs/networks, for GraphBLAS) and/or other computational hardware (RAPIDS and CuPy for GPU architectures), with extensive (but not yet fully complete) support throughout the Pandata stack. Users or domain-specific library authors will typically pick one or more of these data APIs to cover the types of data being used in their domain or their specific problem, and then provide a path to the data to access it using the underlying data-access tools using the data formats and storage available.

Data processing

Once the data is accessible in Python using a data API, the actual computation can begin. Data processing covers many possible computations, but we will consider three main categories: (1) domain-general operations provided by the data API library directly, (2) small custom-compiled loops for arbitrary domain-specific computations at scale, and (3) separate domain-specific or other custom libraries. (1) First, each data API provides a wide variety of inbuilt data-processing routines for selecting, aggregating, and summarizing the data being accessed. For many data-processing workflows, such transformations are sufficient to complete the task, when combined with the rest of the Pandata stack, and using them preserves all the Pangeo qualities like scalability and easy visualization. (2) Second, sometimes the domain-specific code can be isolated to a few small loops with explicit numeric or string computations, now that Pandata covers so many of the other tasks. In such cases, these loops can easily be implemented in a notebook or script using the general-purpose Numba library along with Dask for optional scalability, again providing all of the benefits of Pandata without significant effort.

(3) The third category of computations typically requires leaving the Pandata stack, bringing in domain-specific libraries from the Python ecosystem that focus on the particular models and computations needed in a given domain. In some cases, you can find domain-specific libraries already designed to work well with the Pandata tools like Dask and Xarray, e.g. xarray-spatial (geoscience), dask-image (image processing), dask-ml (machine learning), icepyx (satellite data), panel-chemistry (chemistry), xdart (x-ray analysis), asari (metabolomics), and megspikes (brain imaging). Using a Pandata-compatible domain-specific tool helps ensure that all of the above properties like being scalable, visualizable, and cloud-friendly are maintained. However, in other cases there will be libraries available, but using them will introduce a bottleneck: not being scalable with Dask, not being compiled for efficient operation, not being vectorized to work with large datasets, being tied to a local GUI or local filesystem access, etc. In those cases, you can either try to ensure that the bottleneck only applies to infrequent usage patterns, or you can invest what may require substantial effort to rewrite the domain-specific libraries to work well with a Pandata approach. Rebuilding a particular domain's tools to be fully integrated with the Pandata stack takes time, requiring ambitious projects like building your simulator or analysis tool on top of Dask and Xarray or Pandas and ensuring that fully visualizable distributed objects are available for all stages of a workflow. Still, the focus of such efforts can be solely on the aspects specific to a particular research domain, and they can simply inherit all the capabilities of the Pandata stack (now and into the future) for what the Pandata stack covers.

Visualization

Each step in the complete data-processing pipeline pipeline from raw data to results requires human understanding and validation, often most effectively achieved using visualization tools. Most

Fig. 3: *Pandata stack applied to a geoscience problem*

Python visualization tools (see `pyviz.org` for a complete list) have limitations that prevent them from being suitable for the Pandata stack. Many of them are limited to relatively small amounts of data, lack support for the various Pandata data APIs, lack support for multidimensional arrays or unstructured grids for the fields that need those, or are tied to a desktop OS or GUI. All Pandata tools support basic static-image visualization using Matplotlib, and most also support fully interactive Bokeh- or Plotly-based plotting via hvPlot [13]. hvPlot supports the native `.plot` visualization calls provided by the data APIs, while adding support for efficient server-side distributed rendering using Datashader so that even the largest datasets can be visualized without subsampling or copying the data. Datashader can also be used by itself as a general-purpose scalable server-side rasterizer, turning data of any supported type (point, line, region, polygon, raster, etc.) into a regular grid of values suitable for further processing and combination with other data.

User Interface (UI)

The Jupyter Notebook is a now-ubiquitous domain-independent user interface for working with Python code, and all Pandata tools are fully usable from within Jupyter. Beyond just executing cells full of code, the Panel library in Pandata makes it simple to add interactivity to each Jupyter cell, using a widget to provide control over workflow parameters, and allowing tabular or graphical outputs to be arranged into dashboards or small applications that fit into a cell. Panel also lets users designate certain cells as being "servable" if the notebook code is deployed as a separate, standalone application with a UI independent of Jupyter. Panel thus supports the "interactive" and "shareable" qualities of the Pandata stack, ensuring that your work can have impact on others.

Jupyter focuses on a single researcher or user, but many Pandata-based projects involve collaborations among multiple team members and multiple institutions. Such projects typically use JupyterHub to provide a shared computing environment with the Pandata tools already installed and ready to run. Pandata-based projects often use Nebari (a declarative specification for infrastructure) to simplify configuring JupyterHub, Dask, and associated authentication on cloud servers.

Environments and reproducibility

A final category of tools underlying all the rest shown in figure 1 centers on how these libraries are packaged and put together into Python environments for solving any particular problem. Many different options are available for Python package and environment management, but Pandata tools are typically used with the conda package and environment manager, because it tracks binary dependencies between libraries, including the underlying C and Fortran code that is involved in the domain-specific libraries needed for any particular Pandata workflow. For a specific workflow, once the list of packages needed in the environment has been finalized, the anaconda-project tool (now being developed in a more general form as conda-project) allows the precise versions involved to be locked on every supported platform, achieving cross-platform reproducibility for a Pandata-based project (see [14] for details).

Examples

To help users understand how these tools fit together, a variety of example workflows based on Pandata tools can be found online. The website `examples.holoviz.org` includes research topics in different domains, with Pandata tools featured prominently in most of them and particularly in the Attractors, Census, Ship Traffic, and Landsat examples. The `holoviz.org` site has a tutorial that brings in many of the Pandata packages to solve an example research problem. The Pangeo Project Gallery at `pangeo.io` and Project Pythia at `projectpythia.org` both include detailed examples in the specific area of climate science. Each Pandata package also has its own documentation and website illustrating what it can do, often in combination with other Pandata tools, such as the Dask examples at `stories.dask.org`.

History and background

The Pandata libraries individually address important domain-independent problems for scientific research, engineering, and data analysis. As argued above, together they form a coherent and powerful basis for scientific computing in any discipline. Given that each tool has its own developers, separate management structures, and separate communities, how did it happen that together the tools add up to such a coherent approach? To understand this process, it is necessary to dive into the history of some of these projects and the connections between them.

First, many of the Pandata projects were either originally launched at Anaconda, Inc., or they were adapted by people who were at Anaconda at the time to work well with the other projects. Anaconda's consulting division (led by the author Bednar) and open-source tooling division (with most projects led by the author Durant) have worked with a wide variety of government agencies, private foundations, universities, and companies doing numerical computation and research. Each project is designed to address pain points being experienced by those collaborators, and the Anaconda staff involved help the projects work together to add up to complete solutions. Each new consulting project also extensively tests and validates each library both alone and in combination with the others, ensuring that these particular packages add up to complete solutions for each domain in which they are applied.

Specifically, the fastparquet, fsspec, kerchunk, intake, numba, dask, datashader, bokeh, hvplot, panel, and conda projects were

all created by developers working at Anaconda at the time, with funding from a very wide range of external grants and contracts but with Anaconda playing a central role in developing them and ensuring interoperability between them. These Anaconda-based developers have also contributed extensively to the other projects listed, such as xarray, pandas, awkward-array, graphblas, zarr, and jupyter. Anaconda as a company does not (and cannot!) control the overall set of projects involved, because each has their own contributors, governance structure, and stakeholders. But having a large collection of scientific software developers working at Anaconda on this wide range of projects exercising the code in many different research fields and domains has led naturally to the emergence of Pandata, to which this paper is only now assigning a name and describing the underlying principles guiding this work.

Out of the many external projects that Anaconda developers have been involved in, there is one that deserves a special mention because of how it catalyzed the collection of Pandata tools: Pangeo [15]. The `pangeo.io` project is a climate-science initiative based on bringing modern distributed cloud computing approaches to bear on large-scale modeling and analysis of the Earth's climate. Many of the Pandata tools are also Pangeo tools, and in fact the main distinction between Pandata and Pangeo is simply that Pangeo is domain specific, while Pandata is largely the same stack of tools as Pangeo but explicitly not being tied to any specific research domain (hence the name "Pandata", with Pangeo-style tooling but for any field using data, not just geoscience). Across numerous grants, contracts, and projects, Pangeo's researchers have improved each of the various tools in the Pandata stack and demonstrated how they can be put together to solve very challenging research problems, cost-effectively processing petabytes of climate and remote sensing data in a way enabled by Pandata tools like Dask and JupyterHub. This paper is an effort to publicize that the underlying tools are in fact fully domain general and applicable to *all* of science, with different data API and file format choices but largely the same tools used to cover general data-processing needs that span all research areas. Why start from scratch, when you can build on this battle-hardened, extensively validated set of general-purpose tools addressing much of what you need in any project?

Alternatives to Pandata

The libraries in Pandata are of course not the only alternatives available in Python; each library individually has alternatives that have their own strengths and advantages. However, the alternative libraries have not (yet?) been integrated with Pandata tools, making it much more difficult to apply them to a Pandata-based project. For instance, Ray [16] is an alternative approach to distributed computation that is not supported by these tools, and so if a project uses Ray to manage distributed computation, then they cannot (currently) easily select hvPlot for visualization without first converting the data structures into something hvPlot understands. Similarly, Vaex [17] and Polars [18] offer alternatives to the Pandas/Dask columnar dataframes supported in Pandata, and projects based on those data APIs will not (yet!) easily be able to use Pandata tools for visualization and user interfaces. There are also now alternative tools for server-side rendering of large datasets that in Pandata are handled by Datashader, such as VegaFusion for Vega and Altair, but those are not fully integrated with the other Pandata libraries like Dask and Numba.

There are also full alternative stacks to consider, such as tools like Hadoop [19] and Spark [20] from the Apache Foundation. Most Apache tools rely on the Java Virtual Machine (JVM) that provides OS-independent computation but requires a heavyweight runtime compared to the Pandata tools, making it awkward to combine most Apache tools with Pandata tools. Pandata already offers flexible support for distributed computation without the JVM overhead, making "big data" tools like Hadoop and Spark unnecessary for Pandata applications. Pandata does rely on the low-level Apache Parquet and Arrow projects, which have non-JVM implementations that fit well into the Pandata stack.

Future work

This paper is the first to describe Pandata as an entity or collection of tools. The specific tools we selected are those that in the opinion of the authors are well integrated with the other tools and provide a "mix and match" approach to putting together libraries to implement a particular analysis workflow. There is not currently any entity besides these authors and this paper that defines what Pandata is, which libraries are included, and what process to follow for a library to be included or excluded from Pandata. As Pandata evolves, it will be important to formalize some of these processes, similar to how the Pangeo organization and the HoloViz organization (of which hvPlot and Panel are a part) have been constituted with a independent steering committee and governance policies. Future Pandata directions and development can be tracked at `github.com/panstacks/pandata`, where people interested in Pandata can raise issues and discuss options and alternatives. In the meantime, Pandata is just a concept and an explanation for the work of hundreds of people working mostly independently but with key connections that make these independent projects come together into a coherent stack for performing scientific research.

Other relevant areas that need work include the development of domain-general reusable tools for capturing the metadata, conventions, consistent units, etc. needed in each domain, such as the `cfconventions.org` conventions for climate science. Furthermore, improved tools for collaboration are needed, such as implementing parts of the JupyterHub roadmap (`jupyterhub.readthedocs.io/en/stable/contributing/roadmap.html`). Lastly, better documentation and workflow demonstrations of the Pandata stack are required to facilitate easier onboarding, provide comprehensive usage guidelines, and showcase the full potential of Pandata in solving diverse scientific problems.

Conclusion

The Pandata stack is ready to use today, as an extensive basis for scientific computing in any research area and across many different communities. There are alternatives for each of the components of the Pandata stack, but the advantage of having this very wide array of functionality that works well together is that researchers in any particular domain can just get on with their actual work in that domain, freed from having to reimplement basic data handling in all its forms and freed from the limitations of legacy domain-specific stacks. Everything involved is open source, so feel free to use any of these tools in any combination to solve any problems that you have!

Acknowledgements

Thanks to Christian Capdeville, Deepak Cherian, Andrew Donoho, Kodie Dower, Tetsuo Koyama, Dillon Niederhut,

Dharhas Pothina, Demetris Roumis, Philipp Rudiger, Stan Siebert, Rich Signell, Hunt Sparra, Gene Trantham, and Sophia Yang, for comments and suggested revisions.

REFERENCES

[1] C. R. Harris, K. J. Millman, S. J. van der Walt *et al.*, "Array programming with NumPy," *Nature*, vol. 585, no. 7825, p. 357–362, 2020, https://doi.org/10.1038/s41586-020-2649-2. [Online]. Available: https://doi.org/10.1038/s41586-020-2649-2

[2] W. McKinney, "Data structures for statistical computing in Python," in *Proceedings of the 9th Python in Science Conference*. SciPy, 2010, https://doi.org/10.25080/majora-92bf1922-00a. [Online]. Available: http://conference.scipy.org/proceedings/scipy2010/mckinney.html

[3] J. D. Hunter, "Matplotlib: A 2D graphics environment," *Computing in Science & Engineering*, 2007, https://doi.org/10.1109/MCSE.2007.55. [Online]. Available: http://scitation.aip.org/content/aip/journal/cise/9/3/10.1109/MCSE.2007.55

[4] T. Kluyver, B. Ragan-Kelley, F. Pérez, B. Granger, M. Bussonnier, J. Frederic, K. Kelley, J. Hamrick, J. Grout, S. Corlay, P. Ivanov, D. Avila, S. Abdalla, and C. Willing, "Jupyter notebooks – a publishing format for reproducible computational workflows," in *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, F. Loizides and B. Schmidt, Eds. IOS Press, 2016, pp. 87 – 90.

[5] I. Stančin and A. Jović, "An overview and comparison of free Python libraries for data mining and big data analysis," in *2019 42nd International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, 2019, pp. 977–982, https://doi.org/10.23919/MIPRO.2019.8757088.

[6] N. Briscoe, "Understanding the OSI 7-layer model," *PC Network Advisor*, vol. 120, no. 2, pp. 13–15, 2000.

[7] S. Hoyer and J. Hamman, "xarray: N-D labeled arrays and datasets in Python," *Journal of Open Research Software*, vol. 5, no. 1, 2017, https://doi.org/10.5334/jors.148. [Online]. Available: https://doi.org/10.5334/jors.148

[8] T. Handy, "The modern data stack: Past, present, and future," *dbt blog*, 2020, accessed 2023-07-08. [Online]. Available: https://blog.getdbt.com/future-of-the-modern-data-stack/

[9] T. Jaipuria, "Understanding the modern data stack," *Substack*, July 2022, accessed 2023-07-08. [Online]. Available: https://tanay.substack.com/p/understanding-the-modern-data-stack

[10] S. K. 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*, 2015, pp. 1–6.

[11] S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D. S. Seljebotn, and K. Smith, "Cython: The best of both worlds," *Computing in Science & Engineering*, vol. 13, no. 2, pp. 31–39, 2011.

[12] Dask Development Team, *Dask: Library for dynamic task scheduling*, 2016. [Online]. Available: https://dask.org

[13] P. Rudiger, M. Liquet, J. Signell, S. H. Hansen, J. A. Bednar *et al.*, "holoviz/hvplot: Version 0.8.4," Jun. 2023, https://doi.org/10.5281/zenodo.8009640. [Online]. Available: https://doi.org/10.5281/zenodo.8009640

[14] J. A. Bednar, "8 levels of reproducibility: Future-proofing your Python projects," *Anaconda Maker Blog*, June 2012, www.anaconda.com/blog/8-levels-of-reproducibility. [Online]. Available: https://www.anaconda.com/blog/8-levels-of-reproducibility

[15] T. E. Odaka, A. Banihirwe, G. Eynard-Bontemps, A. Ponte, G. Maze, K. Paul, J. Baker, and R. Abernathey, "The Pangeo ecosystem: Interactive computing tools for the geosciences: Benchmarking on HPC," in *Tools and Techniques for High Performance Computing*, G. Juckeland and S. Chandrasekaran, Eds. Springer International Publishing, 2020, pp. 190–204, https://doi.org/10.1007/978-3-030-44728-1_12. [Online]. Available: https://doi.org/10.1007/978-3-030-44728-1_12

[16] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I. Jordan, and I. Stoica, "Ray: A distributed framework for emerging AI applications," 2018.

[17] M. A. Breddels and J. Veljanoski, "Vaex: Big data exploration in the era of Gaia," *Astronomy & Astrophysics*, vol. 618, p. A13, Oct 2018, https://doi.org/10.1051/0004-6361/201732493. [Online]. Available: https://doi.org/10.1051%2F0004-6361%2F201732493

[18] R. Vink, "Polars: Lightning fast dataframe library for Rust and Python," 2023, accessed: 2023-07-08. [Online]. Available: https://www.pola.rs/

[19] T. White, *Hadoop: The Definitive Guide*, 4th ed. O'Reilly, 2015.

[20] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin *et al.*, "Apache Spark: A unified engine for big data processing," *Communications of the ACM*, vol. 59, no. 11, pp. 56–65, 2016.

Spatial Microsimulation and Activity Allocation in Python: An Update on the Likeness Toolkit

Joseph V. Tuccillo[‡*], James D. Gaboardi[‡]

Abstract—Understanding human security and social equity issues within human systems requires large-scale models of population dynamics that simulate high-fidelity representations of individuals and access to essential activities (work/school, social, errands, health). Likeness is a Python toolkit that provides these capabilities for Oak Ridge National Laboratory's (ORNL) UrbanPop spatial microsimulation project. In step with the initial development phase for Likeness (2021 - 2022), we built out several foundational examples of work/school and health service access. In this paper, we describe expansion and scaling of Likeness capabilities to metropolitan areas in the United States. We then provide an integrated demonstration of our methods based on a case study of Leon County, FL and perform validation exercises on 1) neighborhood demographic composition and 2) visits by demographic cohorts (gender/age) obtained from point of interest (POI) footfall data for essential services (grocery stores). Taking into account lessons learned from our case study, we scope improvements to our model as well as provide a roadmap of the anticipated Likeness development cycle into 2023 - 2024.

Index Terms—activity space, synthetic population, microsimulation, population dynamics

Introduction

Agent-based models (ABMs) of population dynamics are essential for understanding human security and social equity issues within human systems [1], [2], [3]. Such models often rely upon *synthetic populations* – virtual representations of individuals plausibly residing within an area – to assess how individual behaviors and interactions contribute to complex system-level behavior. Applications of ABMs for population dynamics range from urban planning (access to essential services like food and healthcare), public health (facility occupancy, social contact networks), and disaster preparedness (social vulnerability to environmental hazards, evacuation and critical infrastructure planning).

A current challenge for research in population dynamics is to more directly represent population heterogeneity [4]. Individuals exhibit a variety of patterns of life dictating their behavior and interactions [5], which are in turn influenced by social, demographic, and economic characteristics. Incorporating these factors into the design of synthetic populations contributes to more holistic models of how human systems operate.

* *Corresponding author: tuccillojv@ornl.gov*
‡ *Oak Ridge National Laboratory, Geospatial Sciences and Human Security*

To support enhanced ABMs of population dynamics, the UrbanPop spatial microsimulation framework developed by Oak Ridge National Laboratory (ORNL) generates high-fidelity synthetic populations on hundreds of attributes from the American Community Survey (ACS) and its Public-Use Microdata Sample (PUMS) and combines them with nighttime/daytime behavioral routines [3]. Central to UrbanPop's capabilities is Likeness, a Python toolkit that combines population synthesis, spatial network modeling, and activity allocation [6], [7]. Foundational examples produced for Likeness have explored travel routing from home locations to anchor activities (e.g., work, school), POI occupancy characteristics [6], [7], as well as routing incidental travel from anchor to non-anchor activities (e.g., social, errands, health) [8]. This paper builds from these foundational examples to discuss scaling our methods to larger areas of interest. Moreover, it examines the performance of our activity allocation model relative to real-world POI demographics from digital trace (anonymized mobile device) data on POI visitation.

Expansion and Scaling of Likeness Capabilities

Synchronous with creating foundational examples for Likeness, we have developed an integrated workflow that scales the ecosystem (Figure 1) to support microsimulation for any metropolitan statistical area (MSA) in the United States.

Figure 2 outlines the procedure for agent generation at the MSA level. As discussed in [6], the `livelike.acs.puma` class is the core object for residential population synthesis. It stores all census microdata and geographic (e.g., block group, tract) model constraints to support spatial allocation for a single Public-Use Microdata Area (PUMA) in the United States. Residential synthetic populations for MSAs are generated as collections of PUMAs, which are parsed automatically by combining U.S. Census metropolitan/micropolitan delineation files[1] with the Census 2010 PUMA-to-tract relationship file[2]. PUMAs parsed for the target MSA are converted into `livelike.acs.puma` objects in bulk via `livelike.multi.make_pumas()`. This function accepts the PUMA Federal Information Processing Standards (FIPS) codes (unique IDs) in list form, along with the target ACS year. This information is then used to gather the relevant ACS Summary File (SF) constraints for spatial allocation of PUMS responses to small census areas of roughly 8,000 people

1. https://www.census.gov/geographies/reference-files/time-series/demo/metro-micro/delineation-files.html
2. https://www.census.gov/programs-surveys/geography/technical-documentation/records-layout/2010-tract-to-puma-record-layout.html

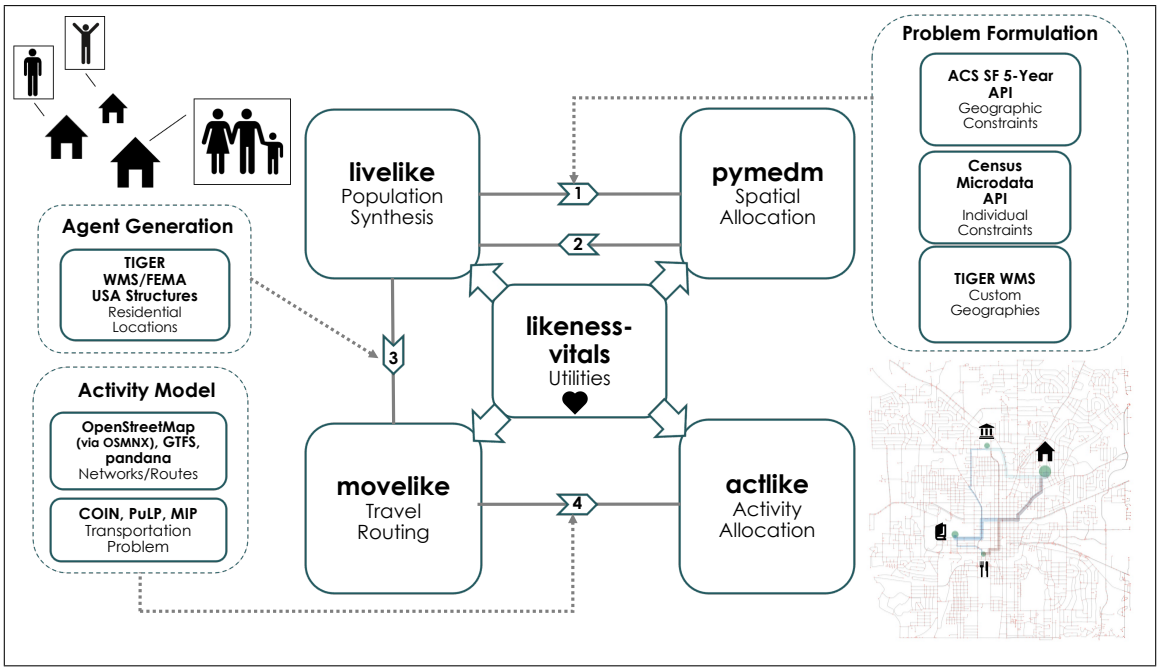

Fig. 1: *Likeness ecosystem overview.*

or less (block groups, tracts). Spatial allocation is subsequently handled with parallel processing, which is supported by both packages Likeness offers for Penalized Maximum-Entropy Dasymetric Modeling (P-MEDM) [9]: `pymedm` (bleeding-edge, Python-native version, based on `jaxopt` [10]) and `pmedm_legacy` (stable bridge to original R/C++ routine[3] via `rpy2`). As demonstrated in [6] and [7], the population synthesis routine also collects diagnostics on how effectively each P-MEDM solution reproduces variable estimates from the ACS SF relative to reported Margins of Error (MOEs). These utilities are available in both `pymedm` and `pmedm_legacy`.

Likeness generates agents for microsimulation in a way that provides realistic home (origin) locations from which to allocate essential activities on transportation networks. Our initial approach, based on census block-level housing density [6], [7], is now implemented in `livelike` as a housing universe generation procedure. Additionally, we are actively developing a method (demonstrated in Integrated Demonstration: Leon County, Florida) that enhances this capability by matching synthesized households to residential locations from building footprint data. These matches are performed by conflating attributes of synthetic households (e.g., dwelling type, income) with building footprint attributes (e.g., floor area, number of units). These utilities are designed to be agnostic to the building footprint provider and can even support custom building features.

Agent residential locations provided by `livelike` act as origin points for simulating travel to essential activities. The next stage in the Likeness workflow employs network analysis to model the cost of travel to these activities [11], [12], [13] and allocate agents to POIs accordingly based on mathematical programming routines [14], [15], [16], [17][4]. In our first iteration of Likeness, both these tasks were accomplished within `actlike`. However, we concluded that the network modeling piece was specialized enough to be split from the `actlike` package, which led to the creation of `movelike`. With a push for varied modes of network traversal, three new modes of travel can now be modeled: walking, biking, and public transit. However, modeling travel behavior via public transportation is less straightforward than for driving, biking, and walking networks due to stricter network topology, including factors like connectivity and directionality of routes. We have made our foray into modeling more realistic public transit behavior within `movelike` through the incorporation of the General Transit Feed Specification (GTFS)[5]. GTFS is a data specification that stipulates the required files, along with their structure and format[6], for publishing, ingesting, and utilizing public transit datasets. The GTFS datasets can be obtained via services such as *TransitFeeds*[7] and *The Mobility Database Catalogs*[8]. In our current iteration we utilize GTFS data feeds to implement a pseudo-transit network space by which agents can engage in limited traversal. This is accomplished through a mask of *OpenStreetMap*[9] (OSM) street segments known to be associated with bus routes. The OSM network is masked by passing a (multi)polygon feature of buffered and unioned bus routes within the study area into `osmnx` [12]. This method demonstrates progression in representing public transit but certainly has room for improvement, which will be discussed in Development Roadmap: 2023 - 2024.

Finally – and at the heart of it all – expansion and scaling of the Likeness ecosystem led to the development of a new package for common utilities, `likeness-vitals`. The `likeness-vitals` package provides support for monitoring

3. https://bitbucket.org/nnnagle/pmedmrcpp
4. The PuLP and Python-MIP open-source optimization Python packages are cited here, along with COIN-OR (a consortium that supports various open-source Operations Research projects) and the COIN-OR Branch-and-Cut solver.
5. https://gtfs.org/
6. https://gtfs.org/schedule/reference/#dataset-files
7. https://transitfeeds.com/
8. https://github.com/MobilityData/mobility-database-catalogs
9. https://www.openstreetmap.org/

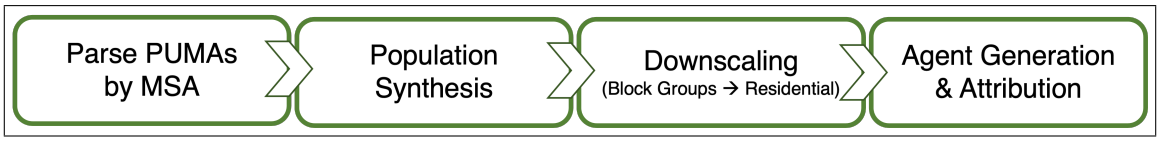

Fig. 2: *Likeness agent generation procedure for Metropolitan Statistical Areas (MSAs) in the United States.*

and timing processes, data manipulation, shared spatial functionality, and Census API access.

Integrated Demonstration: Leon County, Florida

Following the workflow described in Section Expansion and Scaling of Likeness Capabilities, we demonstrate the current capabilities of Likeness and validate our activity allocation routine for Leon County, Florida. Leon County, whose primary city is Tallahassee, features a population of just under 300,000 residents, a compact urban footprint, and a diverse array of transportation modes (driving, transit, bike, walking). Our mobility validation exercise is based on grocery store visits from simulated home locations. Grocery stores provide a useful test case, acting both as catchments for the general population as well as points of access to vital services including food and healthcare. We obtained grocery store visits from Foursquare's Research Visits feed, which provides POI visitation data attributed by demographic cohort (gender by age) for a variety of facility types[10,11].

We first simulated a single synthetic population for the Tallahassee Core-Based Statistical Area (CBSA). Spatial allocation (P-MEDM) was constrained on variables across subjects including sampling universe totals (i.e., population, housing units, households), descriptive factors including demographics, socioeconomic status, housing, mobility, and worker and student characteristics. For the remainder of the analysis, we focused on Leon County alone, removing large outlying areas of the MSA (PUMA `1206300`, "Apalachee Region (Outside Leon County)"). Agents[12] were generated with the Federal Emergency Management Agency's (FEMA) open USA Structures database [18], [19]. Our allocation procedure leveraged `livelike` utilities to match synthetic households to single-family residential, multi-family residential, mobile homes, and group quarters housing types.

We used employment and travel mode characteristics to assign agents to transportation networks used to access grocery POIs from home locations. Agents labeled as 'employed' possess an associated flag that identifies reported commute mode that can take the following values: 'car_truck_van', 'bicycle', 'walked', 'wfh', 'public_transportation', 'other', and 'motorcycle'. Because detailed travel modes are unavailable for agents that are not employed (e.g., retired, active military), we rely on private (i.e., household-level) vehicle ownership instead.

Travel modes assigned to each agent were conflated with four transportation network types: 'walked', 'bicycle', 'public_transportation', and 'drive'. We supported this process by developing a decision tree, visualized in Figure 3, through which we assume:

- Employed agents will use the transportation network that best matches their commute mode to access grocery POIs.
- Agents that are not employed will use a privately-owned vehicle, and thus the 'drive' network, to access grocery POIs when available.
- Agents that are not employed and lack a privately-owned vehicle will use public transportation if they are located in a block group that is served by Tallahassee's bus network (StarMetro) and opt to walk otherwise.

As demonstrated in Table 1, the 'drive' network supports the overwhelming majority of travel to grocery POIs in Leon County, followed by walking, public transportation, and bicycle access.

TABLE 1: *Householder agents >20yo per assigned travel mode*

Mode Assignment	Agent Count
`'walked'`	5,325
`'bicycle'`	1,052
`'public_transportation'`	3,830
`'drive'`	101,681
	111,888

Figure 4 shows that Leon County's agent population is distributed unevenly relative to assigned travel modes. Because Leon County's infrastructure primarily supports travel by car, agents who drive are distributed closest to the area's general population density. The spatial distribution of agents who travel by walking also tends to follow Leon County's settlement patterns, though in more limited numbers than for those who drive. Agents using public transport, meanwhile, are largely present in and near the center of the county, roughy occupying denser urban areas where StarMetro service is available. Bicyclists are distributed similarly to bus takers, but with several individual clusters associated with smaller outlying towns and settlements.

Grocery store POIs with medium to high visit confidence (at least 30 device visits per month, $n = 53$) were obtained from Foursquare for Leon County in January 2023. Destination capacities were estimated based on visit counts weighted by representativeness of the demographic cohort within the state's 2010 Census population[13]. Destination capacities were estimated by the daily average (mean) for each POI during the collection month (01/2023).

After travel modes were assigned to agents, four network cost matrices were calculated from origin (residential location) to destination (grocery store) POIs in `movelike`. Agents were then allocated to a single probable destination POI based on least cost network travel paths with the `actlike.ActivityAllocation` routine, which solves a modified Transportation Problem[14] [20], [21], [22], [23], where destination POI capacities are scaled [24] by the proportion of assigned travel mode for each scenario. All

10. https://location.foursquare.com/places/docs/how-does-places-work
11. https://location.foursquare.com/visits/docs/research-feed-schema
12. Agents less than 20 years old were not included in the Foursquare POI data, thus they are excluded from our analysis.
13. https://location.foursquare.com/visits/docs/foursquare-data-normalization
14. This model is formulated in [6].

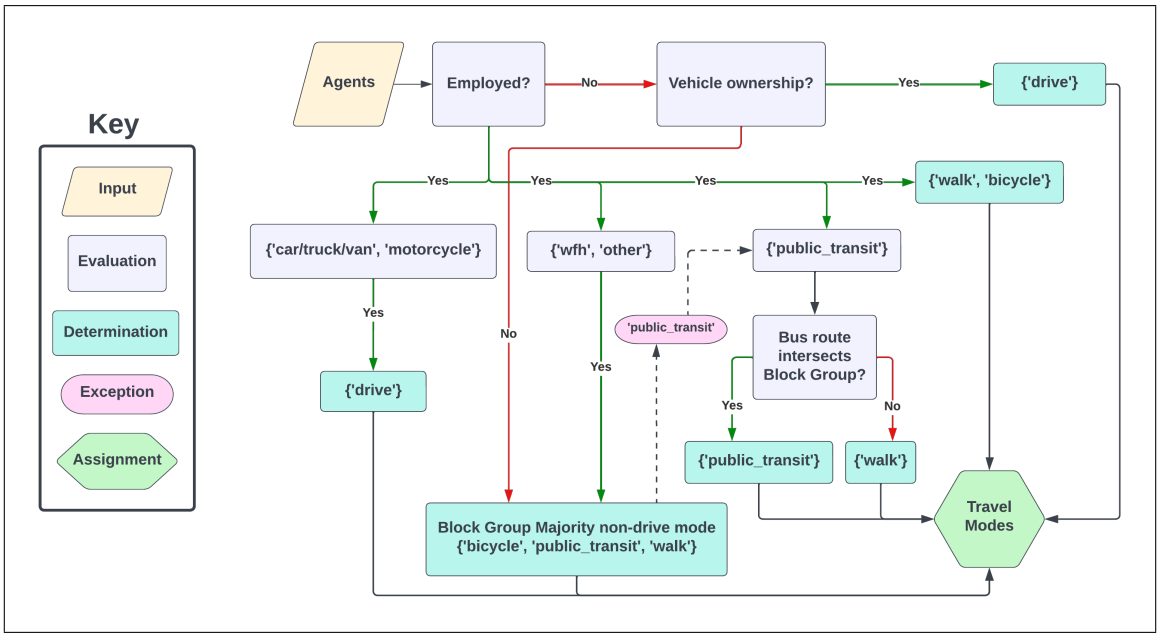

Fig. 3: *Agent travel mode assignment decision tree.*

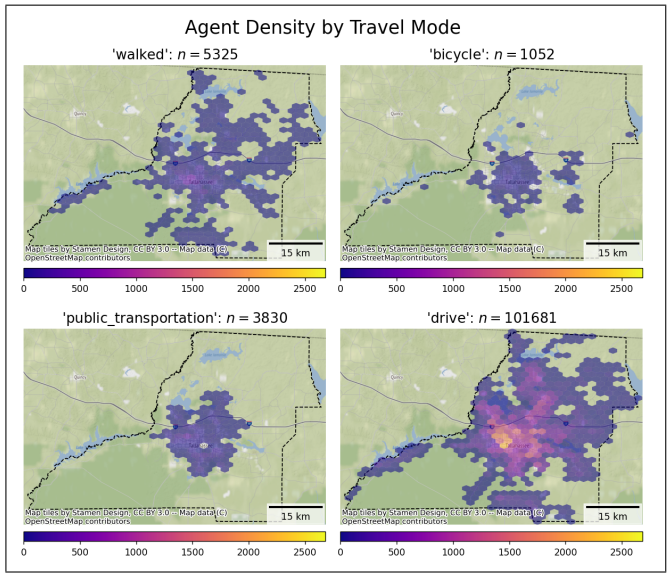

Fig. 4: *Synthetic population distribution by travel mode.*

models were run consecutively on two machines for benchmarking purposes. These were:

- A personal laptop (macOS) with a 2.3 GHz Quad-core Intel Core i7 processor (32 GB RAM).
- A virtual machine (Ubuntu) with a 2.8 GHz 22-core Intel(R) Xeon(R) processor (86 GB RAM).

The large disparity in problem size seen in Table 1 is even more pronounced in solution runtimes, shown in Table 2. Optimal solutions for non-drive models were found in a maximum time of just over 1 minute on both machines, with the drive model taking more than 17 and 8 hours to solve on the macOS and Ubuntu machines, respectively. Considering the solution time for the drive scenario, there is clearly a need for a more effective solution technique, which will be further discussed in Development Roadmap: 2023 - 2024.

TABLE 2: *Allocation Solution Runtimes (min.)*

Mode Assignment	**macOS** 4 cores 32 GB RAM	**Ubuntu** 22 cores 86 GB RAM
`'walked'`	0.72	1.19
`'bicycle'`	0.05	0.26
`'public_transportation'`	0.48	0.69
`'drive'`	1061.43	483.94

Validation procedures

Our validation procedures were designed to quantify the degree to which Likeness 1) resembles reference population estimates provided by the ACS SF with the synthetic populations (**demographic validation**) and 2) allocates activities matching real-world visitation patterns by demographics segments captured by the Foursquare POI data (**mobility validation**).

To produce our demographic validation, we followed [3] and measured our synthetic populations' degrees of conformity with 90% Margins of Error (MOEs) available from the ACS SF. ACS MOEs provide bounds for the expected ranges of values that our variables of interest could take. Tabulating individual attributes within each block group's synthetic population results in a reconstruction of the ACS SF estimates that can be assessed against the 90% MOEs. Greater conformity with the MOEs ("MOE Fit Rate") indicates a synthetic population that could plausibly resemble that block group's "true" population.

Following [3] and [6], we ran our mobility validation using Canonical Correlation Analysis (CCA). CCA, which measures the degree of linear association between two multidimensional datasets [25], is necessary to compare visitation patterns (n grocery store locations by m demographic cohorts). We performed CCA

on both the between-destination (relative prevalence) and within-destination (compositional) characteristics of each POI by demographic group. Both CCA runs were generated from tabulated counts of trips from the observed and synthetic datasets, their key difference being the method of standardization (column-wise for between-destination, row-wise for within-destination). We used the CCA coefficient of determination (R^2) to measure associations between synthetic and observed results. To better understand POI-specific activity allocation performance, we generated an additional local measure of within-destination correspondence. The local within-destination statistic compares the relative sizes of demographic cohorts using Spearman Rank Correlation, a non-parametric measure of the association between the ranks of two variables [26]. We opted for Spearman Correlation due to the relatively small number of demographic cohorts (n = 11).

Results

Demographic Validation

TABLE 3: *Demographic validation*

PUMA	Name	ACS 90% MOE Fit Rate
1206300	Leon County (Central)	0.992
1207300	Leon County (Outer)	0.998
1207301	Apalachee Region (Outside Leon County)	0.994

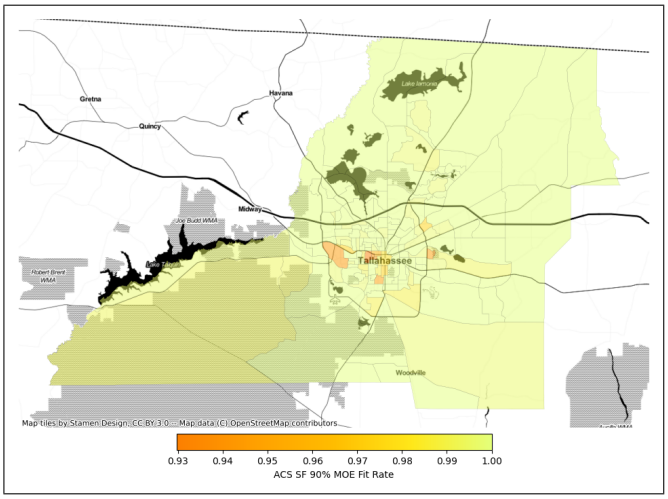

Fig. 5: *Local demographic validation for Leon County.*

Each of the three P-MEDM runs (one for each PUMA in the Tallahassee CBSA) resulted in a synthetic reconstruction of the ACS 90% MOEs. Overall, these reconstructions matched with at least 99% of the published MOEs from the ACS SF (Table 3. At the more granular level of Leon County block groups (Figure 5), MOE Fit Rates were relatively lower but still in agreement with at least 90% of the ACS SF MOEs in each location. We observed diminished performance in areas with large group quarters populations (e.g., college dormitories, prisons), as well as more sparsely populated rural and peri-urban portions of Leon County.

Mobility Validation

Our mobility simulation more faithfully recreated between-destination demographics (R^2 = 0.74) than within-destination

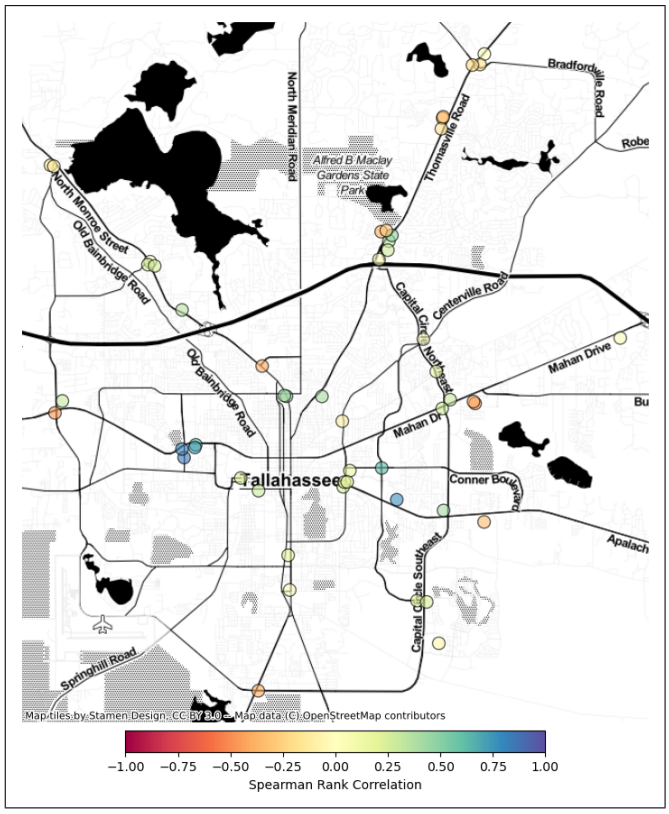

Fig. 6: *Local within-POI mobility validation.*

demographics (R^2 = 0.41) (Table 3). Inspecting the local within-destination scores, mapped in Figure 6, we observed generally greater correspondence between synthetic and observed POI demographics near Tallahassee's downtown core, Florida State University, and Florida A&M University, with diminished performance in suburban and outlying areas of Leon County.

It is difficult to pinpoint the inconsistency in recreating POIs' visitation patterns. In addition to urban density, this is potentially related to diversity of travel modes near the urban core (the increased biking, walking, and public transit use presented in Figure 4), but it requires further investigation. Section Limitations provides some additional confounding factors that are worthy of exploring relative to these results.

Limitations

We found that overall, Likeness approximates travel to non-anchor (grocery store) POIs modestly well. However, its performance tends to be weaker relative to travel to anchor activities (work/school), demonstrated in [6]. This suggests that approximating destination capacities for activities like grocery store visits provide an added challenge for activity allocation, including when real-world observations from visitation data are used. Additionally, multiple confounding factors related to data inputs and model assumptions may have affected our results:

- **Data-Specific:** Our clearest challenge is temporal mismatch between the synthetic population (2019) and POI visit data (2023). We have yet to increment our ACS year due to issues of non-conforming geography between the 2010 PUMAs and 2020 block groups/tracts. We hope to explore solutions to this problem starting with the

forthcoming 2022 ACS releases, which will adopt 2020 PUMAs across all geographic levels[15]. We were also limited to only one month of POI visit data. In future work, we hope to leverage a longer period of record to account for POIs with consistently high rates of visitation.

- **Model-Specific:** Several large assumptions were made that could confound our results, particularly that 1) agents simultaneously travel to grocery stores, 2) agents only select one grocery store to visit, using only one mode of travel, and 3) travel to grocery stores only occurs between home and work. These assumptions can be updated by incorporating information from time-use and travel surveys into Likeness. In this way, we can better reflect the times of day that different demographic cohorts access various activities [27].

We also hope to tighten our assumptions about the feasibility of POI access relative to the various travel modes. For example, in our current assignment process (Figure 3) agents unmatchable to a defined travel mode were considered walkers as a fallback for unavailability of a reachable bus route. Future iterations will refine this decision process by considering a distance threshold for agents to be labeled as either walkers or transit users. For example, we could set a rule that walking agents must be located within a reasonable distance of the closest POI, while agents that use bus service should reside near a bus stop in addition to being close to a bus route. To ensure all agents have at least one feasible POI destination to access, we also plan to incorporate a greater variety of curated locations from ORNL's PlanetSense database [28].

Conclusion and Outlook

This paper presented enhancements and scaling approaches for the Likeness spatial microsimulation toolkit. These include batched population synthesis runs for MSAs in the United States, residential allocation, and large-scale transportation network generation. We demonstrated these new capabilities by developing a mobility validation exercise for Leon County, FL. Our results provided reasonable representation of neighborhood demographics and routing to nearby essential services (grocery stores), with more mixed results related to activity allocation. The activity allocation results, however, do provide new research directions that we plan to explore in our future work. These include temporality of POI travel (travel probabilities relative to demographic cohorts), behavioral factors (willingness to travel given cost and impedances), and the use of multiple travel modes to reach activities.

Given the relative success of our population synthesis procedures in Section Integrated Demonstration: Leon County, Florida, we are interested in also applying Likeness to explore transportation equity in the context of access to essential services like food and healthcare. Such an approach would aim to enhance existing research on transportation and accessibility [29], [30] with a cross-sectional representation of social, demographic, housing, and mobility characteristics. Assuming an agent with some blend of socio-demographic and economic characteristics resides in a particular section of a neighborhood, how many essential services can be readily accessed using their assigned transportation network? Using the Likeness ecosystem, we could develop such a measure for all agents in a synthetic population, allowing the comparison of accessibility to population metrics like mobility difficulty. These insights could in turn be used to guide urban/regional infrastructure planning, pinpointing areas where drive, transit, or bike/walk services could be improved or expanded.

Development Roadmap: 2023 - 2024

- **Tooling for custom geographic extents.** The MSA-specific workflow demonstrated in this paper is limited in that it does not support custom geographic extents. This prevents analysis, for example, of predominantly rural areas. We are actively developing an approach to create residential synthetic populations for custom areas of interest (AOIs), supported by USA Structures. This functionality will also support the development of synthetic populations with national-scale coverage.
- **Open-sourcing core packages.** Though we have yet to meet our goal of open-sourcing the suite of Likeness packages [6], we are still on track to release the core packages for residential population synthesis, `pymedm` and `livelike`, in 2023. Releases of `movelike` and `actlike` are likely to follow in 2024.
- **Packaging schema.** A further consideration related to open-sourcing is whether we should migrate from a confederated toolkit schema where each module is a semi-independent Python package, as is seen in the modern implementation of PySAL [31], to a single monolithic Python package with submodules. Each schema has benefits, and this decision will require much consideration. With regards to the current confederated schema, the main benefit is modularity and reduced burden for continuous integration testing runtimes. This primary benefit is from a developer standpoint. However, providing a single package to install and use is a clear benefit to the user.
- **Consolidating visualization functionality.** We are in the process of consolidating functionality related to the visualization of input, processing, and results that have been used in an ad-hoc manner in the past. An initial push will be made for the inclusion of "made-to-order" population density hexbin plots and network-space allocation routes.
- **Improving mobility modeling.** Modeling public transit is a key area where we intend to develop increasingly more realistic agent "choices." As stated previously in Expansion and Scaling of Likeness Capabilities, there is significant potential in exploring further integration of GTFS data for locally-accurate modeling.
- **Optimization bottleneck.** As demonstrated in Results, there is a clear hit in computational performance and runtime when solving `actlike.ActivityAllocation` problems on increasingly larger model instances (e.g., more agents and more POIs). There are two paths to resolving this issue (which may be considered in concert): 1) Reviewing our modified Transportation Problem mixed-integer program (formulated in [6]); and 2) Utilizing a new underlying solver engine, such as HiGHS[16] [32]. In reviewing our formulation we will first investigate the potential for generating fewer constraints in the model. Following this, as stated above, we may consider formulating the model in a new solver. Depending on the outcome

15. https://www.census.gov/programs-surveys/acs/news/data-releases/2022/release.html

16. https://highs.dev/

of these experiments we may consider a new underlying optimization problem or implement a heuristic.

Acknowledgements

The authors would like to thank Ty Frazier for his contributions to scoping the incorporation of General Transit Feed Specification (GTFS) data during an earlier phase of this project.

The authors would also like to thank the two volunteer reviewers for their constructive feedback and the SciPy Proceedings editorial team for their ongoing support.

Notice: Research reported in this publication was supported by the National Center For Advancing Translational Sciences of the National Institutes of Health under Award Number UL1-TR001409, KL2-TR001432 & TL1-TR001431. The content is solely the responsibility of the authors and does not necessarily represent the official views of the National Institutes of Health.

This manuscript has been authored by UT-Battelle, LLC under Contract No. DE-AC05-00OR22725 with the U.S. Department of Energy. The United States Government retains and the publisher, by accepting the article for publication, acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this manuscript, or allow others to do so, for United States Government purposes. The Department of Energy will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan (http://energy.gov/downloads/doe-publicaccess-plan).

References

[1] T. C. Germann, M. Z. Smith, L. R. Dauelsberg, G. Fairchild, T. L. Turton, M. E. Gorris, C. W. Ross, J. P. Ahrens, D. D. Hemphill, C. A. Manore, *et al.*, "Assessing k-12 school reopenings under different covid-19 spread scenarios–united states, school year 2020/21: A retrospective modeling study," *Epidemics*, vol. 41, p. 100632, 2022. https://doi.org/10.1016/j.epidem.2022.100632.

[2] J. Ozik, J. M. Wozniak, N. Collier, C. M. Macal, and M. Binois, "A population data-driven workflow for covid-19 modeling and learning," *The International Journal of High Performance Computing Applications*, vol. 35, no. 5, pp. 483–499, 2021.

[3] J. V. Tuccillo, R. Stewart, A. Rose, N. Trombley, J. Moehl, N. N. Nagle, and B. Bhaduri, "UrbanPop: A spatial microsimulation framework for exploring demographic influences on human dynamics," *Applied Geography*, vol. 151, p. 102844, 2023. https://doi.org/10.1016/j.apgeog.2022.102844.

[4] United States Department of Energy, "Foundational Science for Biopreparedness and Response: Report from the March 2022 Roundtable." https://science.osti.gov/-/media/Initiatives/pdf/Biopreparedness_Roundtable_Report_092722.pdf, Mar. 2022. https://doi.org/10.2172/1868508.

[5] L. Pappalardo, F. Simini, S. Rinzivillo, D. Pedreschi, F. Giannotti, and A.-L. Barabási, "Returners and explorers dichotomy in human mobility," *Nature communications*, vol. 6, no. 1, p. 8166, 2015. https://doi.org/10.1038/ncomms9166.

[6] J. V. Tuccillo and J. D. Gaboardi, "Likeness: a toolkit for connecting the social fabric of place to human dynamics," in *Proceedings of the 21st Python in Science Conference* (M. Agarwal, C. Calloway, D. Niederhut, and D. Shupe, eds.), pp. 125–135, 2022. https://doi.org/10.25080/majora-212e5952-014.

[7] J. V. Tuccillo and J. D. Gaboardi, "Likeness: a toolkit for connecting the social fabric of place to human dynamics." https://doi.org/10.25080/majora-212e5952-02d, Aug. 2022. https://doi.org/10.25080/majora-212e5952-02d.

[8] J. D. Gaboardi and J. V. Tuccillo, "Spatial Microsimulation and Activity Allocation for Examining COVID-19 Vaccine Access Profiles," Mar. 2023. https://doi.org/10.5281/zenodo.7768810.

[9] N. N. Nagle, B. P. Buttenfield, S. Leyk, and S. E. Spielman, "Dasymetric modeling and uncertainty," *Annals of the Association of American Geographers*, vol. 104, no. 1, pp. 80–95, 2014.

[10] M. Blondel, Q. Berthet, M. Cuturi, R. Frostig, S. Hoyer, F. Llinares-López, F. Pedregosa, and J.-P. Vert, "Efficient and Modular Implicit Differentiation." arXiv preprint arXiv:2105.15183, 2021.

[11] OpenStreetMap contributors, "Planet dump retrieved from https://planet.osm.org." https://www.openstreetmap.org, 2023.

[12] G. Boeing, "OSMnx: New methods for acquiring, constructing, analyzing, and visualizing complex street networks," *Computers, Environment and Urban Systems*, vol. 65, pp. 126–139, Sept. 2017. https://doi.org/10.1016/j.compenvurbsys.2017.05.004.

[13] F. Foti and P. Waddell, "A Generalized Computational Framework for Accessibility: From the Pedestrian to the Metropolitan Scale," in *Transportation Research Board Annual Conference*, pp. 1–14, 2012.

[14] S. Mitchell, M. O'Sullivan, and I. Dunning, "PuLP: A Linear Programming Toolkit for Python." https://optimization-online.org/?p=11731, 2011.

[15] H. G. Santos and T. A. Toffolo, "Mixed Integer Linear Programming with Python." https://www.python-mip.com/, 2020.

[16] R. Lougee-Heimer, "The Common Optimization INterface for Operations Research: Promoting open-source software in the operations research community," *IBM Journal of Research and Development*, vol. 47, no. 1, pp. 57–66, 2003. https://doi.org/10.1147/rd.471.0057.

[17] J. Forrest, T. Ralphs, H. G. Santos, S. Vigerske, J. Forrest, L. Hafer, B. Kristjansson, jpfasano, EdwinStraver, M. Lubin, Jan-Willem, rlougee, jpgoncal1, S. Brito, h-i gassmann, Cristina, M. Saltzman, tosttost, B. Pitrus, F. Matsushima, and to st, "coin-or/cbc: Release releases/2.10.10," Apr. 2023. https://doi.org/10.5281/zenodo.7843975.

[18] H. L. Yang, J. Yuan, D. Lunga, M. Laverdiere, A. Rose, and B. Bhaduri, "Building extraction at scale using convolutional neural network: Mapping of the United States," *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, vol. 11, no. 8, pp. 2600–2614, 2018. https://doi.org/10.1109/jstars.2018.2835377.

[19] H. L. Yang, M. Tuttle, M. Laverdiere, T. Hauser, B. Swan, E. Schmidt, J. Moehl, A. Reith, J. McKee, and M. Whitehead, "ORNL USA Structures 2022." https://figshare.com/collections/ORNL_USA_Structures_2022/6139131/2, Sept. 2022. https://doi.org/10.6084/m9.figshare.c.6139131.v2.

[20] F. L. Hitchcock, "The Distribution of a Product from Several Sources to Numerous Localities," *Journal of Mathematics and Physics*, vol. 20, no. 1-4, pp. 224–230, 1941. https://doi.org/10.1002/sapm1941201224.

[21] T. C. Koopmans, "Optimum Utilization of the Transportation System," *Econometrica*, vol. 17, pp. 136–146, 1949. https://doi.org/10.2307/1907301.

[22] H. J. Miller and S.-L. Shaw, *Geographic Information Systems for Transportation: Principles and Applications*, ch. 6: Network Flows and Facility Location. New York: Oxford University Press, 2001.

[23] H. J. Miller and S.-L. Shaw, "Geographic Information Systems for Transportation in the 21st Century," *Geography Compass*, vol. 9, no. 4, pp. 180–189, 2015. https://doi.org/10.1111/gec3.12204.

[24] R. Lovelace and D. Ballas, "'Truncate, replicate, sample': A method for creating integer weights for spatial microsimulation," *Computers, Environment and Urban Systems*, vol. 41, pp. 1–11, Sept. 2013. https://doi.org/10.1016/j.compenvurbsys.2013.03.004.

[25] D. R. Hardoon, S. Szedmak, and J. Shawe-Taylor, "Canonical correlation analysis: An overview with application to learning methods," *Neural computation*, vol. 16, no. 12, pp. 2639–2664, 2004. https://doi.org/10.1162/0899766042321814.

[26] J. H. Zar, "Significance testing of the spearman rank correlation coefficient," *Journal of the American Statistical Association*, vol. 67, no. 339, pp. 578–580, 1972. https://doi.org/10.1080/01621459.1972.10481251.

[27] C. M. Macal, N. T. Collier, J. Ozik, E. R. Tatara, and J. T. Murphy, "Chisim: An agent-based simulation model of social interactions in a large urban area," in *2018 winter simulation conference (WSC)*, pp. 810–820, IEEE, 2018. https://doi.org/10.1109/wsc.2018.8632409.

[28] G. S. Thakur, B. L. Bhaduri, J. O. Piburn, K. M. Sims, R. N. Stewart, and M. L. Urban, "Planetsense: a real-time streaming and spatio-temporal analytics platform for gathering geo-spatial intelligence from open source data," in *Proceedings of the 23rd SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pp. 1–4, 2015.

[29] M. W. Horner, M. D. Duncan, B. S. Wood, Y. Valdez-Torres, and C. Stansbury, "Do aging populations have differential accessibility to activities? Analyzing the spatial structure of social, professional, and business opportunities," *Travel Behaviour and Society*, vol. 2, no. 3, pp. 182–191, 2015. https://doi.org/10.1016/j.tbs.2015.03.002.

[30] B. S. Wood and M. W. Horner, "Understanding Accessibility to Snap-Accepting Food Store Locations: Disentangling the Roles of Transportation and Socioeconomic Status ," *Applied Spatial Analysis and Policy*, vol. 9, pp. 309–327, 2016. https://doi.org/10.1007/s12061-015-9138-2.

[31] S. J. Rey, L. Anselin, P. Amaral, D. Arribas-Bel, R. X. Cortes, J. D. Gaboardi, W. Kang, E. Knaap, Z. Li, S. Lumnitz, T. M. Oshan, H. Shao, and L. J. Wolf, "The PySAL Ecosystem: Philosophy and Implementation," *Geographical Analysis*, vol. 54, no. 3, pp. 467–487, 2022. https://doi.org/10.1111/gean.12276.

[32] Q. Huangfu and J. A. J. Hall, "Parallelizing the dual revised simplex method," *Mathematical Programming Computation*, vol. 10, no. 1, pp. 119–142, 2018. https://doi.org/10.1007/s12532-017-0130-5.

itk-elastix: Medical image registration in Python

Konstantinos Ntatsis[‡*], Niels Dekker[‡], Viktor van der Valk[‡], Tom Birdsong[§], Dženan Zukić[§], Stefan Klein[¶], Marius Staring[‡], Matthew McCormick[§]

Abstract—Image registration plays a vital role in understanding changes that occur in 2D and 3D scientific imaging datasets. Registration involves finding a spatial transformation that aligns one image to another by optimizing relevant image similarity metrics. In this paper, we introduce `itk-elastix`, a user-friendly Python wrapping of the mature `elastix` registration toolbox. The open-source tool supports rigid, affine, and B-spline deformable registration, making it versatile for various imaging datasets. By utilizing the modular design of `itk-elastix`, users can efficiently configure and compare different registration methods, and embed these in image analysis workflows.

Index Terms—medical imaging, image analysis, registration, elastix, ITK, wrapping, Python

Introduction

Image Registration

Image registration is a fundamental process in the field of scientific imaging that enables the alignment and comparison of images, facilitating the understanding of changes that occur within datasets. It involves finding a spatial transformation that optimizes relevant image similarity metrics, ensuring accurate alignment between images. A frequent registration type is the parametric approach where the spatial transformation is explicitly modeled. Examples of such transformation models are the rigid transform which allows translations and rotations, the affine transform that additionally includes shearing and the B-Spline transform that permits only local deformations. The reader can refer to Modersitzki [1] for an overview of the nonparametric registration. In addition to the parametric model, the choice of similarity metric plays a crucial role in the registration result and is dependent on the relationship of the pixel intensities between the images. Simple metrics such as normalized correlation are suitable for images with a linear intensity relationship, while more complex metrics such as mutual information [2] are employed for non-linear relationships.

Medical imaging heavily relies on image registration techniques [3] [4] to gain valuable insights and quantitative measurements. By registering medical images acquired at different time points or using various imaging modalities such as MRI and CT, researchers can analyze and quantify changes in anatomical structures, track disease progression and assess treatment efficacy. For instance, image registration allows the alignment of medical volumes across subjects to evaluate the impact of specific treatments, or the registration of sequential brain images to monitor tumor growth and response to therapy.

elastix

`elastix` [5] [6] is a well-known and widely used open-source toolbox dedicated to image registration. It provides a comprehensive range of algorithms and utilities designed for aligning images using diverse transformation models, similarity measures, and optimization strategies. One of its key strengths is its modular design, enabling users to easily configure and combine different registration methods to suit application-specific needs. Parameter files govern the registration process by specifying transformation models, similarity measures, optimization strategies, and related parameters. By customizing these configurations, users can seamlessly adapt `elastix` to their specific requirements, ensuring optimal registration outcomes.

For example, when aligning an MRI brain scan with a CT scan using `elastix`, users can configure the transformation model, such as an affine or B-spline transformation model, to capture the geometric relationships between input images. They can also specify the similarity measure, like mutual information or normalized correlation, to evaluate the quality of alignment. Additionally, users have the flexibility to adjust optimization strategies, including parameters like the maximum number of iterations, to fine-tune the registration process and achieve optimal results. `elastix` supports both the more typical pairwise registration but also groupwise registration [7] [8], where no image is specified as fixed but an implicit mean image is used instead as reference.

The `elastix` codebase is implemented in C++ and serves as an extension to the Insight Toolkit (ITK) [9]. Through nearly two decades of development, `elastix` has achieved a mature state, characterized by stability, practical effectiveness, maintainability, and general backward compatibility. ITK Image data structures play a crucial role within `elastix`, representing multi-dimensional pixel data augmented with spatial information. Acting as a vital link between the digital pixel space and the physical space of the imaged object, ITK Images facilitate accurate registration. By computing transformations that map points from the physical space of one image to corresponding points in another, `elastix` achieves precise and meaningful alignment outcomes within the physical space. Complementing `elastix`, a utility software named `transformix` was developed to enable the application of registration results to additional images.

* *Corresponding author: k.ntatsis@lumc.nl*
‡ *Division of Image Processing, Department of Radiology, Leiden University Medical Center, Leiden, the Netherlands*
§ *Medical Computing Group, Kitware, Inc, Carrboro, NC, USA*
¶ *Biomedical Imaging Group Rotterdam, Department of Radiology & Nuclear Medicine, Erasmus MC, Rotterdam, the Netherlands*

The original and still-supported method to utilize `elastix` and `transformix` are command line executables. For the end user, this approach has the advantage that it does not require any external dependencies to be installed, which eases deployment. However, one limitation of this executable-based approach is its reliance on file input/output (I/O) operations. To address this limitation and enable more efficient in-memory operations, a C++ API was developed for `elastix` and `transformix`. This API follows the paradigm established by ITK and its processing filters. By adopting this design approach, `elastix` and `transformix` gained the ability to perform operations directly in memory. This enhancement provides users with greater flexibility and efficiency in their image registration workflows.

To further accommodate the needs of the users in the continuously developing scientific computing ecosystem, wrappings of the C++ code to other languages was developed in the form SimpleElastix [10], which still exists as part of the SimpleITK [11] package. More recently, we have embarked on developing a Python-specific wrapper called `itk-elastix`. This wrapper extends the functionality of `elastix` and offers an ever-expanding collection of Jupyter [12] examples, along with integration with other scientific processing libraries and visualization software. While there are other scientific python image image registration packages, `itk-elastix` stands out as a comprehensive Pythonic package with many image similarity metrics, implementations for 2D, 3D, and 4D images, and the ability to register a variety of imaging modalities. The subsequent sections of this paper delve into these aspects in greater detail.

`itk-elastix`: Python wrapping

The backend C++ `elastix` code is wrapped in Python with the Simplified Wrapper and Interface Generator (SWIG [13]). The Python wrapping of `elastix`, `itk-elastix`, brings the power of `elastix` to the Python ecosystem, providing effortless integration with other scientific processing libraries and visualization software. The `itk-elastix` Python packages builds on the `itk` Python package's pythonic interface and seamless integration with packages in the scientific Python ecosystem such as NumPy [14]. This enables users to leverage the rich functionality of `elastix` within their Python workflows, benefiting from its advanced image registration capabilities alongside popular Python libraries such as NumPy [14], SciPy [15], and MONAI [16] [17].

The process of updating and distributing the `itk-elastix` Python package is as follows: Once a significant number of changes have been made to the C++ `elastix` repository, a pull request is initiated in the `itk-elastix` repository to update its version. This triggers the `itk-elastix` Continuous Integration (CI) system, which performs builds of Python packages across various Python versions (ranging from 3.7 to 3.11 at the moment of writing) and major platforms such as Windows, Linux, and macOS. When a git version tag is provided, the wrapped `itk-elastix` is automatically uploaded to PyPI, accompanied by a comprehensive summary of updates between the versions. As a result, users can easily install the latest `itk-elastix` by executing `pip install itk-elastix` within their Python environment. It is important to note that rigorous testing is conducted on the `elastix` backend functionality, with hundreds of tests performed during each pull request or commit, utilizing the CI system of the C++ repository. The test framework of `elastix` consists of various categories of tests, including low-level unit tests of the elastix library interface, minimal image registration tests on very small synthetic images, and larger regression tests of image registrations on realistic medical data. The tests are implemented using the CMake test driver CTest, the Python unittest module, and GoogleTest.

The Python wrapping for any ITK filter including `elastix` and `transformix`, offers two APIs: one functional and one object-oriented. We will describe the two API options and demonstrate the `itk-elastix` functionality with examples in the two following sections.

Functionality

Registration/transformation example

The following example demonstrates the registration of 2D MRI brain images using the `itk.elastix_registration_method` and subsequent transformation of the corresponding moving mask using the `itk.transformix_filter`. The objective is to compare the overlap measure between the fixed mask and the transformed moving mask. It is important to note that this is a synthetic example where the fixed image intentionally exhibits significant deformations through an artificial non-linear transformation, solely for illustrative purposes. The masks utilized in this example represent segmentations of the head, including the brain and the skull. The procedure begins by reading the fixed and moving images from disk, followed by configuring a default set of B-spline registration parameters to be used for the registration process.

```
import itk
from scipy.spatial.distance import dice

# Load the moving and the fixed image from disk
fixed_image = itk.imread('./data/fixed.mha', itk.F)
moving_image = itk.imread('./data/moving.mha', itk.F)

# Configure a (default) parameter map with all the
# registration parameters
par_obj = itk.ParameterObject.New()
par_map = par_obj.GetDefaultParameterMap('bspline')
par_obj.AddParameterMap(par_map)

# Run the registration
# 1. The Object Oriented way
# elastix_obj = itk.ElastixRegistrationMethod.New(
#                   fixed_image,
#                   moving_image)
# elastix_obj.SetParameterObject(param_obj)
# elastix_obj.Update()
# result_image = elastix_obj.GetOutput()
# rtp = elastix_obj.GetTransformParameterObject()

# 2. The functional way
# rtp: result transform parameter object
result_image, rtp = itk.elastix_registration_method(
                            fixed_image,
                            moving_image,
                            parameter_object=par_obj)
```

Following the registration process, we load the masks from disk and apply the transformation parameters obtained during registration to the moving mask. To preserve the binary nature of the masks and avoid introducing interpolation artifacts, we utilize the nearest neighbor interpolator. This choice ensures that the binary properties of the masks are maintained throughout the transformation process.

```
# Load the corresponding masks
fixed_mask = itk.imread('./data/f_mask.mha', itk.UC)
```

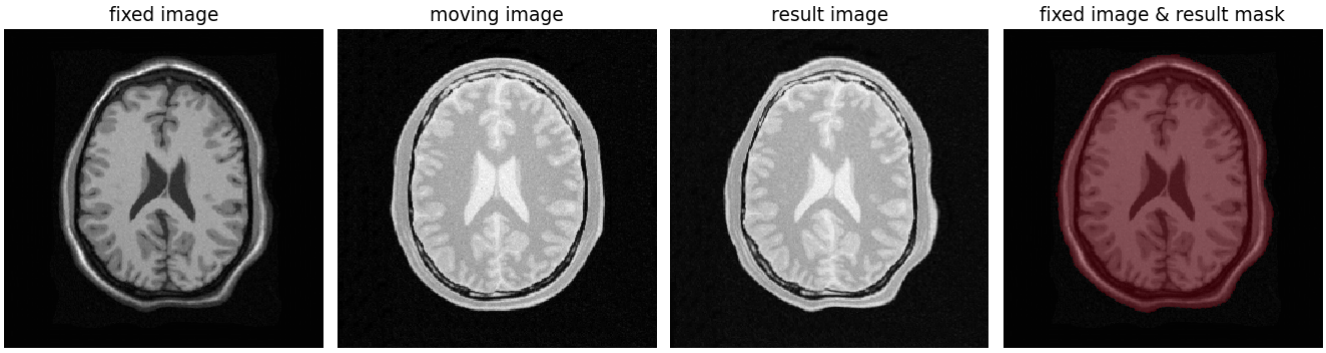

Fig. 1: *Synthetic example of 2D brain registration and transformation of masks.*

```
moving_mask = itk.imread('./data/m_mask.mha', itk.UC)

# Transform the moving mask using the result from the
# registration
rtp.SetParameter(0,
      'ResampleInterpolator',
      'FinalNearestNeighborInterpolator')
result_mask = itk.transformix_filter(moving_mask,
                                     rtp)

# Compute dice on masks
initial_dice = 1 - dice(fixed_mask[:].ravel(),
                        moving_mask[:].ravel())
result_dice = 1 - dice(fixed_mask[:].ravel(),
                       result_mask[:].ravel())

print(initial_dice, result_dice)
```

The last part of the code above calculates the Dice coefficient between the fixed mask and the transformed moving mask by converting the pixel arrays in the ITK Images into NumPy array views and then call `scipy.distance.dice()` on them. The initial Dice score was **97.88%** which increased to **99.37%** after registration. Figure 1 visualizes the fixed, moving and result image as well as an overlay of the fixed image and the transformed mask.

Jupyter Notebook collection

In addition to the core registration and transformation functionality demonstrated above, `itk-elastix` offers other additional features. To help new users who are starting out, and also keep existing users up-to-date with the new feature implementations, we offer an evolving collection of Jupyter Notebooks as usage examples. Each of the Notebooks covers usually a specific topic, can be run independently, and includes comments and detailed explanations. The Notebooks are also tested automatically by CI with each pull-request or commit, and hence it is ensured that they always reflect the current API and functionality of the codebase. Such Notebooks include, but are not limited to:

- specifying masks or point sets for the registration
- transforming point sets and meshes
- groupwise registration
- logging options
- saving output to disk options
- reading/writing transform in hd5f format
- calculation of spatial jacobian
- calculation of deformation field
- calculation of the inverse transform
- visualization of the registration

Interoperability with other packages

ITK Transforms

In addition to the fact that `elastix` is based on ITK, there is an ongoing effort to increase the compatibility between the two libraries even further. One particular example is the Transform classes [18]. In the following example, we show that ITK Transforms can be used directly by `transformix`:

```
# Create an ITK (translation) transform
transform = itk.TranslationTransform.New()
transform.SetOffset([50, -60])

# Specify the image space of the transform
sp = moving_image.shape
parameter_map = {
                 "Direction": ("1", "0", "0", "1"),
                 "Index": ("0", "0"),
                 "Origin": ("0", "0"),
                 "Size": (str(sp[1]), str(sp[0])),
                 "Spacing": ("1", "1")
                }

par_obj = itk.ParameterObject.New()
par_obj.AddParameterMap(parameter_map)

# Pass an ITK transform directly to transformix
transformix_obj = itk.TransformixFilter.New(
                            moving_image)
transformix_obj.SetTransformParameterObject(par_obj)
transformix_obj.SetTransform(transform)
transformix_obj.Update()

# Get transformed (translated) image
translated_image = transformix_obj.GetOutput()
```

NumPy & SciPy

Interoperability with NumPy and, consequently, with SciPy libraries, comes from functionality in ITK to convert ITK Images to NumPy arrays and vice versa. The relevant code is:

```
# itk image -> numpy array (deep copy)
image_array = itk.array_from_image(image_itk)

# itk image -> numpy array (shallow copy / view)
image_array = image_itk[:]

# numpy array -> itk image
image_itk = itk.image_from_array(image_array)
```

Project MONAI

More and more people work on the application of deep learning to medical imaging research. To that end, we developed

Fig. 2: *The user interface of the* `elastix-napari` *plugin. For a larger version of the image: https://github.com/SuperElastix/elastix-napari#elastix-napari.*

itk_torch_bridge as module of the MONAI codebase that allows conversion 1) of an ITK Image to a MONAI MetaTensor and the reverse, while making sure that all relevant metadata remain intact, and 2) an ITK Transform to a MONAI Transform and back. The latter is necessary since the ITK Transforms are defined in the world coordinate system while MONAI uses the pixel/voxel space. Example of a relevant application is performing deep learning registration (e.g. affine) using MONAI, and passing the Transform as initial Transform for `itk-elastix`, which can further register the images (e.g. non-linearly). Below, there is a short code snippet on how to use the module:

```
from monai.data import itk_torch_bridge as itb
import torch

# itk image <-> MONAI metatensor
image_mt = itb.itk_image_to_metatensor(image_itk)
image_itk = itb.metatensor_to_itk_image(image_mt)

# Transform: monai space <-> itk space
# affine_matrix: 3x3, matrix: 2x3, translation: 2x1
matrix, translation = itb.monai_to_itk_affine(
                        image=image,
                        affine_matrix=affine_matrix)
```

Integration with other software

Continuous efforts have been made to make `itk-elastix` accessible to users of various tools. One notable community-driven initiative is SlicerElastix, which seamlessly integrates `elastix` (as an executable) into 3D Slicer [19] medical image visualization software. In addition to this, recent endeavors focused on developing the `elastix-napari` plugin for the Napari [20] visualization software, which is written in Python. Figure 2 illustrates Napari user interface and showcases an `itk-elastix` widget on the right side along with an example visualization of two input images and a transformed image at the center.

Documentation & reproducibility

`elastix` has been extensively used and cited for over a decade, resulting in the accumulation of significant community knowledge. In the spirit of reproducible science, and recognizing the value of building upon previous work, we have compiled a curated list of parameter files in a parameter file model zoo, each linked to its associated publication. This resource allows interested users to easily filter the list based on factors such as anatomical region, modality, or image dimensionality, empowering them to find pre-existing parameter files that suit their needs. By facilitating result replication on their own datasets and providing guidance for novel registration tasks, this initiative promotes reproducibility and collaboration within the community.

The documentation for each parameter, component, and API functionality is continuously updated using Sphinx, ensuring that it stays up-to-date with the latest developments in `elastix`. This allows users to access accurate and relevant information, with in-code descriptions automatically rendered as comments into a website for easy access and query capabilities. In addition, for a more comprehensive understanding of registration and the inner workings of `elastix`, the elastix manual provides in-depth descriptions covering various aspects, including detailed explanations of the algorithms and methodologies employed. To further support users, a community forum hosted as GitHub discussions serves as a valuable resource for asking questions, seeking assistance, and engaging in discussions with experienced users and developers who can provide support, share insights, and address any concerns or challenges faced by users.

Concluding remarks

We presented `itk-elastix`, an easy-to-install and easy-to-use Python package that lowers the entry barrier for multi-dimensional image registration. Its key features are 1) a robust and well-established backend codebase that provides stability and reliability, 2) an extensive collection of tutorials, a parameter file model zoo, and up-to-date documentation as comprehensive resources for user adoption, 3) seamless interoperability with popular scientific libraries in Python, including NumPy, SciPy, and MONAI, and 4) integration into 3D visualization software, facilitating visual analysis and interpretation of registered images. Overall, with `itk-elastix`, researchers and practitioners can effortlessly leverage the strengths of Python and seamlessly integrate it with a wide range of scientific software, which unlocks new possibilities and accelerates advancements in scientific image analysis. Next steps will further improve the applicability of `itk-elastix` on end-to-end deep learning segmentation and registration pipelines of diverse medical datasets. In addition, a port to WebAssembly will enhance the universal accessibility of the package.

Acknowledgment

We gratefully acknowledge the financial support received from the Chan Zuckerberg Initiative (CZI) through the Essential Open Source Software for Science award for Open Source Image Registration: The elastix Toolbox, numbers 2020-218571 and 2021-237680 and the National Institute of Mental Health (NIMH) of the National Institutes of Health (NIH) under the BRAIN Initiative award number 1RF1MH126732.

Useful resources

- itk-elastix repository: https://github.com/InsightSoftwareConsortium/ITKElastix
- jupyter notebook examples: https://github.com/InsightSoftwareConsortium/ITKElastix/tree/main/examples
- elastix-napari plugin: https://github.com/SuperElastix/elastix-napari

- elastix community forum: https://github.com/SuperElastix/elastix/discussions
- parameter file model zoo: https://elastix.lumc.nl/modelzoo/
- elastix documentation and manual: https://elastix.lumc.nl/doxygen/index.html

REFERENCES

[1] J. Modersitzki, *Numerical methods for image registration*. OUP Oxford, 2003.

[2] J. P. Pluim, J. A. Maintz, and M. A. Viergever, "Mutual-information-based registration of medical images: a survey," *IEEE transactions on medical imaging*, vol. 22, no. 8, pp. 986–1004, 2003, https://doi.org/10.1109/JPROC.2003.817864.

[3] J. A. Maintz and M. A. Viergever, "A survey of medical image registration," *Medical image analysis*, vol. 2, no. 1, pp. 1–36, 1998, https://doi.org/10.1016/S1361-8415(01)80026-8.

[4] F. P. Oliveira and J. M. R. Tavares, "Medical image registration: a review," *Computer methods in biomechanics and biomedical engineering*, vol. 17, no. 2, pp. 73–93, 2014, https://doi.org/10.1080/10255842.2012.670855.

[5] S. Klein, M. Staring, K. Murphy, M. A. Viergever, and J. P. Pluim, "Elastix: a toolbox for intensity-based medical image registration," *IEEE transactions on medical imaging*, vol. 29, no. 1, pp. 196–205, 2009, https://doi.org/10.1109/TMI.2009.2035616.

[6] D. P. Shamonin, E. E. Bron, B. P. Lelieveldt, M. Smits, S. Klein, M. Staring, and A. D. N. Initiative, "Fast parallel image registration on CPU and GPU for diagnostic classification of Alzheimer's disease," *Frontiers in neuroinformatics*, vol. 7, p. 50, 2014, https://doi.org/10.3389/fninf.2013.00050.

[7] C. T. Metz, S. Klein, M. Schaap, T. van Walsum, and W. J. Niessen, "Nonrigid registration of dynamic medical imaging data using nD+ t B-splines and a groupwise optimization approach," *Medical image analysis*, vol. 15, no. 2, pp. 238–249, 2011, https://doi.org/10.1016/j.media.2010.10.003.

[8] W. Huizinga, D. H. Poot, J.-M. Guyader, R. Klaassen, B. F. Coolen, M. van Kranenburg, R. Van Geuns, A. Uitterdijk, M. Polfliet, J. Vandemeulebroucke *et al.*, "PCA-based groupwise image registration for quantitative MRI," *Medical image analysis*, vol. 29, pp. 65–78, 2016, https://doi.org/10.1016/j.media.2015.12.004.

[9] M. McCormick, X. Liu, J. Jomier, C. Marion, and L. Ibanez, "ITK: enabling reproducible research and open science," *Frontiers in neuroinformatics*, vol. 8, p. 13, 2014, https://doi.org/10.3389/fninf.2014.00013.

[10] K. Marstal, F. Berendsen, M. Staring, and S. Klein, "SimpleElastix: A user-friendly, multi-lingual library for medical image registration," in *2016 IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, 2016, pp. 574–582, https://doi.org/10.1109/CVPRW.2016.78.

[11] B. C. Lowekamp, D. T. Chen, L. Ibáñez, and D. Blezek, "The design of SimpleITK," *Frontiers in neuroinformatics*, vol. 7, p. 45, 2013, https://doi.org/10.3389/fninf.2013.00045.

[12] T. Kluyver, B. Ragan-Kelley, F. Pérez, B. Granger, M. Bussonnier, J. Frederic, K. Kelley, J. Hamrick, J. Grout, S. Corlay, P. Ivanov, D. Avila, S. Abdalla, C. Willing, and J. development team, "Jupyter notebooks - a publishing format for reproducible computational workflows," in *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, F. Loizides and B. Scmidt, Eds. IOS Press, pp. 87–90. [Online]. Available: https://eprints.soton.ac.uk/403913/

[13] The SWIG development team, "Simplified wrapper and interface generator." [Online]. Available: https://www.swig.org/

[14] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant, "Array programming with NumPy," vol. 585, no. 7825, pp. 357–362, https://doi.org/10.1038/s41586-020-2649-2. [Online]. Available: https://doi.org/10.1038/s41586-020-2649-2

[15] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. J. van der Walt, M. Brett, J. Wilson, K. J. Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. J. Carey, İ. Polat, Y. Feng, E. W. Moore, J. VanderPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, P. van Mulbregt, and SciPy 1.0 Contributors, "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python," *Nature Methods*, vol. 17, pp. 261–272, 2020, https://doi.org/10.1038/s41592-019-0686-2.

[16] M. J. Cardoso, W. Li, R. Brown, N. Ma, E. Kerfoot, Y. Wang, B. Murrey, A. Myronenko, C. Zhao, D. Yang *et al.*, "MONAI: An open-source framework for deep learning in healthcare," *arXiv preprint arXiv:2211.02701*, 2022, https://doi.org/10.48550/arXiv.2211.02701.

[17] A. Diaz-Pinto, S. Alle, A. Ihsani, M. Asad, V. Nath, F. Pérez-García, P. Mehta, W. Li, H. R. Roth, T. Vercauteren *et al.*, "MONAI Label: A framework for AI-assisted interactive labeling of 3D medical images," *arXiv preprint arXiv:2203.12362*, 2022, https://doi.org/10.48550/arXiv.2203.12362.

[18] B. B. Avants, N. J. Tustison, G. Song, B. Wu, M. Stauffer, M. M. McCormick, H. J. Johnson, and J. C. Gee, "A unified image registration framework for itk," in *International Workshop on Biomedical Image Registration*. Springer, 2012, pp. 266–275, https://doi.org/10.3389/fninf.2014.00044.

[19] A. Fedorov, R. Beichel, J. Kalpathy-Cramer, J. Finet, J.-C. Fillion-Robin, S. Pujol, C. Bauer, D. Jennings, F. Fennessy, M. Sonka *et al.*, "3D Slicer as an image computing platform for the Quantitative Imaging Network," *Magnetic resonance imaging*, vol. 30, no. 9, pp. 1323–1341, 2012, https://doi.org/10.1016/j.mri.2012.05.001.

[20] napari contributors, "napari: a multi-dimensional image viewer for python," 2019, https://doi.org/10.5281/zenodo.3555620. [Online]. Available: https://napari.org/stable/

PyQtGraph - High Performance Visualization for All Platforms

Ognyan Moore[‡*], Nathan Jessurun[§], Martin Chase[§], Nils Nemitz[§], Luke Campagnola[¶]

Abstract—PyQtGraph is a plotting library with high performance, cross-platform support and interactivity as its primary objectives. These goals are achieved by connecting the Qt GUI framework and the scientific Python ecosystem. The end result is a plotting library that supports using native python data types and NumPy arrays to drive interactive visualizations on all major operating systems. Whereas most scientific visualization tools for Python are oriented around publication-quality plotting and browser-based user interaction, PyQtGraph occupies a niche for applications in data analysis and hardware control that require real-time visualization and interactivity in a desktop environment.

The well-established framework supports line plots, scatter plots, and images, including time-series 3D data represented as 4D arrays, in addition to the basic drawing primitives provided by Qt.

For datasets up to several hundred thousand points, real-time rendering speed is achieved by optimized interaction with the Python bindings of the Qt framework. For enhanced image processing capabilities, PyQtGraph optionally integrates with CUDA. This ensures rendering capabilities are scalable with increasing data demands. Moreover, this improvement is enabled simply by installing the CuPy[1] library, i.e. requiring no in-depth user configurations.

PyQtGraph provides interactivity not only for panning and scaling, but also through mouse hover, click, drag events and other common native interactions. Since PyQtGraph uses the Qt framework, the user can substitute their own intended application behavior to those events if they feel the library defaults are not appropriate. This flexibility allows the development of customized and streamlined user interfaces for data manipulation.

The included parameter tree framework allows straightforward interactions with arbitrary user functions and configuration settings. Callbacks execute on changing parameter values, even asynchronously if requested.

An active developer community and regular release cycles indicate and encourage further library development. PyQtGraph's support cycle is synchronized with the NEP-29[2] standard, ensuring most popular scientific python modules are continually compatible with each release.

PyQtGraph is available through pypi.org (https://pypi.org/project/pyqtgraph/), conda-forge (https:/ anaconda.org/conda-forge/pyqtgraph) and GitHub (https://github.com/pyqtgraph/pyqtgraph).

Index Terms—Visualization, Qt, NumPy, PyData, Python

Introduction

The benefits of interactive exploration of scientific data were recognized as soon as computer systems gained graphical displays. While early implementations like the PRIM-9 system[3] of the Stanford Linear Accelerator Center were only available to large installations, more affordable microcomputers soon found their place in smaller laboratories also[4][5], controlling experiments and recording data.

Software packages designed to acquire and process this data soon appeared, with MATLAB[6] and LabView[7] both implementing graphical representation of data from their very first versions. The latter was designed to enable data acquisition, processing and visualization all in the framework of a single program. This approach remains common in fields like statistics where the tools for interaction with data are reasonably well-defined. In other areas, the advent of high-level programming languages like Java and Python has enabled researchers to create the tools for their specific needs with reasonable time investment. This is facilitated by a continuously growing open-source infrastructure that provides resources addressing anything from mathematical methods[8] to full-scale laboratory data infrastructure[9], [10].

With less need to recreate existing solutions, it becomes feasible to implement software aiming to reduce turn-around times of iterated experiments: A traditional view of the scientific method envisions a sequence of detailed experiment design, pain-staking note-taking, followed by an exhaustive evaluation resulting in a revised experiment. However, when experiments can be optimized over a wide parameter space, the evaluation quickly becomes the dominant factor. Even for established experimental parameters, external factors such as degraded performance of equipment result in a significant loss of time if they are discovered only in subsequent evaluation.

The solution is to provide immediate feedback to the researcher throughout the experiments, and data visualization has long proven its effectiveness in this regard [Friendly2008]. A challenge lies in providing tools for a detailed inspection of interesting data while new information continues to arrive at rates that for extreme cases are counted in Gb/s even after preselection[11]. These tools also need to provide the flexibility to handle data that falls outside the range expected in design, as this is the most likely to indicate failures or to provide the sought-after discovery.

Here we present a visualization library created with these goals in mind. Although written in Python to allow for easy expansion, a close integration with the cross-platform Qt UI framework[12] it provides the capability to interactively handle datasets of hundreds of thousands of points, or live representation of high-resolution camera data.

* *Corresponding author: ognyan.moore@gmail.com*
‡ *Hobu Inc.*
§ *Unaffiliated*
¶ *Allen Institute*

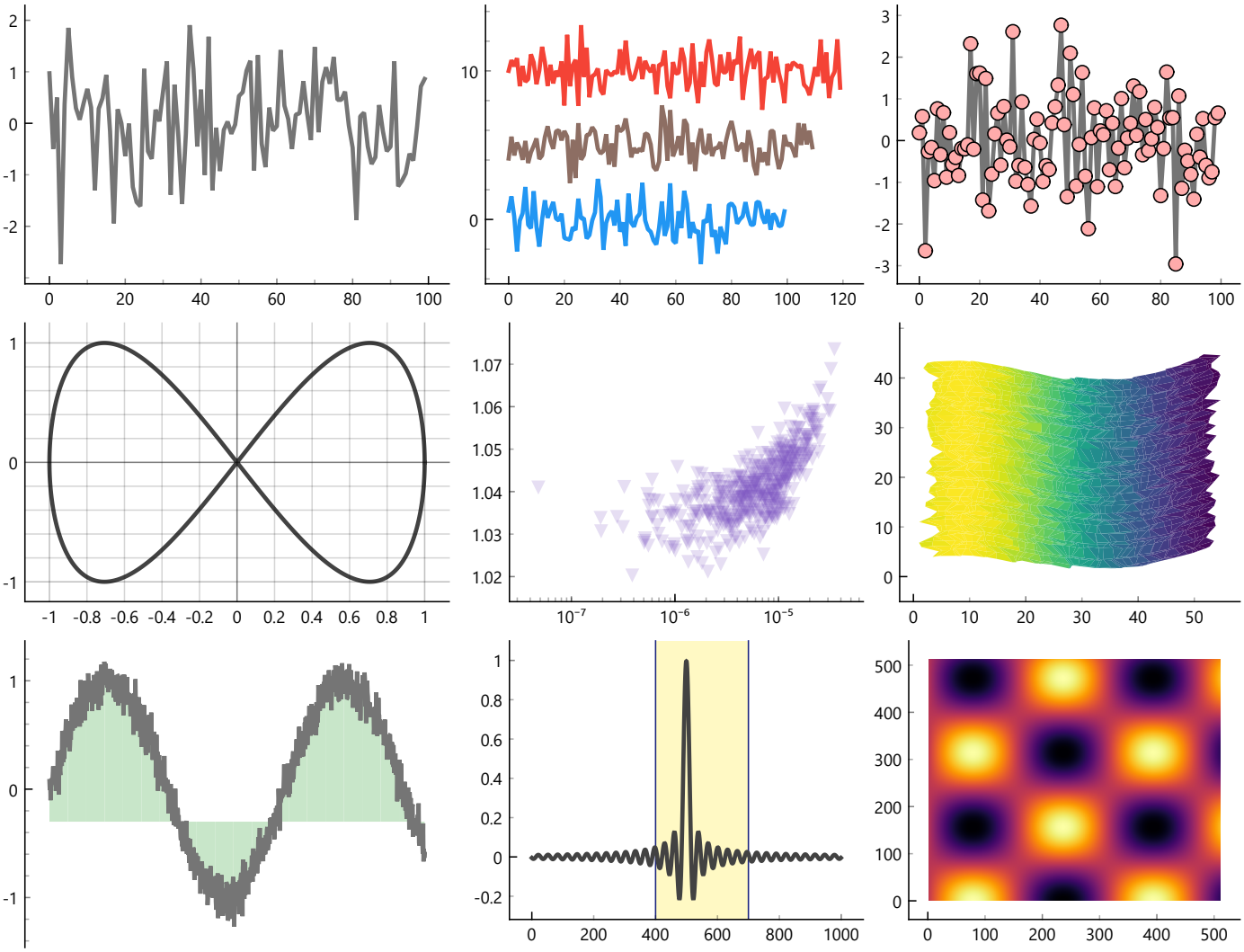

Fig. 1: *A selection of basic plots from PyQtGraph's suite of examples.*

Approach

Python

The Python programming language enjoys a large popularity in scientific research due ease of entry and a robust standard library combined with access to very comprehensive numerical computing packages. This makes Python an attractive alternative to established computational tools such as MATLAB[6] and Mathematica.

The set of most commonly used scientific computing tools in Python are commonly referred to as the SciPy stack. This refers to SciPy, NumPy, and a variety of other libraries that use the NumPy `ndarray` data structure as a container for vectorized operations. The `ndarray` gives developers a high level API to low-level operations with excellent performance. This API allows NumPy and SciPy to provide a wide variety of standard numerical computing operations, all of which are very efficient and help overcome the performance penalty of working with Python as a cross-platform, interpreted, dynamically typed language.

Qt

The Qt framework is a GUI platform written in C++ that allows the creation of cross-platform applications with a single shared code-base. Comprehensive Python bindings (PyQt) expose the complete Qt API. Here, the specific section of interest is the `GraphicsView` framework, which provides a surface for managing and interacting with a large number of custom-made 2D graphical items, with support for zooming and rotation[13]. PyQtGraph is built on this foundation to extend the SciPy stack with performant cross-platform visualization.

Implementation

`GraphicsView` renders line segments in a freely scaled coordinate system through `QPainterPath` objects. The rendering performance of PyQtGraph results from optimized code to create such paths directly from NumPy ndarrays describing sets of x and y coordinates. One illustrative example tightly interfaces with Qt's internal pointers through `QPolygonF` objects to offer significant speedups for `QPainterPath` generation. They use NumPy's structured array functionality to efficiently create a binary compatible structure that can serve as an input stream to a `QPainterPath` item (see the Appendix section for details). This `QPainterPath` is then drawn to the screen by the Graphics View framework. Note that while `arrayToQPolgyonF` is a trivial example of NumPy/Qt integration, a much more complex usage can be found here.

CAPABILITIES

All 2d line rendering functions that handle large quantities start with NumPy arrays and become painter paths through the powerful `arrayToQPath` conversion. This generic NumPy-to-Qt data translator covers all common plotting requirements. Figure 1 shows a demonstration from the suite of examples. All graphs included in this paper were generated using PyQtGraph's interactive export functions, which can store both bitmaps and vector formats, or provide access to the raw plotted data.

Plot Types

PyQtGraph shows all plots within a `PlotItem` object consisting of a `ViewBox` equipped with a set of axes. This allows dynamic pan and zoom through the transforms of Qt's `GraphicsView`, with no need to regenerate the `QPainterPath` objects. Individual elements of the plot are represented by graphics items that share the same coordinate systems and shown in any combination and drawing order.

PyQtGraph represents line plots as `PlotCurveItem` objects and offers typical functionality such as color, width and dashing. "Shadow pen" lines can be underlaid for additional contrast.

Scatter plot items are assigned a default shape, color and size per data set, but each point can also have a unique attributes. Shapes are pre-rendered and cached to optimize performance when the underlying dataset is updated. Depending on the application, symbols can be set to scale with the view or maintain constant size. Functionality is included for items in scatter plots to recognize mouse hover events.

Plots can be extended by both horizontal and vertical error bars and annotated by text labels. Built in routines can also transform the plotted data to provide logarithmic scaling, Fourier transforms, and to show the gradient dy/dt directly over t or as a phase map over y.

Bar graphs and images also make use of this framework and can be added to the same PlotItem, although they are more commonly used separately. Users can also create `QPainterPath` objects to add their own graphical elements using the well documented methods of the Qt Graphics View framework. PyQtGraph's suite of examples[14] illustrates this with some demonstrations.

Performance

We evaluated the plotting performance for line plots of randomly generated datasets of different length. Figure 2 shows the time taken from setting new data to the completion of the drawing process for 1 to 100 separate curves ranging from 100 to 10 million points. We find that even on common hardware, a curve with 10,000 points can be drawn in less than 10 ms, and an update rate of 60 Hz can be maintained up to approximately 30,000 points. Adding more curves introduces additional overhead, such that the same number of 10,000 points, plotted over 100 curves of 100 points each, increases the update time to just below 40 ms. Nevertheless, the number of points in each of the 100 curves can be increased to close to 3,000 before the update rate falls below 10 frames per second (FPS). At this point, the majority of time is spent processing line segments, and their distribution across different numbers of curves is of secondary importance: A single curve allows for 200,000 points to be displayed at 10 FPS.

Plotting the results as the update time divided by the overall number of data points further illustrates this. As the total number of points approaches 100,000, where the more or less fixed overhead of the Qt drawing process is no longer significant, the update times converges to approximately 200 ns per point drawn for both 10 curves and 100 curves. We attribute the increased update time for a single curve to the larger set of data that needs to be handled simultaneously, which may lead to caching issues.

Although the detailed result will vary with platform, system, and data, we consider these results to provide a good reference for the performance that can be expected from PyQtGraph.

Images and Regions of Interest

PyQtGraph also provides the means to display images and other multi-dimensional data. Handling streams of such data, as in live video, is similarly enabled by efficient NumPy methods that convert the input data into a binary representation that can be used directly by the Qt framework. Various analysis and processing tools interact with the image arrays, for example regions of interest (ROIs), look-up tables (LUT) for color-mapped display, or histograms.

Image Views

The principal object in displaying images, `ImageItem`, accepts 2-dimensional (interpreted as grayscale) or 3-dimensional (either color or color and alpha) data of any numeric type. Stored in NumPy arrays, this data can be pre-processed efficiently using any available functions in the SciPy stack. Subtracting a background, for instance, is simply a matter of subtracting the reference frame. This input is then processed by `ImageItem` and converted into Qt's `QImage` format for rapid display. A range of colormaps are provided to enhance detail perception, and can be altered interactively in levels and colors through a `HistogramLUTItem`.

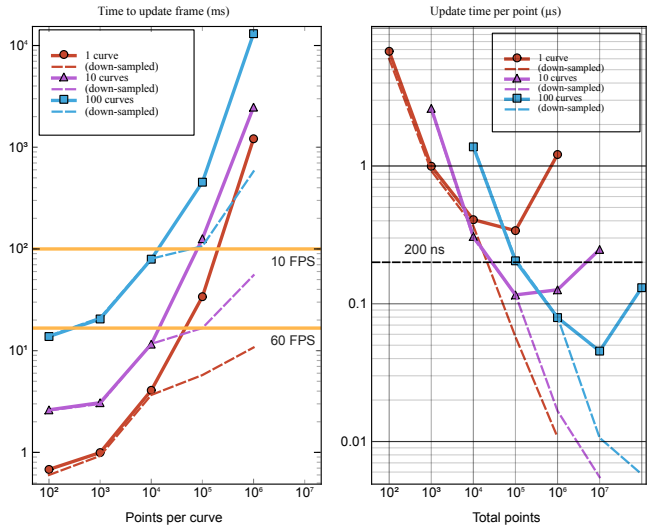

Fig. 2: *Line speed benchmark. The time to render 1, 10 or 100 lines of data is shown for varying numbers of points per line. All data was created using an AMD 5900x Ryzen 9 CPU. Left: Time per update over points per curve. The thresholds for achieving 10 and 60 frames/s are shown by horizontal lines. Right: Update time per point, plotted over the total number of points. For more than 100,000 points, the line-plotting time becomes dominant, and the results converge to 200 ns per point for both 10 and 100 curves, while plotting all points as a single curve increases the time to 500–600 ns per point.*

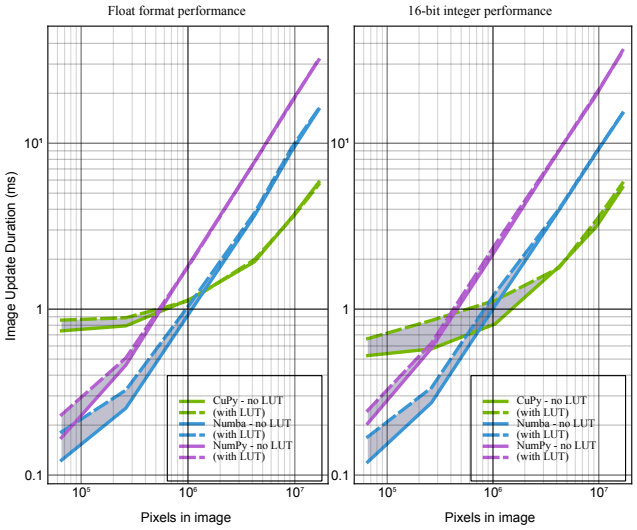

Fig. 3: *Image speed benchmark. The time to update an image frame is shown for different data formats. Left: Using optimized NumPy processing (purple lines), the drawing time is log-scale linear with the number of pixels over a wide range. GPU accelerated CUDA processing using CuPy (green lines) describe a more complex relationship with image size. The need to copy data to and from the GPU creates additional overhead, but as image size grows, the faster processing speed becomes sufficient to compensate for that overhead. The choice of various extra processing tasks like LUTs (dashed lines) show the same basic trends. Alternatively, PyQtGraph's image rendering pipeline can be accelerated in Numba is available on the system. Benchmarks with Numba (blue lines) can be seen as have performance between that of CuPy and NumPy only. Right: For input data in uint16 format, CUDA processing is particularly advantageous and can provide an almost four-fold reduction in drawing time. Benchmarks were performed on an AMD 5900x Ryzen 9 CPU and an NVIDIA RTX 3080 discrete GPU.*

ROIs

A common image analysis task is to define a ROI in a larger original image. This is supported by multiple interactive objects (`LineROI`, `CircleROI`, `PolygonROI`, and others), which provide NumPy slice objects that reference the selected region within the image array. Once extracted, the relevant data can then be further processed. Magnification, live plotting, FFTs and custom analysis are all simple to implement. Multiple ROIs can be bound together in groups to provide background correction or region comparisons both within a single image stream or across many. These ROI objects remain interactive while attached to the image, so that resizing, moving and rotating a ROI can prompt immediate updates of all subsequent plotting and analysis interfaces.

Performance

Numerous factors play into the final performance of a video stream. Data type conversions, LUTs, scaling, and any custom preprocessing all need to occur for each frame, and the computational effort typically scales with image size. A minimum of 20 FPS is generally required for a usable interactive video stream, although 60 FPS is preferred in many applications. In some cases, data can be directly passed to the built-in methods of Qt's `QImage`. Otherwise `ImageItem` relies on the core function `makeARGB` to efficiently convert data types, order data properly, rescale levels and apply a LUT if desired (see the Appendix section for details).

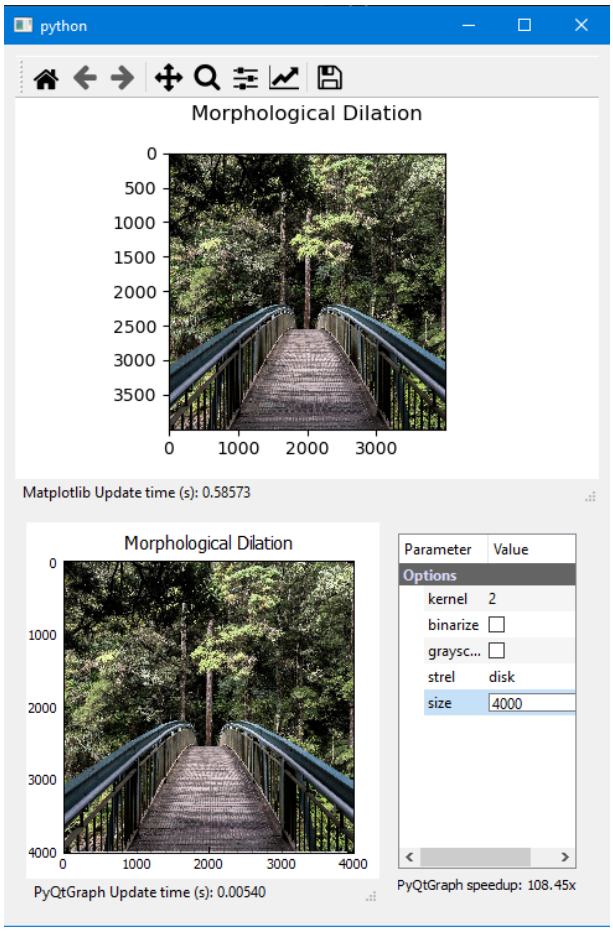

Fig. 4: *Performance test with PyQtGraph and Matplotlib widgets embedded in a Qt5 application. Over a wide range of image sizes, PyQtGraph completes drawing approximately 75–150 times faster, taking only 5.4 ms in this example of a 4000×4000 image. The test is performed without GPU acceleration in a Microsoft Windows environment, and both libraries are set to sub-sample without interpolation. Free-to-use test images are provided by the "Unsplash" service.*

When integrated as a widget in a Qt application (Figure 4), we typically find `ImageItem` to display an image 75–150 times faster than the `FigureCanvas` provided by Matplotlib, a plotting library that emphasizes graphical quality over speed.

Some share of the image processing is by necessity done in the primary event thread of the Qt application, as that thread requires full access to the data to be displayed. Other calculations can be moved to other threads to improve performance and maintain the responsiveness of the UI. For example, larger images can be downsampled before handing them to the main thread for display. This multi-threading consideration extends throughout the application, and any excessive use of the event thread will impact image display performance.

To further accelerate the handling of large datasets, PyQtGraph can make use of a GPU substrate in one of two ways: `GLImageItem` or CuPy. `GLImageItem`, while limited in its interactivity, employs OpenGL for rendering. The CuPy library, a drop-in replacement for NumPy, moves array processing tasks to a CUDA-enabled GPU. This is not beneficial in all applications, since the cost of copying the image data between system memory and the GPU needs to be amortized by a sufficient number of calculations. In the context of image processing, we find that CuPy

provides an advantage for images with several hundred thousands of pixels (Figure 3), depending on target hardware.

Interactivity

Event Driven GUI

The Qt framework is event driven, which allows PyQtGraph to provide seamless mouse interaction. This also enables users to develop their own desired behavior in response to mouse move, hover, leave, enter, double-click, zoom, or drag events. Almost every aspect of PyQtGraph interacts with the Qt events, or provides its own in response to e.g. axis adjustments or changes to a selected region. This interactivity is a core component of the Qt framework, and adding such behavior to a plot in PyQtGraph is no more complicated than generating the plot in the first place.

Responsiveness at scale

Recognizing zoom events enables resolution-aware downsampling of the plotted data. Multiple available methods provide different trade-offs of accuracy against performance, and include a "peak" display that precisely captures the minima and maxima of the data, a "mean" over the down-sampled interval, and a fast "sub-sample" that displays only 1 in N data points. Zooming into the view automatically reveals more detail of the dataset.

Parameter Trees

Another common requirement for user interaction is a mechanism to interact with for configuration settings or algorithm parameters. PyQtGraph provides this capability through the `ParameterTree` object which hosts any number of `Parameters`. Similar to `traitlets`, PyQtGraph's `Parameter` objects encapsulate a value type and allow registering callbacks, performing input validation, and more. However, `Parameters` are different in that most are coupled to a widget representation, i.e., allowing users to easily update the values graphically. `Parameter` objects can be created through a simple Python dictionary listing its specifications (type, value, and traits such as 'readonly' or value range). It is then bound to a Qt widget for editing text, numeric, list-like, and custom data depending on the parameter type. Parameters can be grouped, linked, and dynamically instantiated or removed. Callbacks for user actions or value changes allow results to be recalculated immediately (i.e., while a spinbox is changing its value) or after the value has settled. Parameter trees can save and load their states hierarchically to easily create persistent configuration files. Since accessing parameters mimics a Python dictionary, they can function as a drop-in replacement for programmatically adjusted settings rather than forcing users to interface through widgets alone.

EXAMPLES

Rapid iteration of processing parameters

Figure 5 shows such a parameter tree in use. In applications such as image processing, immediate feedback for a choice of algorithmic parameters can help to rapidly reduce the exploration space in the search for viable solutions. For instance, it might be difficult to tell the appropriate kernel size for a morphological operation without testing multiple combinations of image types, parameter values, and more. These factors often make fine-tuning a laborious process. Parameter trees assist in creating a tool to integrate the user with the testing space, quickly and without large amounts of boilerplate code. Using callbacks to provide immediate response, workable parameter combinations can be explored, and candidate solutions can be stored to configuration files, both for comparison to alternative approaches and for application to specific data types.

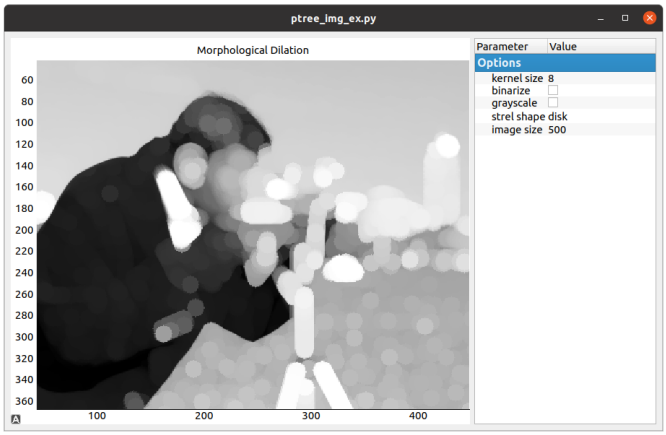

Fig. 5: *Sample use of parameter trees for user interaction, where various image processing parameters can be quickly updated. The displayed image reflects these changes in real-time.*

Model Prototyping

The supplementary information contains a similar application of PyQtGraph's capabilities to a machine learning model. Here the parameter trees allows tuning aspects of the input data, model structure and output formats. The plotting functions provide live feedback for how these changes affect model accuracy, greatly assisting a rapid prototyping process.

Monitoring of real-time data

Visualization can provide immediate feedback on measurement results and the operational state of the equipment involved. Figure 6 shows an application of the opportunities provided by PyQtGraph's interactive facilities in this application. For most applications, no data reduction is necessary to maintain smooth display of a sufficiently large buffer, and no additional code is needed to alternate between monitoring of new data and close inspection of specific events.

Additional examples

The supplementary information includes video demonstrations of two additional applications that make heavy use of PyQtGraph functionality to explore spectral data and to visualize volumetric data representing the 3d structure of multilayer circuit boards.

SOFTWARE DEVELOPMENT

The original motivation for pyqtgraph was in data acquisition software, where there is a need to be able to display video and plots with realtime frame rates and interactivity that allows data exploration. Interactivity was highly important; the established `matplotlib` library already existed and was excellent for visualizing data in a way that tells a particular story. New data, though, doesn't have this story yet. You want to be able to slice it

```
def ndarray_from_qpolygonf(polyline):
    # polyline.data() will be None if the pointer was null.
    # voidptr(None) is the same as voidptr(0).
    vp = Qt.compat.voidptr(polyline.data(), len(polyline)*2*8, True)
    return np.frombuffer(vp, dtype=np.float64).reshape((-1, 2))

def create_qpolygonf(size):
    polyline = QtGui.QPolygonF()
    if hasattr(polyline, 'resize'):
        # (PySide) and (PyQt6 >= 6.3.1)
        polyline.resize(size)
    else:
        polyline.fill(QtCore.QPointF(), size)
    return polyline

def arrayToQPolygonF(x, y):
    """
    Utility function to convert two 1D-NumPy arrays representing curve data
    (X-axis, Y-axis data) into a single open polygon (QtGui.PolygonF) object.
    """
    # Validation asserts both x and y are same-shaped and 1D, not shown here
    size = x.size
    polyline = create_qpolygonf(size)
    memory = ndarray_from_qpolygonf(polyline)
    memory[:, 0] = x
    memory[:, 1] = y
    return polyline
```

TABLE 1: *PyQtGraph source code for the core `arrayToQPolygonF` function.*

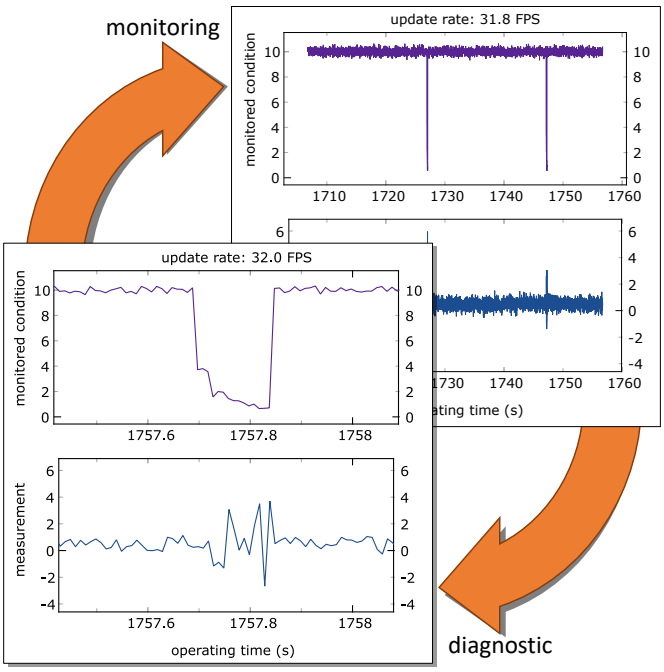

Fig. 6: *Monitoring and diagnostic of a (simulated) experiment with intermittent failures. Incoming data at 100 samples/s for two measurement channels is recorded into a rolling 5,000 point buffer and continuously displayed at 30 frames/s. When a failure is observed, it can quickly be brought into focus with simple mouse interactions (click-and-drag and mousewheel zoom) for inspection, or to record accurate time stamps. Afterwards, a single click returns the view to automatic scaling without loss of any incoming data.*

and stretch it and look at it from every possible angle, quickly and easily, so that you can decide what story to tell.

At the time, most acquisition software would have been written in C/C++ for efficiency. However, newer developments meant python interfaces to Qt's C++ logic provided a good mix between speed and ease of use. PyQwt was perfect for this purpose, but went through a long period without a maintainer (presumably at the time, it was a huge burden maintaining and distributing compiled python packages). So pyqtgraph began as a replacement for PyQwt that would be pure-python, and thus easier to develop and distribute. Following that template, it was also to include UI elements that have common use in acquisition/analysis applications, but are missing from Qt (for example, tools for adjusting image contrast, parameter trees, etc.).

PyQtGraph was first released in 2012, under the open source MIT license. It is known to run on systems ranging from the Raspberry Pi to IBM's s390x architecture. Development is coordinated by volunteer maintainers, with additional code provided by occasional contributors. A continuous integration system asserts that the codebase passes a suite of tests for different combinations of Qt bindings, Python versions and operating systems. PyQtGraph has adopted NEP-29[2] to establish a support timeline for Python and NumPy versions in line with the rest of the Python community and development occurs in close communication with projects such as ACQ4[15] and Orange3[16] that constitute a large part of the user base.

OUTLOOK

With a growing number of both maintainers and contributors, PyQtGraph is well positioned to take advantage of technological developments. The support of hardware acceleration in recent versions of NumPy has already been used to add CUDA integration to some time-critical code, but there is still plenty of potential for further improvements to performance and capabilities. Increased use of multi-threaded patterns is a goal in this respect, both throughout the library, and in user code supported by appropriate

```
import cupy as cp
import numpy as np

def makeARGB(data, lut=None, levels=None, scale=None, useRGBA=False, output=None):
    # condensed variant, full code at:
    # https://github.com/pyqtgraph/pyqtgraph/blob/pyqtgraph-0.12.0/pyqtgraph/functions.py#L1102-L1331
    xp = cp.get_array_module(data) if cp else np

    nanMask = None
    if data.dtype.kind == "f" and xp.isnan(data.min()):
        nanMask = xp.isnan(data)
    # Scaling
    if isinstance(levels, xp.ndarray) and levels.ndim == 2: # rescale each channel independently
        newData = xp.empty(data.shape, dtype=int)
        for i in range(data.shape[-1]):
            minVal, maxVal = levels[i]
            if minVal == maxVal:
                maxVal = xp.nextafter(maxVal, 2 * maxVal)
            rng = maxVal - minVal
            rng = 1 if rng == 0 else rng
            newData[..., i] = (data[..., i] - minVal) * (scale / rng)
        data = newData
    else:
        minVal, maxVal = levels
        rng = maxVal - minVal
        data = (data - minVal) * (scale / rng)
    # LUT
    if xp == cp: # cupy.take only supports "wrap" mode
        data = cp.take(lut, cp.clip(data, 0, lut.shape[0] - 1), axis=0)
    else:
        data = np.take(lut, data, axis=0, mode='clip')

    imgData = output
    if useRGBA:
        order = [0, 1, 2, 3]  # array comes out RGBA
    else:
        order = [2, 1, 0, 3]  # channels line up as BGR in the final image.
    # attempt to use library function to copy data into image array
    fastpath_success = try_fastpath_argb(xp, data, imgData, useRGBA)
    if fastpath_success:
        pass
    elif data.ndim == 2:
        for i in range(3):
            imgData[..., i] = data
    elif data.shape[2] == 1:
        for i in range(3):
            imgData[..., i] = data[..., 0]
    else:
        for i in range(0, data.shape[2]):
            imgData[..., i] = data[..., order[i]]
    if data.ndim != 3 or data.shape[2] != 4:
        imgData[..., 3] = 255
    # apply nan-mask through alpha channel
    if nanMask is not None:
        if xp == cp: # Workaround for https://github.com/cupy/cupy/issues/4693
            imgData[nanMask, :, 3] = 0
        else:
            imgData[nanMask, 3] = 0
    return imgData
```

TABLE 2: *PyQtGraph source code for the core* `makeARGB` *function. For brevity, edge cases and null checks have been omitted.*

documentation, examples and API design. The growing maturity of the Numba just-in-time compiler [17] for Python code provides additional opportunities for acceleration beyond what NumPy's array operations can provide.

APPENDIX

Implementation of arrayToQPolygonF

The function `arrayToQPolygonF` is one of the simpler cases that demonstrates the how PyQtGraph bridges the gap between NumPy and Qt Table 1

Execution takes two `ndarray` objects of the same length, representing *x*- and *y*-coordinates for a series of line segments. A `QPolygonF` object is instantiated and resized to store enough points that represent the *x*- and *y*-coordinates that were passed in. From there, a void-pointer of the `QPolygonF`'s internal memory is retrieved in a NumPy format allowing easy assignment of the user data. Lastly, we fill that NumPy array with the *x*- and *y*-coordinates that were initially provided. In this process, we went from NumPy arrays representing *x*- and *y*- coordinates to a `QPolygonF` object without performing any serialization, iteration or casting.

Implementation of makeARGB

The function `makeARGB` provides the data conversions used in displaying image data. It is included here as Table 2 to show the approach and the integration of CUDA GPU support discussed in section .

The segment of memory within a `QImage` object that will ultimately be displayed on the screen can be accessed and written to as a contiguous, row-major, 3-dimensional NumPy `ndarray` of unsigned 8-bit integers; i.e. the red, green and blue color values and alpha value of each pixel, one row at a time. With this array as the output target, an incoming image data goes through a number of processing steps. Many of the steps are only conditionally executed, depending on the shape and type of the incoming data, as well as the use of LUTs or rescaling. Some of the respective branches and decision trees have been omitted here for brevity. In a best-case scenario, the incoming data is already in the correct format, and the steps converting data type and element order can then also be omitted. The CuPy library provides CUDA support by replicating large sections of NumPy functionality, allowing for near-identical code paths. The two if-statements seen here address the lack of a 'clip' mode in CuPy's 'take' function, as well as differing behavior for masks as indices.

Acknowledgements

The authors wish to thank all prior, present and future contributors to the PyQtGraph project. Their efforts enable all that is presented here. One regular contributor, @pijyoi, has made significant contributions to the NumPy and Qt interoperability, as well as reviewed pull requests from other contributors and maintainers and provided countless bug-fixes. Finally, we would like to thank maintainers and contributors to the NumPy, SciPy and CuPy projects.

References

[1] R. Okuta, Y. Unno, D. Nishino, S. Hido, and C. Loomis, "CuPy: A NumPy-compatible library for NVIDIA GPU calculations," in *Proceedings of Workshop on Machine Learning Systems (LearningSys) in The Thirty-first Annual Conference on Neural Information Processing Systems (NIPS)*, 2017. [Online]. Available: http://learningsys.org/nips17/assets/papers/paper_16.pdf

[2] *NEP 29 — Recommend Python and NumPy version support as a community policy standard*, available at https://numpy.org/neps/nep-0029-deprecation_policy.html.

[3] J. H. Friedman and W. Stuetzle, "John W. Tukey's work on interactive graphics," *The Annals of Statistics*, vol. 30, pp. 1626–1639, 2002, https://doi.org/10.1214/aos/1043351250.

[4] J. S. Byrd, "Microcomputers for nuclear instrumentation," *presented at Conference and Exhibits on Small Computers, May 23-24 1979, Clemson, USA*, 1 1979, available at https://www.osti.gov/biblio/6060192.

[5] A. V. Reed, "On choosing an inexpensive microcomputer for the experimental psychology laboratory," *Behavior Research Methods & Instrumentation*, vol. 12, pp. 607–613, 1980, https://doi.org/10.3758/BF03201852.

[6] C. Moler and J. Little, "A history of MATLAB," in *Proceedings of the ACM on Programming Languages*, vol. HOPL 4. ACM New York, NY, USA, 2020, pp. 81.1–81.67, https://doi.org/10.1145/3386331.

[7] S. Josifovska, "The father of LabView," *IEE Review*, vol. 49, pp. 30–33, 2003, https://doi.org/10.1049/ir:20030905.

[8] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant, "Array programming with NumPy," *Nature*, vol. 585, no. 7825, pp. 357–362, 2020, https://doi.org/10.1038/s41586-020-2649-2.

[9] J. L. Johnson, H. tom Wörden, and K. van Wijk, "PLACE: An open-source Python package for laboratory automation, control, and experimentation," *Journal of Laboratory Automation*, vol. 20, pp. 10–16, 2015, https://doi.org/10.1177/2211068214553022.

[10] L. J. Koerner, T. A. Caswell, D. B. Allan, and S. I. Campbell, "A Python instrument control and data acquisition suite for reproducible research," *IEEE Transactions on Instrumentation and Measurement*, vol. 69, pp. 1698–1707, 2020, https://doi.org/10.1109/TIM.2019.2914711.

[11] C. Bozzi, S. Roiser, and the LHCb Collaboration, "The LHCb software and computing upgrade for run 3: opportunities and challenges," in *IOP Conf. Series: Journal of Physics: Conf. Series*, 2017, https://doi.org/10.1088/1742-6596/898/11/112002.

[12] *Qt widget toolkit*, https://www.qt.io.

[13] *QGraphicsView Class*, Qt documentation, March 2021, https://doc.qt.io/qt-5/qgraphicsview.html. [Online]. Available: https://doc.qt.io/qt-5/qgraphicsview.html

[14] *Example Application*, PyQtGraph, can be run after installation by `python -m pyqtgraph.examples`.

[15] L. Campagnola, M. Kratz, and P. Manis, "Acq4: an open-source software platform for data acquisition and analysis in neurophysiology research," *Frontiers in Neuroinformatics*, vol. 8, p. 3, 2014, https://doi.org/10.3389/fninf.2014.00003. [Online]. Available: https://www.frontiersin.org/article/10.3389/fninf.2014.00003

[16] J. Demšar, T. Curk, A. Erjavec, Črt Gorup, T. Hočevar, M. Milutinovič, M. Možina, M. Polajnar, M. Toplak, A. Starič, M. Štajdohar, L. Umek, L. Žagar, J. Žbontar, M. Žitnik, and B. Zupan, "Orange: Data mining toolbox in Python," *Journal of Machine Learning Research*, vol. 14, pp. 2349–2353, 2013. [Online]. Available: http://jmlr.org/papers/v14/demsar13a.html

[17] S. K. 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*, 2015, pp. 1–6.

aPhyloGeo-Covid: A Web Interface for Reproducible Phylogeographic Analysis of SARS-CoV-2 Variation using Neo4j and Snakemake

Wanlin Li[‡*], Nadia Tahiri[‡]

Abstract—The gene sequencing data, along with the associated lineage tracing and research data generated throughout the Coronavirus disease 2019 (COVID-19) pandemic, constitute invaluable resources that profoundly empower phylogeography research. To optimize the utilization of these resources, we have developed an interactive analysis platform called aPhyloGeo-Covid, leveraging the capabilities of Neo4j, Snakemake, and Python. This platform enables researchers to explore and visualize diverse data sources specifically relevant to SARS-CoV-2 for phylogeographic analysis. The integrated Neo4j database acts as a comprehensive repository, consolidating COVID-19 pandemic-related sequences information, climate data, and demographic data obtained from public databases, facilitating efficient filtering and organization of input data for phylogeographical studies. Presently, the database encompasses over 113,774 nodes and 194,381 relationships. Additionally, aPhyloGeo-Covid provides a scalable and reproducible phylogeographic workflow for investigating the intricate relationship between geographic features and the patterns of variation in diverse SARS-CoV-2 variants. The code repository of platform is publicly accessible on GitHub (https://github.com/tahiri-lab/iPhyloGeo/tree/iPhylooGeo-neo4j), providing researchers with a valuable tool to analyze and explore the intricate dynamics of SARS-CoV-2 within a phylogeographic context.

Index Terms—Phylogeography, Neo4j, Snakemake, Dash, SARS-CoV-2

Introduction

Phylogeography is a field of study that investigates the geographic distribution of genetic lineages within a particular species, including viruses. It combines principles from evolutionary biology and biogeography to understand how genetic variation is distributed across various spatial scales [1]. In the context of viruses, phylogeography aims to uncover the evolutionary history and spread of viral lineages by analyzing their genetic sequences and geographical locations. By examining the genetic diversity of viruses collected from various geographic locations, researchers can reconstruct the patterns of viral dispersal and track the movement and transmission dynamics of viral populations over time [2] [3] [4]. In phylogeographic studies of viruses, the integration of genetic sequences, geographic information, and temporal data is essential. Integrating genetic sequences with geographical data enables researchers to conduct robust analysis of phylogenetic relationships among viral strains and uncover intricate patterns of viral migration and transmission across diverse regions.Through the integration of genetic and temporal information, researchers can derive insights into the timescale of viral evolution and elucidate the origins as well as dispersal patterns of distinct viral lineages [5].

Throughout the COVID-19 pandemic, researchers worldwide sequenced the genomes of thousands of SARS-CoV-2 viruses. These endeavors have significantly enhanced researchers' ability to analyze the intricate temporal and geographic dynamics of virus evolution and dissemination, consequently playing a pivotal role in informing the development of effective public health strategies for the proactive control of future outbreaks. However, the abundance of genetic sequences and the accompanying geographic and temporal data are scattered across multiple databases, making it challenging to extract, validate, and integrate the information. For instance, in order to conduct a phylogeographic study in SARS-CoV-2, a researcher require access to data regarding the geographic distribution of specific lineages. This includes information on the predominant countries in which these lineages are prevalent, along with the earliest and latest recorded detection dates. The Pango Lineages Report serves as a valuable resource for obtaining such data [6]. Following this, researchers can utilize databases such as NCBI Virus resource [7] or GISAID [8] to access sequencing data corresponding to the identified country and lineage. Daily climate data (e.g., humidity, wind speed, and temperature) for each location involved during the pandemic can be obtained from reputable sources such as NASA/POWER DailyGridded Weather [9]. To supplement the analysis, epidemiological information, including COVID-19 testing and vaccination rates, can be sourced from projects such as Our World in Data [10]. In summary, conducting phylogeographic research in viruses entails not only the meticulous screening and selection of sequencing data but also the proficient management of associated geographic information and the integration of substantial volumes of environmental data. This multifaceted process can be time-consuming and susceptible to errors. The challenges associated with data collection, extraction, and integration have hindered the advancement of phylogeographic research within the field [11] [12].

To tackle these challenges, we employed the highly scalable and adaptable Neo4j graph database management system [13]

* *Corresponding author: Nadia.Tahiri@USherbrooke.ca*
‡ *Department of Computer Science, University of Sherbrooke, Sherbrooke, Canada*

for the storage, management, and querying of extensive SARS-CoV-2 variants-related data. Differing from traditional relational databases that employ tables and rows, Neo4j represents data as an interconnected network of nodes and relationships [14]. Graph theory, with its inherent advantages in relation analysis, has found extensive applications in Phylogeny. For instance, Laddada et al. (2022) [15] employed Neo4j to track and analyze mutation occurrences by treating each nucleotide site of SARS-CoV-2 sequences as a node, thereby exploring the connections between mutations. In our research, we utilize graph theory to trace the relationships among location, environmental factors, and lineages. By leveraging graph theory, this framework offers a robust foundation for modeling, storing, and analyzing intricate relationships between entities [16] [17].

On the other hand, while recent phylogeographic studies have extensively analyzed the genetic data of species across different geographic regions, many have primarily focused on species distribution or provided visual representations, without investigating the correlation between specific genes (or gene segments) and environmental factors [18] [19] [20] [21]. To bridge this gap, a novel algorithm applying sliding windows to scan the genetic sequence information related to their climatic conditions was developed by our team [22]. This algorithm utilizes sliding windows to scan genetic sequence information in relation to climatic conditions. Multiple sequences are aligned and segmented into numerous alignment windows based on predefined window size and step size. To assess the relationship between variation patterns within species and geographic features, the Robinson and Foulds metric [23] was employed to quantify the dissimilarity between the phylogenetic tree of each window and the topological tree of geographic features. Nonetheless, this process was computationally intensive as each window needed to be processed independently. Additionally, determining the optimal sliding window size and step size often required multiple parameter settings to optimize the analysis. Thus, reproducibility played a critical role in this process.

To address these challenges, we devised a phylogeographic pipeline that harnesses the capabilities of Snakemake, a modern computational workflow management system [24]. Distinguishing itself from other workflow management systems such as Galaxy [25] and Nextflow [26], Snakemake stands out as a Python-based solution, guaranteeing exceptional portability and the convenience of executing Snakefiles with a Python installation [27]. Leveraging various Python packages, including Biopython [28] and Pandas [29] [30], the Snakemake workflow efficiently handles tasks such as sequencing data reading and writing, as well as conducting phylogenetic analysis. Given these capabilities, Snakemake serves as an optimal choice for aPhyloGeo-Covid. Furthermore, Snakemake supports parallel execution of jobs, significantly enhancing the performance and speed of the pipeline. This pipeline implementation facilitates efficient and reproducible analysis, thereby streamlining the phylogeographic research workflow of the aPhyloGeo-Covid.

With a clear focus on addressing the aforementioned limitations, this study aims to develop an integrated, open-source phylogeographic analysis platform. This platform consists of two vital components: data pre-processing and phylogeographical analysis. In the data pre-processing phase, we employ searchable graph databases, enabling rapid exploration and offering a visual overview of SARS-CoV-2 lineages and their associated environmental factors. This efficient approach allows researchers to navigate through vast datasets and extract pertinent information for their analyses. In the subsequent phylogeographical analysis phase, our modularized Snakemake workflow is utilized to examine how genetic variation patterns within different SARS-CoV-2 variants align with geographic features. Leveraging this workflow, researchers can systematically and reproducibly investigate the relationship between viral genetic diversity and specific geographic factors. By adopting this comprehensive approach, a deeper understanding of the intricate interplay among viral evolution, transmission dynamics, and environmental influences can be achieved.

Methodology

A diverse range of data sources pertaining to SARS-CoV-2, covering the period from January 1, 2020, to December 31, 2022, were meticulously extracted, transformed, and loaded into a Neo4j graph database. These sources encompassed:

(1) SARS-CoV-2 sequences from the SARS-CoV-2 Data Hub [31]
(2) Lineage development information from Cov-Lineages [6]
(3) Population density by country, positivity rates, vaccination rates, diabetes rates, aging data from Our World in Data [10]
(4) Climate data from NASA/POWER [9]

To enable efficient querying, configuration of analysis parameters, and output generation within the database, a driver object was established using the Neo4j Python driver to establish seamless connections with the Neo4j database. For phylogeographic analysis, a streamlined workflow was implemented using the Snakemake workflow management system, ensuring an efficient and structured analysis process. Moreover, the interactive visualization capabilities offered by the Dash-Plotly library [32] [33] were leveraged for data exploration, analysis parameter setting, and interactive visualization of results, enhancing the interpretability and user-friendliness of the platform.

Data Integration

Within the Neo4j database, five labels were employed to effectively organize the data, encompassing Lineage, Protein, Nucleotide, Location, and Location Day (See Figure 1). The Protein and Nucleotide labels serve as repositories for sequencing data information, including accession number, sequence length, collection date, and collected country. The Lineage label stores lineage development information, encompassing the most common country, latest date, and earliest date associated with each lineage. Climate information such as temperature, precipitation, wind speed, humidity, and sky shortwave irradiance for each location and specific day is stored under the LocationDay label. The Location label contains fundamental information regarding hospitals, health, and the economy of each country, encompassing GDP, median age, life expectancy, population, proportion of people aged 65 and older, proportion of smokers, proportion of extreme poverty, diabetes prevalence, human development index, and other pertinent factors (See Table 1).

Lineage nodes establish connections with Nucleotide and Protein nodes, representing the relationships between lineages and their corresponding genetic sequence data. Moreover, Lineage nodes establish relationships with Location nodes, utilizing the most common occurrence rate as a property. This design empowers researchers to determine the most common countries based

TABLE 1: *Neo4j schema labels and properties for data integration.*

Label	Properties List
Protein	accession number, sequence length, collection date, collected country
Nucleotide	accession number, sequence length, collection date, collected country
Lineage	most common country, latest date, earliest date
Location Day	temperature, precipitation, wind speed, humidity, sky shortwave irradiance
Location	GDP, median age, life expectancy, population, proportion of people aged 65 and older, proportion of smokers, proportion of extreme poverty, diabetes prevalence, human development index

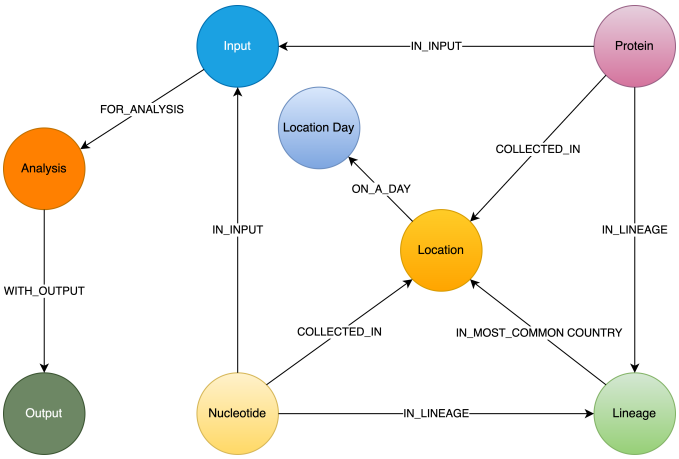

Fig. 1: *Schema of Neo4j Database for Phylogeographic Analysis of SARS-CoV-2 Variation. The schema includes key entities and relationships essential for organizing and querying data related to samples of protein, samples of nucleotide, locations, lineages, analysis input, output and parameters. Each entity represents a distinct aspect of the analysis process and facilitates efficient data organization and retrieval.*

on lineage names or search for lineages that were predominant in specific countries during specific time periods. This well-structured and interconnected design within the Neo4j database enhances the ability to explore, analyze, and extract meaningful insights from the integrated phylogeographic dataset.

Input exploration

An interactive platform using Dash-Plotly [32] [33] was developed for efficient data exploration and selection. The integration of the Dash platform with the Neo4j graph database allows for the seamless retrieval of pertinent data from interconnected nodes based on user-provided keywords related to lineages or locations. This functionality enables efficient identification and filtering of datasets for subsequent phylogeographic analysis. The integration of the powerful Neo4j database with the user-friendly interactive platform facilitates seamless data exploration and selection, supporting researchers in their comprehensive analysis of SARS-CoV-2 variation.

The aPhyloGeo-Covid offers two distinct approaches for selecting input datasets: 1) lineage-based approach for retrieving corresponding sequences based on selected lineage name and 2) location-based approach for retrieving corresponding sequences based on selected location and time period.

1. Lineage-based approach for retrieving corresponding sequences based on selected lineage name

The multi-step process is facilitated by the Neo4j Python package [34] and the interactive Dash web page. Initially, specific lineages of interest are selected from a checklist provided on the Dash web page. Subsequently, the selected lineages are utilized to query the graph database, extracting information about the predominant countries where these lineages are prevalent. The earliest and latest recorded dates, along with their corresponding predominant rates, are also retrieved. The obtained results are presented as an interactive Dash Table, providing an interface for applying column and row filters. This functionality allows for the exclusion of irrelevant locations or lineages based on specific research criteria. Additionally, predominant rates can be applied as a filter to exclude certain samples. Finally, based on the filtered table and the selected sequence type, all related sequences are extracted by accession number. These filtered sequences are then collected as input data for subsequent phylogeographic analysis.

Updating the sample table based on provided lineage names and sequence types, as mentioned earlier, is a crucial step in exploring input data for phylogeographic analysis. The following callback function accepts a sequence type (amino acid or nucleotide) and a list of selected lineage names as input and generates a Dash table containing relevant sample information as the output.

```
@app.callback(
    Output('lineage-table', 'data'),
    Input('button-confirm', 'n_clicks'),
    State('checklist-lineage', 'value'),
    State('dropdown-seqType', 'value')
)
def update_lineage_table(n_clicks,
                         checklist_value,
                         seqType_value):
    ...
    starts_with_conditions = " OR ".join(
            [f'n.lineage STARTS WITH "{char}"'
                 for char in checklist_value])
    query = f"""
      MATCH (n:Lineage) - [r] -> (l: Location)
      WHERE {starts_with_conditions}
      RETURN n.lineage as lineage,
             n.earliest_date as earliest_date,
             n.latest_date as latest_date,
             l.iso_code as iso_code,
             n.most_common_country as country,
             r.rate as rate
                """
    cols = ['lineage', 'earliest_date',
             'latest_date', 'iso_code',
             'country', 'rate']

    if checklist_value and seqType_value:
      #Query in Neo4j database
      # Transform Cypher results to dataframe
       df=neo_manager.queryToDataframe(query,cols)
       table_data = df.to_dict('records')
       return table_data
    ....
```

2. Location-based approach for retrieving corresponding sequences based on selected location and time period

Specific locations and a date period are defined by employing the Dash web page. Subsequently, the Neo4j database is queried to identify lineages prevalent in the specified locations during the

defined time period. The retrieved information includes the earliest and latest detected dates of the lineages in each country, along with their predominant rates. To present these findings, an interactive Dash Table is employed, facilitating the application of filters to exclude study areas or lineages below a predetermined threshold. Subsequently, the accession numbers of the corresponding sequences are extracted from the graph database. These filtered sequences are then collected for subsequent phylogeographic analysis.

The following function updates the sample table by incorporating selected start and end dates, sequence type and a list of selected locations. A Cypher query is employed to retrieve lineage data from the Neo4j database and apply filtering based on specified location and date criteria. This function empowers researchers to explore lineage data associated with diverse geographic regions within a specified date range.

```
@app.callback(
    Output('location-table', 'data'),
    Input('button-confirm', 'n_clicks'),
    State('date-range-lineage', 'start_date'),
    State('date-range-lineage', 'end_date'),
    State('checklist-location', 'value'),
    State('dropdown-seqType', 'value')
)
def update_table(n_clicks,
                 start_date,
                 end_date,
                 checklist_value,
                 seqType_value):
    ...
    query = f"""
      MATCH (n:Lineage) - [r] -> (l: Location)
      WHERE
        n.earliest_date > datetime("{start_date}")
       AND
        n.earliest_date < datetime("{end_date}")
       AND
        l.location in {checklist_value}
      RETURN n.lineage as lineage,
             n.earliest_date as earliest_date,
             n.latest_date as latest_date,
             l.iso_code,
             l.location as country,
             r.rate
                """
            cols = ['lineage', 'earliest_date',
                    'latest_date', 'iso_code',
                    'country', 'rate']
    if start_date_string and end_date_string
            and checklist_value and seqType_value:
        # Transform Cypher results dataframe
        df=neo_manager.queryToDataframe(query,cols)
        table_data = df.to_dict('records')
        return table_data
     ...
```

In summary, these approaches enable user-guided sequencing searches. Once the input sequences are defined, an Input node is generated in our graph database and appropriately labeled. This Input node establishes connections with the relevant sequencing (Nucleotide or Protein) nodes used in the analysis, highlighting relationships between the input data and the corresponding sequences. Each Input node is assigned a unique ID, which is provided for reference and traceability. These user-driven approaches provide a robust framework for sequencing searches, allowing researchers to define and explore input data relationships.

The generation of unique ID for nodes plays a crucial role in ensuring traceability for each analysis. To address this requirement, the provided function ensures that every new node is assigned a traceable ID.

```
def generate_unique_name(nodesLabel):
    driver = GraphDatabase.driver(URI,
                                  auth=("neo4j",
                                  password))
    with driver.session() as session:
        random_name = generate_short_id()

        result = session.run(
            "MATCH (u:" + nodesLabel +
            " {name: $name})
            RETURN COUNT(u)",
            name=random_name)
        count = result.single()[0]

        while count > 0:
            random_name = generate_short_id()
            result = session.run(
                "MATCH (u:" + nodesLabel +
                " {name: $name}) RETURN COUNT(u)",
                name=random_name)
            count = result.single()[0]

        return random_name
```

The following function facilitates the integration of input nodes with relationships to relevant sequence nodes within the Neo4j database, thereby enhancing the organization and management of input data and analysis entities in the network.

```
def add_Input_Neo(nodesLabel,
                  inputNode_name,
                  id_list):
    # Execute the Cypher query
    driver = GraphDatabase.driver(URI,
                                  auth=("neo4j",
                                  password))

    # Create a new node
    with driver.session() as session:
        session.run(
           "CREATE (userInput:Input {name: $name})",
           name=inputNode_name)
    # Perform MATCH query to retrieve nodes
    with driver.session() as session:
        result = session.run(
           "MATCH (n:" + nodesLabel + ")" +
           "WHERE n.accession IN $id_lt RETURN n",
           nodesLabel=nodesLabel,
           id_lt=id_list)
        # Create relationship for each matched node
        with driver.session() as session:
            for record in result:
                other_node = record["n"]
                session.run(
                   "MATCH (u:Input {name: $name}),
                          (n:" + nodesLabel +
                        " {accession: $id}) "
                    "CREATE (n)-[r:IN_INPUT]->(u)",
                    name=inputNode_name,
                    nodesLabel=nodesLabel,
                    id=other_node["accession"])
```

Parameters setting and tuning

After defining the input data, which includes sequence data and associated location information, researchers can utilize the platform to select the analysis parameters. This pivotal step entails creating an Analysis label, where the parameter values are stored as properties. These parameters encompass the step size, window size, RF distance threshold, bootstrap threshold, and the list of environmental factors involved in the analysis. Furthermore, a connection is established between the Input Node and the Analysis Node, offering several advantages. Firstly, it allows researchers to compare results obtained from the same input samples but with

different parameter settings. Secondly, it facilitates the comparison of analysis results obtained using the same parameter settings but different input samples. The interconnected Input, Analysis, and Output nodes (See Figure 1) ensure the repeatability and comparability of analysis results.

After confirming the parameters, the corresponding sequences are downloaded from NCBI [7] using the Biopython package [28], followed by performing multiple sequence alignments (MSA) [35] using the MAFFT method [36]. Subsequently, the Snakemake workflow is triggered in the backend, taking the alignment results and associated environmental data as input. Once the analysis is completed, a unique output ID is generated, enabling the results to be queried on the web platform.

The following function performs the preparation and storage of parameters and input data, subsequently triggering the workflow.

```
def trigger_workflow(df_params_geo):
    df = pd.DataFrame(df_params_geo)
    analysisNode = generate_unique_name("Analysis")
    outputNode = generate_unique_name("Output")

    # record parameters in config file
    with open('config/config.yaml', 'r') as file:
        config = yaml.safe_load(file)
    # Update the values
    config['accession_lt'] = df['id'].tolist()
    config['feature_names'] = df.columns.tolist()
    config['analysis_name'] = analysisNode
    config['output_name'] = outputNode
    # create geographic input dataset
    csv_file_name = config['geo_file']
    dff.to_csv(csv_file_name,
                index=False,
                encoding='utf-8')
    # create sequence input dataset
    aln_file_name = config['seq_file']
    seq_beforeMSA_fname = aln_file_name + '_raw'
    if config['data_type'] == 'aa':
        db_type = "protein"
    else:
        db_type = "nucleotide"
    accession_list = config['accession_lt']

    # update config dictionary to the YAML file
    with open('config/config.yaml', 'w') as file:
        yaml.dump(config, file)
    # (6) download sequences from NCBI

    seq_manager.downFromNCBI(
                    db_type,
                    accession_list,
                    seq_beforeMSA_fname)
    # (6) alignment
    seq_manager.align_MAFFT(seq_beforeMSA_fname,
                            aln_file_name)
    # (7) run aphylogeo snakemake workflow
    os.system("snakemake --cores all")
    # (8) In Neo4j create :Analysis node
    neo_manager.addAnalysisNeo()

    # (9) When Analysis finished,
     #save output dataframe into Output node
    neo_manager.addOutputNeo()
      ...
```

Output exploration

After each analysis, a unique output node is generated in the Neo4j graph database, connected to interrelated nodes that store input and parameter information, forming an intricate network of relationships. Through the ID of output node, analysis results can be conveniently traced and accessed. The platform not only facilitates querying individual results but also empowers the comparison of multiple analysis outcomes. Furthermore, as the platform is utilized, this network of input, analysis, and output nodes expands, enabling the acquisition of valuable insights from the data and facilitating comprehensive analysis of the phylogeographic patterns of SARS-CoV-2 variation.

Snakemake workflow for phylogeographic analysis

To investigate the potential correlation between the diversity of specific genes or gene fragments and their geographic distribution, a sliding window strategy was employed in addition to traditional phylogenetic analyses. As depicted in Figure 2, firstly, the multiple sequence alignment (MSA) was partitioned into windows by specifying the sliding window size and sliding window progress step size. Then a phylogenetic tree for each window was constructed. Secondly, cluster analyses for each geographic factor were performed by calculating a distance matrix and creating a reference tree based on the distance matrix and the Neighbor-Joining clustering method [37] [38]. Reference trees (based on geographic factors) and phylogenetic trees (based on sliding windows) were defined on the same set of leaves (i.e., names of species). Subsequently, the correlation between phylogenetic and reference trees was evaluated using the Robinson and Foulds (RF) distance calculation [23]. RF distances were calculated for each combination of the phylogenetic tree and the reference tree. Finally, bootstrap and RF thresholds were applied to identify gene fragments in which patterns of variation within species coincided with a particular geographic feature. These fragments can serve as informative reference points for future studies.

By scanning the complete Multiple Sequence Alignment sequences with a sliding window strategy, the phylogeographic research can effectively focus on sequence information for specific window lengths. To address the integration of genetic and environmental data, complex computational workflows are required, consisting of multiple interdependent processing steps. The aPhyloGeo snakemake workflow addresses this challenge by connecting each step through Snakemake rules, resulting in a comprehensive and easily automatable workflow. This workflow ensures reproducibility and facilitates result comparability across different sampling strategies, window sizes, and step sizes. Additionally, the aPhyloGeo workflow enables efficient processing of large datasets on parallel and distributed systems, leading to reasonable runtime.

Various tools and software were utilized to accomplish these analysis tasks, including Biopython [28], raxml-ng [39], fasttree [40], and Python libraries such as robinson-foulds, NumPy [41], and Pandas [29] [42]. A manuscript for aPhyloGeo-pipeline is available on Github Wiki (https://github.com/tahiri-lab/aPhyloGeo-pipeline/wiki).

Results and discussion

The SARS-CoV-2 virus has a genome size of approximately 30kb (See Figure 3). The first two-thirds of its genome, located at the 5'-terminal, encodes the instructions for the synthesis of two major proteins, namely pp1a, and pp1ab. Following viral enzyme processing, these proteins are transformed into 16 smaller non-structural proteins (Nsps). Specifically, ORF1a encodes nsp1–nsp10, while ORF1b encodes nsp1–nsp16, which play pivotal roles in viral replication and transcription [43]. Consequently, our first assessment of the aPhyloGeo-Covid performance focused on the pp1a region.

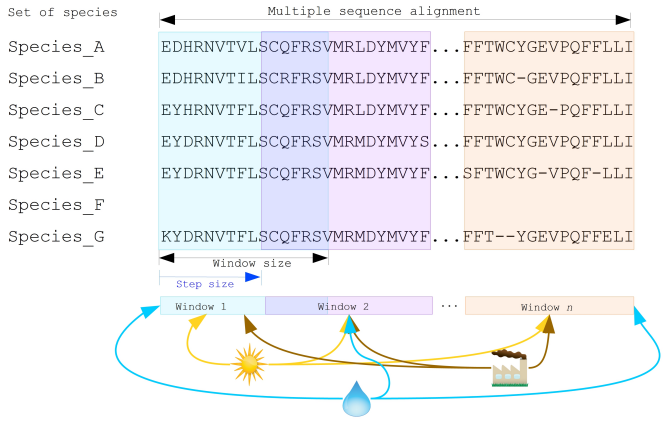

Fig. 2: *Integrated analysis of genetic data and environmental data. The aPhyloGeo workflow can analyze both amino acid sequence alignment data and nucleic acid sequence alignment data. By setting the window size and step size, the alignment of multiple sequences was segmented into sliding windows. For each sliding window, Robinson and Foulds distances are computed for every combination of the sliding window of phylogenetic tree and the reference tree created from environmental factors.*

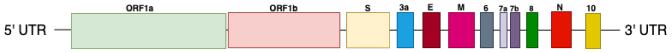

Fig. 3: *Schematic presentation of the SARS-CoV-2 genome Structure. SARS-CoV-2 follows the typical Betacoronavirus genome organization. The full-length RNA genome of approximately 29,903 nucleotides contains a replicase complex (composed of ORF1a and ORF1b) and four genes responsible for the production of structural proteins: Spike gene (S), Envelope gene (E), Membrane gene (M), and Nucleocapsid gene (N).*

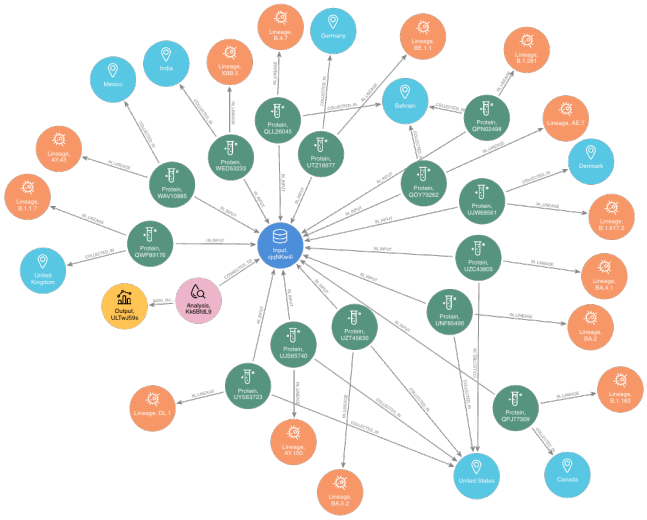

Fig. 4: *The networks of a single analysis experiment. For a specific analysis, the network highlights all entities serving as input data sources and their relationships. The Input node establishes connections between the data source objects and the specific analysis object. The Analysis node captures the parameters associated with the analysis, while the Output node stores the resulting analysis data.*

To identify and filter the appropriate datasets for further phylogeographic analysis around pp1a, 14 lineages starting with the codes AE, AY, B, BA, BE, DL, or XBB were selected from the checklist on the aPhyloGeo-Covid web page. Subsequently, with the Neo4j graph database, eight relevant locations were retrieved, where at least one of selected lineage was most prevalent (See Figure 4). An input node was created based on the lineages with connections of all the nodes of input sequences. The aPhyloGeo-Covid web page facilitated the definition of specific parameters for analysis, including a step size of 3 residues, a window size of 100 residues, an RF distance threshold of 100%, a bootstrap threshold of 0%, and a list of climate factors such as humidity, wind speed, sky shortwave irradiance, and precipitation (See Figure 5). These parameters were associated with the node of analysis and stored as properties within the node. Finally, the Snakemake workflow was triggered in the backend. At the completion of analysis, an output node with a unique identifier was generated within the Neo4j graph database (See Figure 4).

In this analysis experiment, we used aPhyloGeo-Covid to query preloaded climatic data from our Neo4j database for each sample connected to the input node. The climatic data was based on the most prevalent country and the time of initial collection. The meteorological parameters considered in our analysis included Precipitation Corrected, Relative Humidity at 2 Meters, Specific Humidity at 2 Meters, All Sky Surface Shortwave Downward Irradiance, Wind Speed within a 10-Meter Range, and Wind Speed within a 50-Meter Range. For statistical analysis, a user-defined average calculation interval of 3 days was applied. As shown in Figure 5 the 14 samples exhibited a range of precipitation from 0 mm/day to 8.57 mm/day with an average of 2.13 mm/day. The specific humidity ranged from 2.44 g/kg to 19.33 g/kg, averaging at 9.77 g/kg. The relative humidity ranged from 45.76% to 94.22%, with an average of 73.17%. Compared to other parameters, wind speed variability and sky surface shortwave downward irradiance showed relatively small variations across the samples. The sky surface shortwave downward irradiance ranged from 0.67 kW-hr/m2/day to 7.38 kW-hr/m2/day, with an average of 4.25 kW-hr/m2/day. The wind speed at 10 meters ranged from 1.90 m/s to 6.32 m/s, averaging at 3.24 m/s, while the wind speed at 50 meters ranged from 3.22 m/s to 6.40 m/s with an average of 4.39 m/s

At the end of the aPhyloGeo-Covid analysis workflow, a table was generated containing the RF distance between the phylogenetic tree of that window and the reference tree of a particular environmental feature. The distribution of normalized RF distances resulting from the phylogeographic analysis of the input dataset is presented in Figure 6. Windows exhibiting relatively lower RF distances merit further investigation. As illustrated in Figure 6, the RF distance range from 87.82% to 100%. Among the six climatic factors involved in the analysis, the sliding window region with the lower RF distance was exclusively identified in the integrated analysis involving precipitation. For this exploration, a scanning approach was employed, utilizing a window size of 100 residues and a step size of 3 residues for sequence analysis. Within the regions identified with low RF distance, special attention should be given to regions 792-940. Notably, a consistently low RF distance value of 81.82% was observed across all 17 windows spanning positions from 792 to 840. Furthermore, in accordance with SWISS-MODEL [44], the previous research validates the presence of a specific region of Nsp3 called Ubl1 (110 residues, position 819-929) within the identified sequence region. Ni et al. (2023) [45] revealed that the Ubl1 protein of SARS-CoV-2

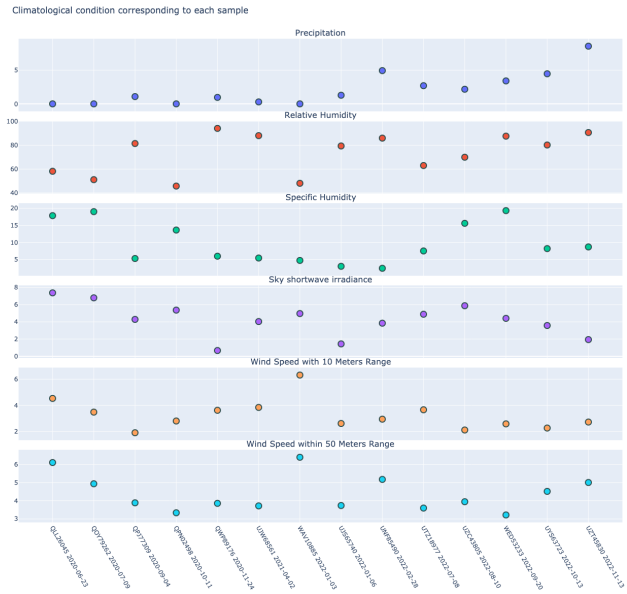

Fig. 5: *Climatic conditions of each sample in most common country at the time of first collection. The climate factors involved include Precipitation Corrected (mm/day), Relative Humidity at 2 Meters (%), Specific Humidity at 2 Meters (g/kg), All Sky Surface Shortwave Downward Irradiance (kW-hr/m^2/day), Wind Speed within 10 Meters Range (m/s), Wind Speed within 50 Meters Range (m/s).*

exhibits competitive binding with RNA molecules to the N protein, resulting in the dissociation of viral ribonucleoprotein complexes. Based on these findings, they propose a model that explains how the N protein binding to the Ubl1 domain of Nsp3 leads to the dissociation of viral ribonucleoprotein complexes.

Our phylogeography-based exploration revealed a notable correlation between mutations in the region [792-940] and precipitation. As a reproducible phylogeographic platform, aPhyloGeo-Covid offers the potential to expand the sample size for further investigation and facilitates the comparability of analysis results.

In addition of correlation analysis between correlation diversity of subfregment of gene and climate condition, we also inferred the ORF1a phylogeny and window regions 792-940 of ORF1a using the RAxML-NG method [39], and then conducted a detailed horizontal gene transfer (HGT) and recombination analyses (See Figure 7) using the HGT-Detection program available on the T-Rex web server [46]. The HGT-Detection program allows one to infer all possible horizontal gene transfer events for a given group of species by reconciling the species tree (i.e. ORF1a gene tree in our case) with different gene phylogenies built for regions of individual genes [47] [48]. Significantly, every identified horizontal gene transfer event can be understood from three perspectives: Firstly, it may signify a distinct complete or partial HGT occurrence between genetically distant species. Secondly, it could indicate the occurrence of parallel evolution, where the involved species underwent similar genetic changes independently. Lastly, it could also indicate the emergence of a new species (referred to as a gene transfer recipient) resulting from the recombination of the donor species genome with that of a neighboring recipient in the species' evolutionary history [49].

The minimum-cost transfer scenario with five HGTs necessary to reconcile the variants and gene phylogenies is shown in Figure 7 (HGTs are depicted by numbered arrows). The analysis initially

TABLE 2: *Putative horizontal gene transfer events in the window regions of 792-940 residue (amino acid sequences) of 14 SARS-Cov-2 variants. Each iteration of the horizontal gene transfer (HGT) algorithm is accompanied by the Robinson-Foulds distance (RF) and bipartition distance (BD) values were calculated. The HGT transfer occurs from the origin of the subtree to the destination of the subtree.*

Iteration	RF distance	BD distance	Origin Subtree	Destination Subtree
1	10	7.5	QWP89176	WAV10885
2	6	3.5	QLL26045	(QPJ77309, QWP89176, WAV10885)
3	4	2.5	UJS65740	(QLL26045, QPJ77309, QPN02498, QWP89176, UJW68561, WAV10885)
4	2	1.5	UTZ18977	UNF85490
5	0	0.0	(UNF85490, UTZ18977)	UZC43805

measured the Robinson and Foulds distance (RF) between the phylogenetic tree of ORF1a and the inferred phylogenetic trees of the window regions 792-940 of ORF1a, yielding a dissimilarity of 16. Five iterations led to the identification of HGT events (See Table 2): the first iteration detected an HGT from subtree QWP89176 to subtree WAV10885 (RF = 10 and BD = 7.5), followed by an HGT from subtree QLL26045 to subtrees QPJ77309, QWP89176, and WAV10885 (RF = 6 and BD = 3.5). The third iteration revealed an HGT from subtree UJS65740 to subtrees QLL26045, QPJ77309, QPN02498, QWP89176, UJW68561, and WAV10885 (RF = 4 and BD = 2.5). In the fourth iteration, an HGT event occurred from subtree UTZ18977 to subtree UNF85490 (RF = 2 and BD = 1.0). Finally, the fifth iteration showed an HGT from subtrees UNF85490 and UTZ18977 to subtree UZC43805 (RF = 0 and BD = 0.0). Overall, five HGT events were identified throughout the analysis.

Conclusions and future work

This project demonstrates the creation of an open-source, interactive platform that aims to enhance phylogeographic research. By integrating graph databases and a modularized Snakemake workflow, the platform effectively addresses the challenges posed by manual tools, streamlining the extraction, validation, and integration of genetic and environmental data. The platform primarily focuses on advancing the analysis of geographic and environmental data associated with SARS-CoV-2.

The utilization of the platform leads to the accumulation of diverse findings, contributed by researchers conducting various analyses. As more researchers join the platform, this network of data sources and analysis outputs continues to expand. The centralized database acts as a repository, providing researchers with access to a wide range of results and facilitating exploration and knowledge sharing within the scientific community. Although the platform is currently undergoing testing, it is expected that the interconnectedness of analyses will increase as the platform gains popularity and attracts more researchers. This network enables researchers to compare their findings and identify meaningful patterns. Overall, the platform facilitates the dissemination of research findings, encourages collaboration and building upon each

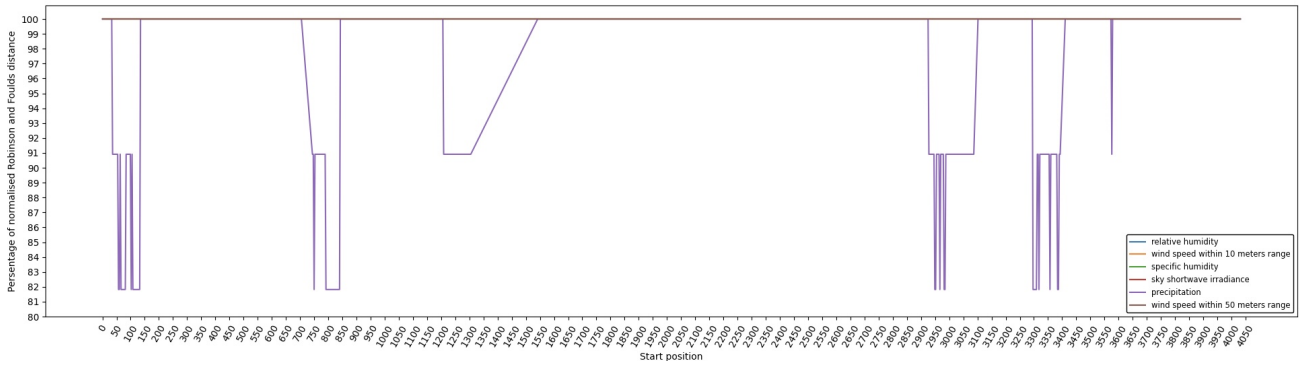

Fig. 6: *Variation of normalized Robinson and Foulds (RF) distance on the Multiple Sequence Alignment (MSA) for different climate facters. A sliding window approach with a window size of 100 residues and a step size of 3 residues was applied. X-axis indicates the start position of sliding windows on the MSA. Various colors represent six analysed climate factors which are relative humidity (blue), specific humidity (green), wind speed within 10 meters range (yellow), wind speed within 50 meters range (brown), sky shortwave irradiance (red), and precipitation (purple).*

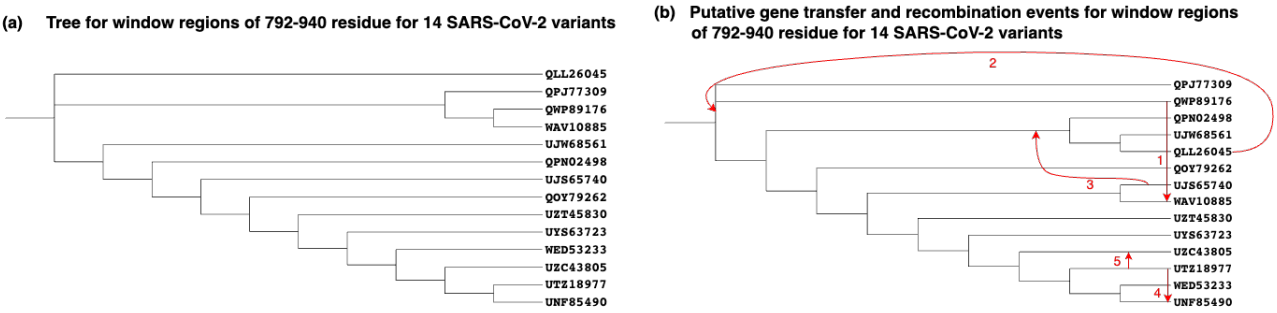

Fig. 7: *Putative horizontal gene transfer events found for the window regions of 792-940 residue (amino acid sequences) of 14 SARS-Cov-2 variants. (a) presents the phylogenetic tree of the window regions 792-940 of ORF1a. (b) presents the phylogenetic tree of ORF1a (amino acid sequences) with putative horizontal gene transfers mapped into it.*

previous work, and fosters a sense of community and scientific advancement.

To further enhance aPhyloGeo-Covid, several potential avenues for improvement can be explored:

1) Expanding the scope of available data resources, with a specific focus on augmenting geographic and environmental data. By enriching and diversifying the dataset, the aPhyloGeo-Covid project can unlock greater potential to uncover valuable insights regarding the dynamics of SARS-CoV-2 transmission and its intricate relationship with geographical and environmental variables.
2) Broadening the scope of phylogeographic analysis and comprehensively investigating the evolutionary dynamics and spatial spread of the virus can be achieved by expanding the existing pipeline of aPhyloGeo-Covid. In addition to the current pipeline, which focuses on exploring the correlation between specific genes or gene fragments and their geographic distribution, incorporating additional phylogeographic analysis workflows is recommended. By incorporating a diverse range of analysis approaches, aPhyloGeo-Covid can offer a more extensive toolkit for studying the evolutionary dynamics and spatial dissemination of SARS-CoV-2. This expanded toolkit will contribute to a more comprehensive understanding of the virus and its transmission patterns.
3) To meet the increasing research demands and accommodate larger datasets, prioritizing scalability and efficiency is crucial in the development of aPhyloGeo-Covid. Enhancing scalability and efficiency will enable the platform to handle substantial volumes of data while maintaining optimal performance. This capability is vital for researchers and public health practitioners, as it ensures fast and reliable analyses, even as the data continues to grow. By ensuring scalability and efficiency, aPhyloGeo-Covid can effectively support decision-making processes and provide valuable insights into the spatial spread and evolution of SARS-CoV-2.

Acknowledgements

The authors thank SciPy conference and reviewers for their valuable comments on this paper. This work was supported by the Natural Sciences and Engineering Research Council of Canada, the Université de Sherbrooke grant, and the Centre de recherche en écologie de l'Université de Sherbrooke (CREUS).

References

[1] S. Dellicour, C. Troupin, F. Jahanbakhsh, A. Salama, S. Massoudi, M. K. Moghaddam, G. Baele, P. Lemey, A. Gholami, and H. Bourhy, "Using phylogeographic approaches to analyse the dispersal history, velocity and direction of viral lineages—application to rabies virus spread in iran," *Molecular ecology*, vol. 28, no. 18, pp. 4335–4350, 2019, https://doi.org/10.1111/mec.15222.

[2] C. B. Vogels, D. E. Brackney, A. P. Dupuis, R. M. Robich, J. R. Fauver, A. F. Brito, S. C. Williams, J. F. Anderson, C. B. Lubelczyk, R. E. Lange *et al.*, "Phylogeographic reconstruction of the emergence and spread of powassan virus in the northeastern united states," *Proceedings of the National Academy of Sciences*, vol. 120, no. 16, p. e2218012120, 2023, https://doi.org/10.1073/pnas.2218012120.

[3] G. Franzo, G. Faustini, M. Legnardi, M. Cecchinato, M. Drigo, and C. M. Tucciarone, "Phylodynamic and phylogeographic reconstruction of porcine reproductive and respiratory syndrome virus (prrsv) in europe: Patterns and determinants," *Transboundary and Emerging Diseases*, vol. 69, no. 5, pp. e2175–e2184, 2022, https://doi.org/10.1111/tbed.14556.

[4] A. Munsey, F. N. Mwiine, S. Ochwo, L. Velazquez-Salinas, Z. Ahmed, F. Maree, L. L. Rodriguez, E. Rieder, A. Perez, S. Dellicour *et al.*, "Phylogeographic analysis of foot-and-mouth disease virus serotype o dispersal and associated drivers in east africa," *Molecular ecology*, vol. 30, no. 15, pp. 3815–3825, 2021, https://doi.org/10.1111/mec.15991.

[5] E. C. Holmes, "The phylogeography of human viruses," *Molecular ecology*, vol. 13, no. 4, pp. 745–756, 2004, https://doi.org/10.1046/j.1365-294X.2003.02051.x.

[6] Á. O'Toole, V. Hill, O. G. Pybus, A. Watts, I. I. Bogoch, K. Khan, J. P. Messina, T. COVID, B.-U. C. G. Network, H. Tegally *et al.*, "Tracking the international spread of sars-cov-2 lineages b. 1.1. 7 and b. 1.351/501y-v2 with grinch," *Wellcome Open Research*, vol. 6, 2021, https://doi.org/10.12688/wellcomeopenres.16661.2.

[7] J. R. Brister, D. Ako-Adjei, Y. Bao, and O. Blinkova, "Ncbi viral genomes resource," *Nucleic acids research*, vol. 43, no. D1, pp. D571–D577, 2015, https://doi.org/10.1093/nar/gku1207.

[8] S. Khare, C. Gurry, L. Freitas, M. B. Schultz, G. Bach, A. Diallo, N. Akite, J. Ho, R. T. Lee, W. Yeo *et al.*, "Gisaid's role in pandemic response," *China CDC weekly*, vol. 3, no. 49, p. 1049, 2021, https://doi.org/10.46234/ccdcw2021.255.

[9] O. A. Marzouk, "Assessment of global warming in al buraimi, sultanate of oman based on statistical analysis of nasa power data over 39 years, and testing the reliability of nasa power against meteorological measurements," *Heliyon*, vol. 7, no. 3, p. e06625, 2021, https://doi.org/10.1016/j.heliyon.2021.e06625.

[10] E. Mathieu, H. Ritchie, E. Ortiz-Ospina, M. Roser, J. Hasell, C. Appel, C. Giattino, and L. Rodés-Guirao, "A global database of covid-19 vaccinations," *Nature human behaviour*, vol. 5, no. 7, pp. 947–953, 2021, https://doi.org/10.1038/s41562-021-01122-8.

[11] J. E. McCormack, S. M. Hird, A. J. Zellmer, B. C. Carstens, and R. T. Brumfield, "Applications of next-generation sequencing to phylogeography and phylogenetics," *Molecular phylogenetics and evolution*, vol. 66, no. 2, pp. 526–538, 2013, https://doi.org/10.1016/j.ympev.2011.12.007.

[12] A. McGaughran, L. Liggins, K. A. Marske, M. N. Dawson, L. M. Schiebelhut, S. D. Lavery, L. L. Knowles, C. Moritz, and C. Riginos, "Comparative phylogeography in the genomic age: Opportunities and challenges," *Journal of Biogeography*, vol. 49, no. 12, pp. 2130–2144, 2022, https://doi.org/10.1111/jbi.14481.

[13] J. Guia, V. G. Soares, and J. Bernardino, "Graph databases: Neo4j analysis." in *ICEIS (1)*, 2017, pp. 351–356, https://doi.org/10.5220/0006356003510356.

[14] S. Timón-Reina, M. Rincón, and R. Martínez-Tomás, "An overview of graph databases and their applications in the biomedical domain," *Database*, vol. 2021, 2021, https://doi.org/10.1093/database/baab026.

[15] W. Laddada, C. Zanni-Merk, and L. F. Soualmia, "Analyzing sars-cov-2 sequence patterns by semantic trajectories." *Stud Health Technol Inform*, pp. 197–200, 2022, https://doi.org/10.3233/SHTI220696.

[16] R. Angles, "A comparison of current graph database models," in *2012 IEEE 28th International Conference on Data Engineering Workshops*. IEEE, 2012, pp. 171–177, https://doi.org/10.1109/ICDEW.2012.31.

[17] C. Vicknair, M. Macias, Z. Zhao, X. Nan, Y. Chen, and D. Wilkins, "A comparison of a graph database and a relational database: a data provenance perspective," in *Proceedings of the 48th annual Southeast regional conference*, 2010, pp. 1–6, https://doi.org/10.1145/1900008.1900067.

[18] O. Uphyrkina, W. E. Johnson, H. Quigley, D. Miquelle, L. Marker, M. Bush, and S. J. O'Brien, "Phylogenetics, genome diversity and origin of modern leopard, panthera pardus," *Molecular ecology*, vol. 10, no. 11, pp. 2617–2633, 2001, https://doi.org/10.1046/j.0962-1083.2001.01350.x.

[19] S.-J. Luo, J.-H. Kim, W. E. Johnson, J. v. d. Walt, J. Martenson, N. Yuhki, D. G. Miquelle, O. Uphyrkina, J. M. Goodrich, H. B. Quigley *et al.*, "Phylogeography and genetic ancestry of tigers (panthera tigris)," *PLoS biology*, vol. 2, no. 12, p. e442, 2004, https://doi.org/10.1371/journal.pbio.0020442.

[20] D. J. Taylor, S. J. Connelly, and A. A. Kotov, "The intercontinental phylogeography of neustonic daphniids," *Scientific Reports*, vol. 10, no. 1, p. 1818, 2020, https://doi.org/10.1038/s41598-020-58743-8.

[21] M. A. Aziz, O. Smith, H. A. Jackson, S. Tollington, S. Darlow, A. Barlow, M. A. Islam, and J. Groombridge, "Phylogeography of panthera tigris in the mangrove forest of the sundarbans," *Endangered Species Research*, vol. 48, pp. 87–97, 2022, https://doi.org/10.3354/esr01188.

[22] A. Koshkarov, W. Li, M.-L. Luu, and N. Tahiri, "Phylogeography: Analysis of genetic and climatic data of sars-cov-2," 2022, https://doi.org/10.25080/majora-212e5952-018.

[23] D. F. Robinson and L. R. Foulds, "Comparison of phylogenetic trees," *Mathematical biosciences*, vol. 53, no. 1-2, pp. 131–147, 1981, https://doi.org/10.1016/0025-5564(81)90043-2.

[24] J. Köster and S. Rahmann, "Snakemake—a scalable bioinformatics workflow engine," *Bioinformatics*, vol. 28, no. 19, pp. 2520–2522, 2012, https://doi.org/10.1093/bioinformatics/bty350.

[25] V. Jalili, E. Afgan, Q. Gu, D. Clements, D. Blankenberg, J. Goecks, J. Taylor, and A. Nekrutenko, "The galaxy platform for accessible, reproducible and collaborative biomedical analyses: 2020 update," *Nucleic acids research*, vol. 48, no. W1, pp. W395–W402, 2020, https://doi.org/10.1093/nar/gkaa434.

[26] V. Spišaková, L. Hejtmánek, and J. Hynšt, "Nextflow in bioinformatics: Executors performance comparison using genomics data," *Future Generation Computer Systems*, 2023, https://doi.org/10.1016/j.future.2023.01.009.

[27] L. Wratten, A. Wilm, and J. Göke, "Reproducible, scalable, and shareable analysis pipelines with bioinformatics workflow managers," *Nature methods*, vol. 18, no. 10, pp. 1161–1168, 2021, https://doi.org/10.1038/s41592-021-01254-9.

[28] P. J. Cock, T. Antao, J. T. Chang, B. A. Chapman, C. J. Cox, A. Dalke, I. Friedberg, T. Hamelryck, F. Kauff, B. Wilczynski *et al.*, "Biopython: freely available python tools for computational molecular biology and bioinformatics," *Bioinformatics*, vol. 25, no. 11, pp. 1422–1423, 2009, https://doi.org/10.1093/bioinformatics/btp163.

[29] Wes McKinney, "Data Structures for Statistical Computing in Python," in *Proceedings of the 9th Python in Science Conference*, Stéfan van der Walt and Jarrod Millman, Eds., 2010, pp. 56 – 61, https://doi.org/10.25080/Majora-92bf1922-00a.

[30] P. Lemenkova, "Processing oceanographic data by python libraries numpy, scipy and pandas," *Aquatic Research*, vol. 2, no. 2, pp. 73–91, 2019, https://doi.org/10.3153/AR19009.

[31] E. L. Hatcher, S. A. Zhdanov, Y. Bao, O. Blinkova, E. P. Nawrocki, Y. Ostapchuck, A. A. Schäffer, and J. R. Brister, "Virus variation resource–improved response to emergent viral outbreaks," *Nucleic acids research*, vol. 45, no. D1, pp. D482–D490, 2017, https://doi.org/10.1093/nar/gkw1065.

[32] S. Hossain, C. Calloway, D. Lippa, D. Niederhut, and D. Shupe, "Visualization of bioinformatics data with dash bio," in *Proceedings of the 18th Python in Science Conference*, vol. 126. SciPy, Austin, Texas, pp. 126–133, 2019, p. 133, https://doi.org/10.25080/Majora-7ddc1dd1-012.

[33] V. Liermann and S. Li, "Dynamic dashboards," in *The Digital Journey of Banking and Insurance, Volume II: Digitalization and Machine Learning*. Springer, 2021, pp. 155–180, https://doi.org/10.1007/978-3-030-78829-2_9.

[34] G. Jordan and G. Jordan, "Neo4j+ python," *Practical Neo4j*, pp. 169–213, 2014, https://doi.org/10.1007/978-1-4842-0022-3_9.

[35] R. C. Edgar and S. Batzoglou, "Multiple sequence alignment," *Current opinion in structural biology*, vol. 16, no. 3, pp. 368–373, 2006, https://doi.org/10.1016/j.sbi.2006.04.004.

[36] K. Katoh and D. M. Standley, "Mafft multiple sequence alignment software version 7: improvements in performance and usability," *Molecular biology and evolution*, vol. 30, no. 4, pp. 772–780, 2013, https://doi.org/10.1093/molbev/mst010.

[37] N. Saitou and M. Nei, "The neighbor-joining method: a new method for reconstructing phylogenetic trees." *Molecular biology and evolution*, vol. 4, no. 4, pp. 406–425, 1987, https://doi.org/10.1093/oxfordjournals.molbev.a040454.

[38] R. Mihaescu, D. Levy, and L. Pachter, "Why neighbor-joining works," *Algorithmica*, vol. 54, pp. 1–24, 2009, https://doi.org/10.1007/s00453-007-9116-4.

[39] A. M. Kozlov, D. Darriba, T. Flouri, B. Morel, and A. Stamatakis, "Raxml-ng: a fast, scalable and user-friendly tool for maximum likelihood phylogenetic inference," *Bioinformatics*, vol. 35, no. 21, pp. 4453–4455, 2019, https://doi.org/10.1093/bioinformatics/btz305.

[40] M. N. Price, P. S. Dehal, and A. P. Arkin, "Fasttree: computing large minimum evolution trees with profiles instead of a distance matrix," *Molecular biology and evolution*, vol. 26, no. 7, pp. 1641–1650, 2009, https://doi.org/10.1093/molbev/msp077.

[41] S. Van Der Walt, S. C. Colbert, and G. Varoquaux, "The numpy array: a structure for efficient numerical computation," *Computing in science & engineering*, vol. 13, no. 2, pp. 22–30, 2011, https://doi.org/10.1109/MCSE.2011.37.

[42] J. Bernard and J. Bernard, "Python data analysis with pandas," *Python Recipes Handbook: A Problem-Solution Approach*, pp. 37–48, 2016, https://doi.org/10.1007/978-1-4842-0241-8_5.

[43] M. T. Khan, M. Irfan, H. Ahsan, A. Ahmed, A. C. Kaushik, A. S. Khan, S. Chinnasamy, A. Ali, and D.-Q. Wei, "Structures of sars-cov-2 rna-binding proteins and therapeutic targets," *Intervirology*, vol. 64, no. 2, pp. 55–68, 2021, https://doi.org/10.1159/000513686.

[44] A. Waterhouse, M. Bertoni, S. Bienert, G. Studer, G. Tauriello, R. Gumienny, F. T. Heer, T. A. P. de Beer, C. Rempfer, L. Bordoli *et al.*, "Swiss-model: homology modelling of protein structures and complexes," *Nucleic acids research*, vol. 46, no. W1, pp. W296–W303, 2018, https://doi.org/10.1093/nar/gky427.

[45] X. Ni, Y. Han, R. Zhou, Y. Zhou, and J. Lei, "Structural insights into ribonucleoprotein dissociation by nucleocapsid protein interacting with non-structural protein 3 in sars-cov-2," *Communications Biology*, vol. 6, no. 1, p. 193, 2023, https://doi.org/10.1038/s42003-023-04570-2.

[46] A. Boc, A. B. Diallo, and V. Makarenkov, "T-rex: a web server for inferring, validating and visualizing phylogenetic trees and networks," *Nucleic acids research*, vol. 40, no. W1, pp. W573–W579, 2012, https://doi.org/10.1093/nar/gks485.

[47] A. Boc and V. Makarenkov, "Towards an accurate identification of mosaic genes and partial horizontal gene transfers," *Nucleic acids research*, vol. 39, no. 21, pp. e144–e144, 2011, https://doi.org/10.1093/nar/gkr735.

[48] E. Denamur, G. Lecointre, P. Darlu, O. Tenaillon, C. Acquaviva, C. Sayada, I. Sunjevaric, R. Rothstein, J. Elion, F. Taddei *et al.*, "Evolutionary implications of the frequent horizontal transfer of mismatch repair genes," *Cell*, vol. 103, no. 5, pp. 711–721, 2000, https://doi.org/10.1016/S0092-8674(00)00175-6.

[49] V. Makarenkov, B. Mazoure, G. Rabusseau, and P. Legendre, "Horizontal gene transfer and recombination analysis of sars-cov-2 genes helps discover its close relatives and shed light on its origin," *BMC ecology and evolution*, vol. 21, pp. 1–18, 2021, https://doi.org/10.1186/s12862-020-01732-2.

Pandera: Going Beyond Pandas Data Validation

Niels Bantilan[‡§*]

Abstract—Data quality remains a core concern for practitioners in machine learning, data science, and data engineering, and many specialized packages have emerged to fulfill the need of validating and monitoring data and models. However, as the open source community creates new data processing frameworks - notably, new highly performant entrants such as Polars - existing data quality frameworks need to catch up to support them, and in some cases, the Python community more broadly creates new data validation libraries for these new data frameworks. This paper outlines pandera's motivation and challenges that took it from being a pandas-only data validation framework [1] to one that is extensible to other non-pandas-compliant dataframe-like libraries. It also provides an informative case study of the technical and organizational challenges associated with expanding the scope of a library beyond its original boundaries.

Index Terms—data validation, data testing, data science, machine learning, data engineering

Introduction

Data validation is the process of falsifying data against a particular set of assumptions [2]. Framed differently, it is the act of verifying data against a set of properties and constraints that are explicitly established by the data practitioner. In this context, the term "data practitioner" refers to anyone using a programming language to analyze, transform, or otherwise process data. It includes, but is not limited to, data scientists, data engineers, data analysts, machine learning engineers, and machine learning researchers. This paper describes the trajectory of pandera from a pandas-only validation library to a more generic framework that can validate any dataframe-like object.

Origins

Pandera started as a small project in 2018 with the goal of providing a lightweight, flexible, and expressive API to validate pandas DataFrames [3]. This introductory section provides a brief primer on data validation with pandera, providing insights into how its design facilitates code-first schema authoring and maintenance. The operating assumption is that this, in turn, gives rise to safer and more robust data pipelines.

Why Validate Data?

As stated in the introduction, data validation is the act of falsifying (or verifying) data against a particular set of assumptions, expressed as a schema of validation rules. These rules are explicitly established by the data practitioner without interference from automated processes, like data profiling, and verified at runtime on real-world data.

In machine learning (ML) and statistical analysis use cases, this is critical because invalid data, e.g. incorrect types, invalid values, and otherwise corrupted data, can pass silently along a data pipeline and propagate those errors to various endpoints, which cause adverse ripple-effects to the downstream consumers that rely on high-quality data. These endpoints can be models, analyses, and visualizations, and errors in any of these artifacts call into question the trustworthiness of the conclusions that they entail. Though this is especially important in scientific research and business-critical applications, data validation ought to be a core part of the quality assurance pipeline of data teams.

Data practitioners build statistical domain knowledge about the data they are working with by inspecting the data via exploratory data analysis (EDA), data profiling tools, or a combination of these two approaches. By building a mental model of how their data looks like and envisioning a set of constraints that express what the ideal "clean" dataset looks like, the data practitioner can then encode this understanding as a schema that they can use to validate new incoming data. This schema serves not only as documentation for themselves and future maintainers, but also as a stateless data drift monitoring system for data transformation, model training, and production inference pipelines. The benefit of this statelessness is that the data practitioner can reason about what counts as valid data through their code and their version control system of choice, which captures changes in the assumptions about valid data over time.

However, the process of writing down these schemas is a laborious and often thankless task and not as exciting as getting to the modeling/analysis/visualization part of the development process. As stated in [1], to lower the barrier to explicitly writing down schemas for maintaining data quality, pandera was created with the following design principles in mind:

1) Expressing schemas as code should feel familiar to pandas users.
2) Data validation should be compatible with the different workflows and tools in in the data science and ML stack without a lot of setup or configuration.
3) Defining custom validation rules should be easy.
4) The validation interface should make the debugging process easier.
5) Integration with existing code should be as seamless as possible.

These principles were codified to guide the development of pandera project towards ease of learning and incremental adoption.

* *Corresponding author: niels@union.ai*
‡ *Union.ai*
§ *pyOpenSci*

Pandera's Programming Model

With these principles in mind, pandera sought to be minimally invasive, quick to integrate into existing data science and ML codebases, and easy to learn for data scientists, data engineers, and ML engineers who use Python (refer to the *Related Tools* section of [1] for a discussion of similar projects in the Python space). The original object-based syntax makes it clear how defining a DataFrameSchema is similar to defining pandas DataFrames:

```
import pandera as pa

schema = pa.DataFrameSchema({
    "column1": pa.Column(
        int, pa.Check.gt(0)
    ),
    "column2": pa.Column(
        str, pa.Check.isin([*"ABC"])
    ),
    "column3": pa.Column(
        float,
        pa.Check.in_range(
            min_value=0.0,
            max_value=1.0,
        )
    ),
})
```

In the example above, we expect our data to have three columns that have specific names, data types, and data value constraints. By reading the code the data practitioner themselves or their collaborators can immediately see what the minimum requirements are for valid data. For example, the `pa.Check.gt(0)` constraint indicates that `column1` just always be greater than zero.

Pandera emphasizes code-first schema authoring and maintenance. As opposed to yaml-, json- or UI-based schema authoring, code-first schemas lower the barrier for DS/ML practitioners to create and maintain these schemas because they don't have to learn a DSL or a set of entirely new concepts.

The hypothesis was that this would give rise to safer and more robust data pipelines in different parts of the data ecosystem: from research projects in academia, to nonprofits seeking to create valuable data assets, or to industry practitioners who want to use pandas in a production ETL pipeline. Pandera's core programming model is simple:

Pandera embraces the data testing development process, which involves validating real data as well as the functions that produce them. The process of developing data pipelines with data testing in mind involves the iterative definition of both data transformations and schemas, which can be used as "fancy assertions" in your code, or as reusable components in the pipeline's unit test suite.

As depicted in 1, this process is roughly as follows: by whatever means necessary, typically via EDA or data profiling (the programmatic creation of summary statistics and visualization), the data practitioner arrives at a schema, which states the columns and properties that the data should adhere to. The schema is then used to validate data in-line, or at the interface boundary of critical functions in the data pipeline. The data practitioner can start with a basic schema, which may include column names and their expected types. As they build more statistical domain knowledge about what counts as valid data, the can refine the schemas to better fit the requirements of their analysis using `Checks`.

```
import pandas as pd

# inline validation
data = pd.DataFrame({
    "column1": [1, 2, 3],
    "column2": ["A", "B", "C"],
    "column3": [0.2, 0.41, 0.87],
})
schema.validate(data)

# validating the input-output function boundary
@pa.check_input(schema)
def transform(data):
    ...

# pandera automatically validates the input
# when the transform function is called
transform(data)
```

If validation succeeds, the schema returns the valid data. If it fails, pandera raises a SchemaError or SchemaErrors exception. These exceptions contain metadata about what caused the failure at varying levels of granularity: either at the schema-level, e.g. wrong column types, or at the data-value-level, e.g. numbers being out of range:

```
invalid_data = pd.DataFrame({
    "column1": [1, -1, 3],
    "column2": ["A", "B", "D"],
    "column3": [0.2, 0.41, 100.0],
})
# try to validate as all columns and constraints
# before raising an error with lazy=True
try:
    schema.validate(data, lazy=True)
except pa.errors.SchemaErrors as exc:
    print("Failure cases")
    print(
        exc.failure_cases[
            ["column", "failure_case", "index"]
        ]
    )

# Output:
Failure cases
    column failure_case  index
0  column1           -1      1
1  column2            D      2
2  column3        100.0      2
```

The exception raised during validation contains several attributes, including the original failed data in the `.data` attribute, but more importantly, it contains a normalized DataFrame view of all the failure cases in the data via the `.failure_cases` attribute. This is reported at the most granular level so that the data practitioner can quickly understand what's wrong with their data.

Evolution

After its first set of releases, pandera continued to improve with bug fixes, feature enhancements, and documentation improvements. This section highlights four major events in pandera's development. In chronological order, these events were: documentation improvements, support for a class-based API, data synthesis strategies, and the pandera type system.

Documentation Improvements

Documentation is one of the most critical pieces to any software project. Even if the underlying code is well-written, performant, and useful, ultimately if the documentation is unclear or otherwise difficult to read and navigate, the software itself will be inaccessible to end users.

The first set of major contributions came with the help of Nigel Markey, who helped considerably in documentation efforts, making pandera easy to learn and adopt through examples, tutorials, and a comprehensive API reference. This helped pandera to

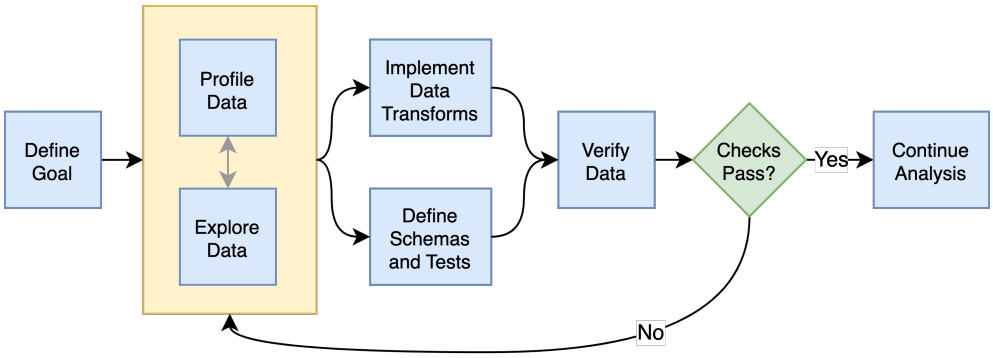

Fig. 1: *The pandera programming model is an iterative loop between building statistical domain knowledge, implementing data transforms and schemas, and verifying data.*

become part of pyOpenSci [4], which helped further improve its quality and usability through further review and refinement.

Class-based API

The second major improvement in pandera was contributed by Jean-Francois Zinque, who implemented the class-based syntax that's more akin to Python dataclasses and the pydantic library [5]. This modernized pandera to use syntax that was familiar to developers who use classes as types to express the form and properties of the data structures they want to use.

```
class Model(pa.DataFrameModel):
   column1: int = pa.Field(gt=0, lt=100)
   column2: str = pa.Field(isin=[*"ABC"])
   column3: float = pa.Field(
       in_range={"min_value": 0.0, "max_value": 1.0}
   )
```

This also enabled pandera to take advantage of type hints as a convenient way of expressing the input-output types of a function and enforcing data quality at runtime.

```
from pandera.typing import DataFrame

class Input(pa.DataFrameModel):
    x: float
    y: float

class Output(Input):
    z: float

    @pa.dataframe_check
    def check_z(cls, df):
        """Column z must be the sum of x and y."""
        return df["z"] == (df["x"] + df["y"])

# This decorator does runtime checks on the
# input and output dataframe.
@pa.check_types
def fn(data: DataFrame[Input]) -> DataFrame[Output]:
    return data.assign(z=lambda df: df.x + df.y)
```

Data Synthesis Strategies

The third major improvement was adding support for data synthesis strategies using the hypothesis library [6]. This expanded pandera's scope from a data validation library to a "data testing" toolkit by allowing the data practitioner to easily create mock data for testing not only real data, but the functions that produce/clean/transform the data. Note that the `hypothesis` library for doing property-based testing is not to be confused with statistical `Hypothesis` checks, which were already supported by pandera.

```
import pytest
from hypothesis import given

# This will generate data for testing the correct
# implementation of fn
@given(Input.strategy(size=3))
def test_fn(input_data)
    fn(input_data)

class WrongInput(pa.DataFrameModel):
    a: int
    b: str

# This will fail on the output check
@given(WrongInput.strategy(size=3))
def test_fn_wrong_input(input_data)
    with pytest.raises(pa.SchemaError):
        fn(input_data)
```

Hypothesis handles generating valid data under the pandera schema's constraints, which relieves the developer from manually hand-crafting dataframes and allows unit tests to catch edge cases that would not otherwise be caught by the hand-crafted test cases. This can be seamlessly integrated with `pytest`, since one can think of pandera schemas as essentially "fancy assertion" statements.

Pandera Type System

Finally, the fourth major improvement was contributed by Jean-Francois Zinque, who implemented pandera's type system, which provides a consistent interface for defining semantic and logical types not only for pandas, but also potentially for other dataframe libraries like pyspark and modin.

This allows pandera users to, for example, implement an `IPAddress` type, which requires both specifying the data type and checking the actual values of the data to verify:

```
import re
from typing import Optional, Iterable, Union
from pandera import dtypes
from pandera.engines import pandas_engine


@pandas_engine.Engine.register_dtype
@dtypes.immutable
class IPAddress(pandas_engine.NpString):

    REGEX = re.compile(
        r"(\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3})"
    )

    def check(
        self,
        pandera_dtype: dtypes.DataType,
```

```
        data_container: Optional[pd.Series] = None,
    ) -> Union[bool, Iterable[bool]]:
        # ensure that the data container's data
        # type is correct
        correct_type = super().check(pandera_dtype)
        if not correct_type:
            return correct_type
         if data_container is None:
             raise ValueError

        # ensure IP address pattern
        return data_container.map(
            lambda x: self.REGEX.match(x) is not None
        )

# using it in a DataFrame model
class IPAddressModel(pa.DataFrameModel):
    ip_address: IPAddress
```

Expanding Scope

After gaining traction over the years, the author, the contributors, and the growing community of pandera users also began to expand pandera's scope to support pandas-compliant data frameworks such as GeoPandas [7], Dask [8], Modin [9], and Pyspark Pandas [10] (formerly Koalas). As requests for other dataframe-like libraries increased in frequency, it became clear that pandera in its existing state was not well-suited for extension beyond Pandas objects.

Design Weaknesses

The fundamental design flaw in pandera's internals was that the schema specification and validation engine were interleaved through out the code base. This presented the following challenges for supporting non-pandas dataframe libraries:

- **Schemas were strongly coupled to pandas**: The schema class had a lot of assumptions about pandas, which manifested as method calls and operations that assumed that pandera was operating on a pandas DataFrame.
- **Checks were strongly coupled to pandas**: Pandera has core checks that are exposed in the schema/schema component object, which were all implemented with pandas-specific code.
- **Error reporting assumed in-memory data**: Error reporting of metadata and value checks assumed in-memory, small-to-medium-sized datasets. For any larger scale data that requires a distributed dataframe, the error-reporting mechanism doesn't work well because the worst case scenario of all data values being invalid would produce an failure case report that was potentially even larger than the original data.
- **Leaky abstractions**: The pandera schema API leaked certain pandas-specific abstractions, e.g. Index and MultiIndex, which don't apply to other frameworks, e.g. Spark and Polars.

These weaknesses were uncovered after-the-fact, when the author and contributors analyzed the existing codebase to determine how to best support other dataframe objects.

Design Strengths

With these limitations in mind, it's also important to note some of the design choices that significantly eased the subsequent internals rewrite. In particular:

- **Generic schema interface**: Within the domain of tabular, dataframe-like datastructures, pandera's schema API is generic enough to support both columnar and row-wise statistical data objects, which can be defined as objects that expose methods for statistical analysis.
- **Flexible Check abstraction**: pandera's `Check` object — the core validator abstraction — was sufficiently flexible. Check functions assume that it returns a boolean scalar, Series or DataFrame. This allows data pandera to report value errors at varying levels of granularity: e.g. for distributed dataframes, reporting all failure cases incurs unacceptable overhead for distributed dataframes, which would require full table scans.
- **Flexible type system**: The type system was also sufficiently flexible to support types for different dataframe libraries, allowing for simple types, generic types, parameterized types, and logical types.

Rewriting Pandera Internals

For practical purposes, the first set of DataFrame libraries supported by pandera were pandas-compliant frameworks such as GeoPandas, Modin, Dask, and Koalas (now `pyspark.pandas`). Even though these libraries do deviate somewhat from the pandas API, they were close enough such that the parts of the pandas API that pandera leveraged were just a subset of the full API. Therefore, supporting these additional libraries required only a few code changes [11]. This approach was the path to least resistance for making data validation more scalable, and validating the notion that the community would actually find it useful.

In contrast, in order to support additional non-pandas-compliant libraries like pyspark, polars, and vaex, pandera needed to overhaul the schema objects by decoupling the schema specification from the validation engine. At a high-level, the approach was to introduce the following abstractions:

- A `pandera.api` subpackage, which contains the schema specification that defines the properties of an underlying data structure.
- A `pandera.backends` subpackage, which leverages the schema specification and implements the actual validation logic.
- A backend registry, which maps a particular API specification to a backend based on the DataFrame type being validated.
- A common type-aware Check namespace and registry, which registers type-specific implementations of built-in checks and allows contributors to easily add new built-in checks.

This new architecture allows contributors to implement a schema validator for any data structure they want. In pseudo-code, supporting a fictional dataframe library called `sloth` would look something like this:

```
import sloth as sl
from pandera.api.base.schema import BaseSchema
from pandera.backends.base import BaseSchemaBackend

class DataFrameSchema(BaseSchema):
    def __init__(self, **kwargs):
        # add properties that this dataframe
        # would contain

class DataFrameSchemaBackend(BaseSchemaBackend):
```

```
    def validate(
        self,
        check_obj: sl.DataFrame,
        schema: DataFrameSchema,
        *,
        **kwargs,
    ):
        # implement custom validation logic

# register the backend
DataFrameSchema.register_backend(
    sloth.DataFrame,
    DataFrameSchemaBackend,
)
```

Similarly, the built-in checks can easily be extended to support `sloth` data structures:

```
import sloth as sl

from pandera.api.extensions import register_builtin_check

@register_builtin_check(
    aliases=["eq"],
    error="equal_to({value})"
)
def equal_to(
    data: sl.Series, value: Any
) -> sl.Series:
    return data.equals(value)
```

Organizational and Development Challenges

Although the road to an internals rewrite was fairly straightforward from a technical perspective, there were additional meta-challenges that added to the complexity of implementing the rewrite in practice:

- **Multi-tasking the rewrite with PR reviews**: As with any open source project, there were community-contributed PRs for bug fixes and feature enhancements, many of which created merge conflicts since they assumed the pre-rewrite state of the code base. The author had to block such contributions until the rewrite was complete and fast-forward these PRs to fit the structure of the new code base.
- **Centralized knowledge**: Because the author was the primary maintainer of the project and was the only maintainer who understood the codebase as a whole well enough to make the changes, incorporating non-conflicting pull requests took time away from the rewrite, further delaying the timeline that would unblock other would-be contributors who wanted to implement support for other libraries, e.g. polars.
- **Informal governance**: Because pandera has an informal contributor and governance structure, the author effectively made unilateral decisions with respect to the abstractions necessary to decouple the schema specification from the validation backend. This turned out to be appropriate, with a successful case of a community-contributed `pyspark.sql` integration being almost complete as of the writing of this paper. This integration is planned for release in the next minor version `0.16.0`. However, the pandera project would benefit from a more formal governance structure involving a broader set of stakeholders when it comes to wide-sweeping internal or user-facing changes.

Retrospective

With all of these challenges in mind, the internals rewrite was completed in pull request 913 [12] on January 24th, 2023 and the follow-up pull request 1109 [13] on March 13th, 2023. A few factors facilitated the rewrite itself and also reduced the risk of regressions:

- **Unit tests**: A comprehensive unit test suite caught many issues, but not all of them. This was partly due to lack of complete test coverage, but new tests also had to be written for abstractions introduced during the re-write process.
- **Localized pandas coupling**: Pandas-specific code was mostly localized in easy-to-identify locations in the code-base.
- **Lessons learned from pandas-compliant integrations**: Earlier integrations with pandas-compliant libraries revealed operations/assumptions that are likely to break in out-of-core DataFrame libraries, which typically involved indexes and sorting assumptions.

In retrospect, there are additionally things the author would have done differently to make pandera more flexible and extensible:

- **Thoughtful design work**: With some careful design work, it would have been obvious to decouple schema specification from validation backend much sooner.
- **Library-independent error reporting**: Make error reporting more flexible by decoupling error reporting data structures from the specific DataFrame library, e.g. by using native python data structures like lists and dictionaries instead of pandas DataFrames to report failure cases.
- **Decoupling metadata from data**: Distinguish between DataFrame metadata schema errors (e.g. missing columns) and data value errors (e.g. out-of-range values).
- **Investing in governance and community**: Invest more in governance and formalize contributor and community RFC processes sooner to help with design and feature enhancement efforts.

Updated Design Principles

Given all of the developments and updates that pandera has seen in recent years, pandera's design principles also need to be updated with one amendment and one additions:

1) **Amendment**: Expressing schemas as code should feel familiar to *Python users, regardless of the dataframe library they're using*.
2) Data validation should be compatible with the different workflows and tools in in the data science and ML stack without a lot of setup or configuration.
3) Defining custom validation rules should be easy.
4) The validation interface should make the debugging process easier.
5) Integration with existing code should be as seamless as possible.
6) **Addition**: *Extending the interface to other statistical data structures should be easy using a core set of building blocks and abstractions.*

Conclusion

Pandera has evolved from a pandas-specific data validation library to a comprehensive toolkit that provides a standard schema interface for easily extending and supporting validation backends for arbitrary statistical data containers. This paper provides an

overview of data validation and testing, focusing on pandera's core programming model and its extended functionality to support property-based testing. This paper also provides a useful case study of the technical and organizational challenges associated with expanding the scope of a library beyond its original boundaries.

The author's hope is that, by highlighting the technical and organizational dimensions of this evolution, that other open source authors and maintainers can learn and avoid some of the pitfalls encountered during the internals rewrite that now enables pandera to support a whole suite of statistical data containers moving forward.

REFERENCES

[1] Niels Bantilan, "pandera: Statistical Data Validation of Pandas Dataframes," in *Proceedings of the 19th Python in Science Conference*, Meghann Agarwal, Chris Calloway, Dillon Niederhut, and David Shupe, Eds., 2020, pp. 116 – 124, https://doi.org/10.25080/Majora-342d178e-010.

[2] M. Van der Loo and E. De Jonge, *Statistical data cleaning with applications in R*. Wiley Online Library, 2018.

[3] Wes McKinney, "Data Structures for Statistical Computing in Python," in *Proceedings of the 9th Python in Science Conference*, Stéfan van der Walt and Jarrod Millman, Eds., 2010, pp. 56 – 61, https://doi.org/10.25080/Majora-92bf1922-00a.

[4] "pyopensci," accessed: 2 June 2023. [Online]. Available: https://www.pyopensci.org/

[5] "pydantic," accessed: 2 June 2023. [Online]. Available: https://docs.pydantic.dev/latest/

[6] D. MacIver, Z. Hatfield-Dodds, and M. Contributors, "Hypothesis: A new approach to property-based testing," *Journal of Open Source Software*, vol. 4, no. 43, p. 1891, 11 2019, https://doi.org/10.21105/joss.01891. [Online]. Available: http://dx.doi.org/10.21105/joss.01891

[7] K. Jordahl, J. V. den Bossche, M. Fleischmann, J. Wasserman, J. McBride, J. Gerard, J. Tratner, M. Perry, A. G. Badaracco, C. Farmer, G. A. Hjelle, A. D. Snow, M. Cochran, S. Gillies, L. Culbertson, M. Bartos, N. Eubank, maxalbert, A. Bilogur, S. Rey, C. Ren, D. Arribas-Bel, L. Wasser, L. J. Wolf, M. Journois, J. Wilson, A. Greenhall, C. Holdgraf, Filipe, and F. Leblanc, "geopandas/geopandas: v0.8.1," Jul. 2020, https://doi.org/10.5281/zenodo.3946761. [Online]. Available: https://doi.org/10.5281/zenodo.3946761

[8] Matthew Rocklin, "Dask: Parallel Computation with Blocked algorithms and Task Scheduling," in *Proceedings of the 14th Python in Science Conference*, Kathryn Huff and James Bergstra, Eds., 2015, pp. 126 – 132, https://doi.org/10.25080/Majora-7b98e3ed-013.

[9] D. Petersohn, S. Macke, D. Xin, W. Ma, D. Lee, X. Mo, J. E. Gonzalez, J. M. Hellerstein, A. D. Joseph, and A. Parameswaran, "Towards scalable dataframe systems," 2020, https://doi.org/10.48550/arXiv.2001.00888.

[10] "pysparkpandas," accessed: 2 June 2023. [Online]. Available: https://spark.apache.org/docs/latest/api/python/reference/pyspark.pandas/index.html

[11] "Modin support," accessed: 26 June 2023. [Online]. Available: https://github.com/unionai-oss/pandera/pull/660

[12] "core and backend pandera api internals rewrite," accessed: 2 June 2023. [Online]. Available: https://github.com/unionai-oss/pandera/pull/913

[13] "internals rewrite: clean up checks and hypothesis functionality," accessed: 2 June 2023. [Online]. Available: https://github.com/unionai-oss/pandera/pull/1109

libyt: a Tool for Parallel In Situ Analysis with yt

Shin-Rong Tsai[‡§*], Hsi-Yu Schive[‡¶‖**], Matthew J. Turk[§]

Abstract—In the era of exascale computing, storage and analysis of large scale data have become more important and difficult. We present `libyt`, an open source C++ library, that allows researchers to analyze and visualize data using `yt` or other Python packages in parallel during simulation runtime. We describe the code method for organizing adaptive mesh refinement grid data structure and simulation data, handling data transition between Python and simulation with minimal memory overhead, and conducting analysis with no additional time penalty using Python C API and NumPy C API. We demonstrate how it solves the problem in astrophysical simulations and increases disk usage efficiency. Finally, we conclude it with discussions about `libyt`.

Index Terms—astronomy data analysis, astronomy data visualization, in situ analysis, open source software

Introduction

In the era of exascale computing, storage and analysis of large-scale data has become a critical bottleneck. Simulations often use efficient programming language like C and C++, while many data analysis tools are written in Python, for example `yt`[1] [1]. `yt` is an open-source, permissively-licensed Python package for analyzing and visualizing volumetric data. It is a light weight tool for quantitative analysis for astrophysical data, and it has also been applied to other scientific domains. Normally, we would have to dump simulation data to hard disk first before conducting analysis using existing Python tools. This takes up lots of disk space when the simulation has high temporal and spatial resolution. This also forces us to store full datasets, even though our region of interest might contain only a small portion of simulation domain. It makes large simulation hard to analyze and manage due to the limitation of disk space. Is there a way to probe those ongoing simulation data using robust Python ecosystem? So that we don't have to re-invent data analysis tools and solve the disk usage issue at the same time.

In situ analysis, which is to analyze simulation data on-site, without intermediate step of writing data to hard disk is a promising solution. It also reduces the barrier of analyzing data by using well-developed tools instead of creating our own. We introduce in situ analysis tool `libyt`[2] , an open source C++ library, that allows researchers to analyze and visualize data by directly calling `yt` or any other Python packages during simulations runtime under parallel computation. Through wrapping ongoing simulation data using NumPy C API [2], constructing proper Python C-extension methods and Python objects using Python C API [3], we can reuse C++ runtime data and realize back-communication of simulation information, allowing user to define their own data generating C function, and use it to conduct analysis inside Python ecosystem. This is like using a normal Python prompt, but with direct access to simulation data. `libyt` provides another way for us to interact with simulations.

In this proceeding, we will describe the methods in Section Code Method, demonstrate how `libyt` solve the problem in Section Applications, and conlude it with Section Discussions.

Code Method

Overview of libyt

`libyt` serves as a bridge between simulation processes and Python instances as illustrated in Fig 1. It is the middle layer that handles data IO between simulations and Python instances, and between MPI processes. When launching *N* MPI processes, each process contains one piece of simulation and one Python interpreter. Each Python interpreter has access to simulation data. When doing in situ analysis, every simulation process pauses, and a total of *N* Python instances will work together to conduct Python tasks in the process space of MPI.

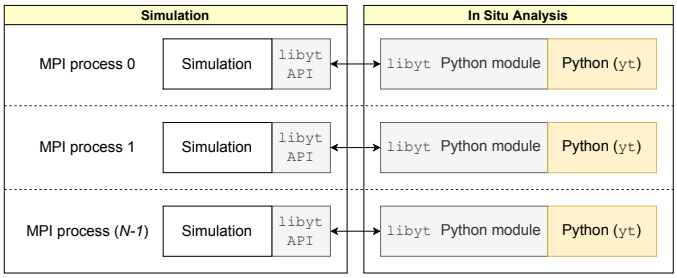

Fig. 1: *This is the overall structure of `libyt`, and its relationship with simulation and Python. It provides an interface for exchanging data between simulations and Python instances, and between each process, thereby enabling in situ parallel analysis using multiple MPI processes. `libyt` can run arbitrary Python scripts and Python modules, though here we focus on using `yt` as its core analysis platform.*

Simulations use `libyt` API[3] to pass in data and run Python codes during runtime, and Python instances use `libyt` Python

∗ *Corresponding author: srtsai@illinois.edu*
‡ *National Taiwan University, Department of Physics*
§ *University of Illinois at Urbana-Champaign, School of Information Sciences*
¶ *National Taiwan University, Institute of Astrophysics*
‖ *National Taiwan University, Center for Theoretical Physics*
∗∗ *National Center for Theoretical Sciences, Physics Division*

1. https://yt-project.org/

2. https://github.com/yt-project/libyt

module to request data directly from simulations using C-extension method and access Python objects that contain simulation information. Using `libyt` for in situ analysis is very similar to running Python scripts in post-processing under MPI platform, except that data are stored in memory instead of hard drives. `libyt` is for general-purpose and can launch arbitrary Python scripts and Python modules, though here, we focus on using yt as our core analysis tool.

Connecting Python and Simulation

We can extend the functionality of Python by calling C/C++ functions, and, likewise, we can also embed Python in a C/C++ application to enhance its capability. Python and NumPy provides C API for users to connect objects in a main C/C++ program to Python.

Currently, `libyt` supports only adaptive mesh refinement (AMR) grid data structure.[4] How `libyt` organizes simulation with AMR grid data structure is illustrated in Fig 2. It first gathers and combines local adaptive mesh refinement grid information (e.g., levels, parent id, grid edges, etc) in each process, so that every Python instance contains full information. Next, it allocates array using `PyArray_SimpleNew` and stores the information in a linear fashion according to global grid id. The array can be easily looked up, and we can retrieve information by `libyt` at C side using `PyArray_GETPTR2`. The operation only involves reading elements in an array. The array is accessible both in C/C++ and Python runtimes. For simulation data, `libyt` wraps those data pointers using NumPy C API `PyArray_SimpleNewFromData`. This tells Python how to interpret block of memory (e.g., shape, type, stride) and does not make a copy. `libyt` also marks the wrapped data as read-only[5] to avoid Python accidentally alters it, since they are actual data used in simulation's iterative process.

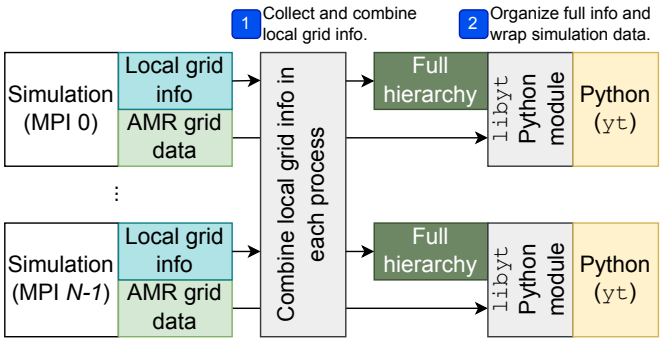

Fig. 2: *This diagram shows how `libyt` loads and organizes simulation information and data that is based on adaptive mesh refinement (AMR) grid data structure. `libyt` collects local AMR grid information and combines them all, so that each Python instance contains whole information. As for simulation data, `libyt` wraps them using NumPy C API, which tells Python how to interpret block of memory without duplicating it.*

`libyt` also supports back-communication of simulation information. Fig 3 shows the mechanism behind it. The process is triggered by Python when it needs the data generated by a user-defined C function. This usually happens when the data is not part of the simulation iterative process and requires simulation to generate it, or the data isn't stored in a contiguous memory block and requires simulation to help collect it. When Python needs the data, it first calls C-extension method in `libyt` Python module. The C-extension method allocates a new data buffer and passes it to user-defined C function, and the function writes data in it. Finally, `libyt` wraps the data buffer and returns it back to Python. `libyt` makes the data buffer owned by Python[6], so that the data gets freed when it is no longer needed.

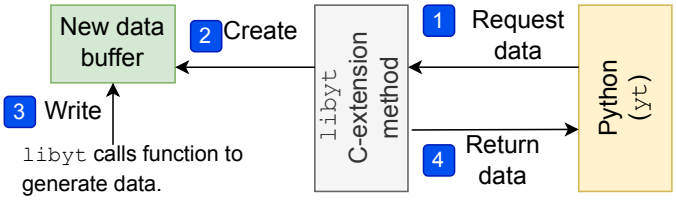

Fig. 3: *This diagram describes how `libyt` requests simulation to generate data using user-defined C function, thus enabling back-communication of simulation information. Those generated data is freed once it is no longer used by Python.*

Grid information and simulation data are properly organized in dictionaries under `libyt` Python module. One can import it during simulation runtime:

```
import libyt  # Import libyt Python module
```

In Situ Analysis Under Parallel Computing

Each MPI process contains one simulation code and one Python instance. Each Python instance only has direct access to the data on local computing nodes, thus all Python instances must work together to make sure everything is in reach. During in situ Python analysis, workloads may be decomposed and rebalanced according to the algorithm in Python packages. It is not necessary to align with how data is distributed in simulation. Even though `libyt` can call arbitrary Python modules, we focus on how it uses `yt` and MPI to do analysis under parallel computation here.

`yt` supports parallelism feature[7] using `mpi4py`[8] as communication method. `libyt` borrows this feature and utilizes it directly. The way `yt` calculates and distributes jobs to each MPI process is based on data locality, but it does not always guarantee to do so[9]. In other words, in in situ analysis, the data requested by `yt` in each MPI process does not always locate in the same process.

Furthermore, there is no way for `libyt` to know what kind of communication pattern a Python script needs in advance. For a much more general case, it is difficult to schedule point-to-point communications that fit any kind of algorithms and any number of MPI processes. `libyt` uses one-sided communication in MPI, also known as Remote Memory Access (RMA), by which one no longer needs to explicitly specify senders and receivers. Fig 4

3. For more details, please refer to `libyt` documents. (https://yt-project.github.io/libyt/libytAPI)

4. We will support more data structures (e.g., octree, unstructured mesh grid, etc) in the future.

5. This can be done by using `PyArray_CLEARFLAGS` to clear writable flag `NPY_ARRAY_WRITEABLE`.

6. This can be done by using `PyArray_ENABLEFLAGS` to enable own-data flag `NPY_ARRAY_OWNDATA`.

7. See Parallel Computation With yt for more details.

8. `mpi4py` is Python bindings for MPI. (https://mpi4py.readthedocs.io/en/stable/)

9. `yt` functionalities like `find_max`, `ProjectionPlot`, `create_profile`, `PhasePlot`, etc are based on data locality, others like `OffAxisProjectionPlot`, `SlicePlot`, `OffAxisSlicePlot`, etc don't.

describes the data redistribution process in `libyt`. `libyt` first collects requested data in each process and askes each process to prepare it. Then `libyt` creates an epoch, for which all MPI processes will enter, and each process can fetch the data located on different processes without explicitly waiting for the remote process to respond. The caveat in data exchanging procedure in `libyt` is that it is a collective operation, and requires every MPI process to participate.

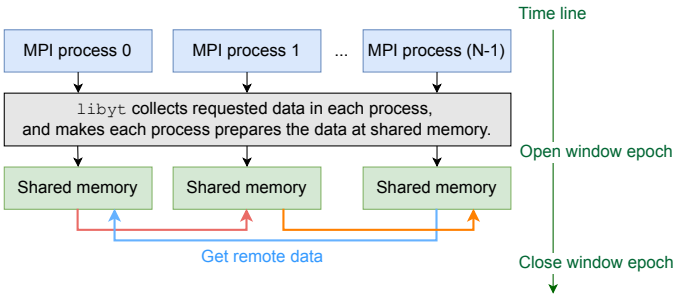

Fig. 4: *This is the workflow of how `libyt` redistributes data. It is done via one-sided communication in MPI. Each process prepares the requested data from other processes, after this, every process fetches data located on different processes. This is a collective operation, and data is redistributed during this window epoch. Since the data fetched from other processes is only for analysis purpose, it gets freed once Python doesn't need it at all.*

Executing Python Codes and Handling Errors

`libyt` imports user's Python script at the initialization stage. Every Python statement is executed inside the imported script's namespace using `PyRun_SimpleString`. The namespace holds Python functions and objects. Every change made will also be stored under this namespace and will be brought to the following round.

Using `libyt` for in situ analysis is just like running Python scripts in post-processing. The only difference lies in how the data is loaded. Post-processing has everything store on hard disk, while data in in situ analysis is distributed in memory space in different computing nodes. Though `libyt` can call arbitrary Python modules, here, we focus on using `yt` as the core method. This is an example of doing slice plot using `yt` function `SlicePlot` in post-processing:

```
import yt
yt.enable_parallelism()
def do_sliceplot(data):
    ds = yt.load(data)
    slc = yt.SlicePlot(ds, "z", ("gamer", "Dens"))
    if yt.is_root():
        slc.save()
if __name__ == "__main__":
    do_sliceplot("Data000000")
```

Converting the post-processing script to inline script is a two-line change. We need to import `yt_libyt`[10], which is the `yt` frontend for `libyt`. And then we change `yt.load` to `yt_libyt.libytDataset()`. That's it! Now data is loaded from `libyt` instead of loading from hard disk. The following is the inline Python script:

```
import yt_libyt
import yt
yt.enable_parallelism()
def do_sliceplot_inline():
    ds = yt_libyt.libytDataset()
    slc = yt.SlicePlot(ds, "z", ("gamer", "Dens"))
    if yt.is_root():
        slc.save()
```

Simulation can call Python function using `libyt` API `yt_run_Function` and `yt_run_FunctionArguments`. For example, this calls the Python function `do_sliceplot_inline`:

```
yt_run_Function("do_sliceplot_inline");
```

Beside calling Python function, `libyt` also provides interactive prompt for user to update Python function, enter statements, and get feedbacks instantly.[11] This is like running Python prompt inside the ongoing simulation with access to data. Fig 5 describes the workflow. The root process takes user inputs and checks the syntax through compiling it to code object using `Py_CompileString`. If error occurs, it parses the error to see if this is caused by input not done yet or a real error. If it is indeed caused by user hasn't done yet, for example, when using an `if` statement, the prompt will continue waiting for user inputs. Otherwise, it simply prints the error to inform the user. If the code can be compiled successfully, the root process broadcasts the code to every other MPI processes. Then they evaluate the code using `PyEval_EvalCode` inside the script's namespace simultaneously.

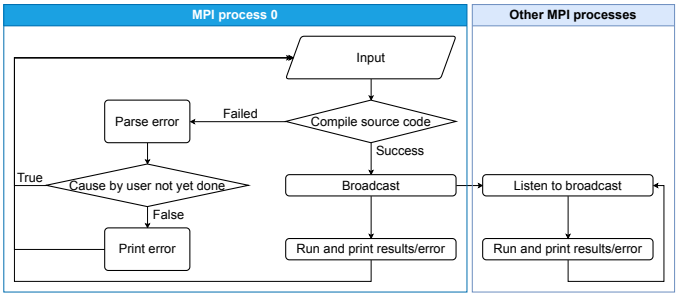

Fig. 5: *The procedure shows how `libyt` supports interactive Python prompt. It takes user inputs on root process and executes Python codes across whole MPI processes. The root process handles syntax errors and distinguishes whether or not the error is caused by user hasn't done inputing yet.*

Applications

`libyt` has already been implemented in `GAMER`[12] [4] and `Enzo`[13] [5]. `GAMER` is a GPU-accelerated adaptive mesh refinement code for astrophysics. It features extremely high performance and parallel scalability and supports a rich set of physics modules. `Enzo` is a community-developed adaptive mesh refinement simulation code, designed for rich, multi-physics hydrodynamic astrophysical calculations.

Here, we demonstrate the results from `GAMER` using `libyt`, and we show how `libyt` solves the problem of limitation in disk space and improves disk usage efficiency.

10. https://github.com/data-exp-lab/yt_libyt

11. Currently, `libyt` interactive prompt only works on local machine or submitting the job to HPC platforms using interactive queue (e.g., `qsub -I` on PBS scheduler). We will support accessing through Jupyter Notebook in the future.

12. https://github.com/gamer-project/gamer

13. https://enzo-project.org/

Analyzing Fuzzy Dark Matter Vortices Simulation

Fuzzy dark matter (FDM) is a promising dark matter candidate [6]. It is best described by a classical scalar field governed by the Schrödinger-Poisson equation, because of the large de Broglie wavelength compared to the mean interparticle separation. FDM halos feature a central compact solitonic core surrounded by fluctuating density granules resulting from wave function interference. Quantum vortices can form in density voids caused by fully destructive interference [7] [8]. The dynamics of these vortices in FDM halo have not been investigated thoroughly, due to the very high spatial and temporal resolution is required, which leads to tremendously huge disk space. `libyt` provides a promising approach for this study.

We use `GAMER` to simulate the evolution of an FDM halo on the Taiwania 3[14]. We use 560 CPU cores by launching 20 MPI processes with 28 OpenMP threads per MPI process to run the simulation. The simulation box size is 2.5×10^5 pc, covered by a 640^3 base-level grid with six refinement levels. The highest level has a maximum resolution of 6.2 pc, so that it is able to resolve the fine structure and dynamical evolution of vortices within a distance of 3200 pc from the center. To properly capture the dynamics, we aim for analyzing vortex properties with a temporal resolution of 3.5×10^{-2} Myr, corresponding to 321 analysis samples. Each simulation snapshot, including density, real part, imaginary part, gravitational potential, and AMR grid information, takes 116 GB. It will take $\sim$ 37 TB if we do this in post-processing, which is really expensive. However, it is actually unnecessary to dump all these snapshots since our region of interest is only the vortex lines around the halo center.

We solve this by using `libyt` to invoke `yt` function `covering_grid` to extract a uniform-resolution grid centered at the halo center and store these grid data instead of simulation snapshots on disk. The uniform grid has dimension 1024^3 with spatial resolution 6.2 pc (i.e., the maximum resolution in the simulation), correspnding to the full extracted uniform grid width of 6300 pc. By storing only the imaginary and real parts of the wave function in single precision, each sample step now consumes only 8 GB, which is 15 times smaller than the snapshot required in post-processing.

We further analyze these uniform grids in post-processing, and do volume rendering and create animation[15] using ParaView [9]. Fig 6 is the volume rendering of the result. Vortex lines and rings are manifest in the entire domain. Fig 7 shows a zoom in version of Fig 6, where reconnection of vortex lines take place. With the help of `libyt`, we are able to achieve a very high temporal resolution and very high spatial resolution at the same time.

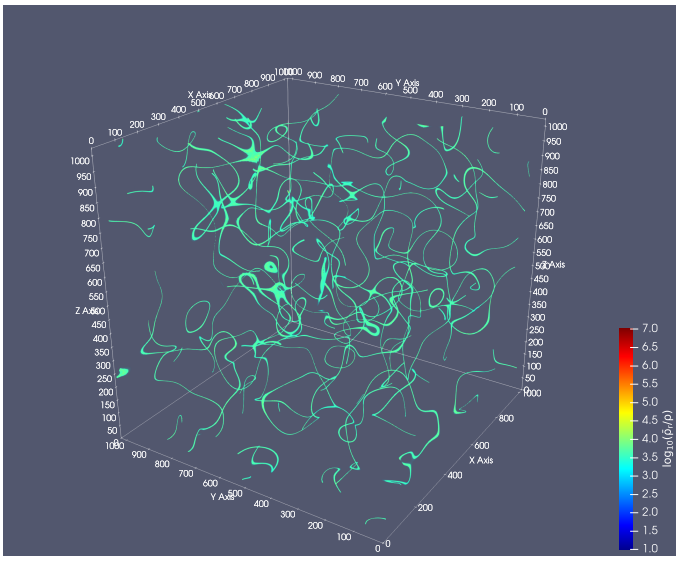

Fig. 6: *Volume rendering of quantum vortices in a fuzzy dark matter halo with `GAMER`. Here we use `libyt` to extract uniform-resolution data from an AMR simulation on-the-fly, and then visualize it with ParaView in post-processing. The colormap is the logarithm of reciprocal of density averaging over radial density profile, which highlights the fluctuations and null density. Tick labels represent cell indices.*

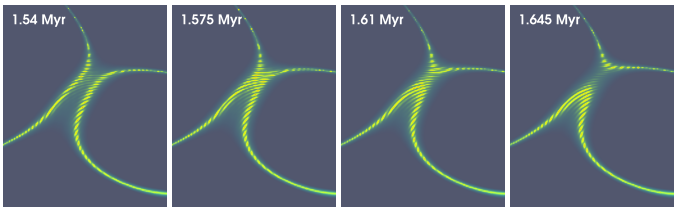

Fig. 7: *Vortex reconnection process in a fuzzy dark matter halo. This is the result we get if we zoom in to one of the vortex lines in Fig 6 where reconnection of lines take place. We are able to clearly capture the dynamics, and at the same time, preserve high spatial resolution.*

Analyzing Core-Collapse Supernova Simulation

We use `GAMER` to simulate core-collapse supernova explosions. The simulations have been performed on a local cluster using 64 CPU cores and 4 GPUs by launching 8 MPI processes with 8 OpenMP threads per MPI process, and having two MPI processes access the same GPU. The simulations involve a rich set of physics modules, including hydrodynamics, self-gravity, a parameterized light-bulb scheme for neutrino heating and cooling with a fixed neutrino luminosity [10], a parameterized deleptonization scheme [11], an effective general relativistic potential [12], and a nuclear equation of state [13]. For the hydrodynamics scheme, we adopt the van Leer predictor-corrector integrator [14] [15], the piecewise parabolic method for spatial data reconstruction [16], and the HLLC Riemann solver [17]. The simulation box size is 2×10^4 km. The base-level grid dimension is 160^3 and there are eight refinement levels, reaching a maximum spatial resolution of ~ 0.5 km.

We use `libyt` to closely monitor the simulation progress during runtime, such as the grid refinement distribution, the status and location of shock wave (e.g., stalling, revival, breakout), and the evolution of the central proto-neutron star. `libyt` calls `yt` function `SlicePlot` to draw entropy distribution every 1.5×10^{-2} ms. Fig 8 is the output in a time step. Since entropy is not part of the variable in simulation's iterative process, these entropy data will only be generated through user-defined C function, which in turn calls the nuclear equation of state defined inside `GAMER` to get entropy, when they are needed by `yt`. `libyt` tries to minimize memory usage by generating relevant data only. We can combine every output figure and animate the actual simulation process[16] without storing any datasets.

14. Supercomputer at the National Center for High-performance Computing in Taiwan. (https://www.nchc.org.tw/)

15. https://youtu.be/tUjJYGbWgUc

16. https://youtu.be/6iwHzN-FsHw

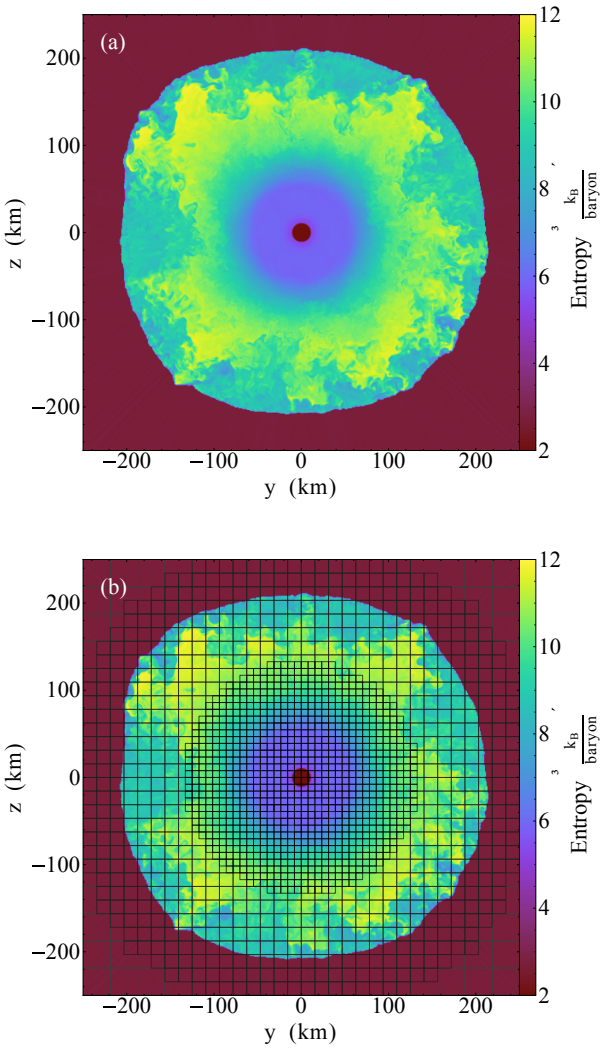

Fig. 8: *Entropy distribution in a core-collapse supernova simulated by* `GAMER` *and plotted by* `yt` *function* `SlicePlot` *using* `libyt`*. Plot (a) shows a thin slice cut through the central proto-neutron in the post-bounce phase. The proto-neutron star has a radius of* ~ 10 *km and the shock stalls at* ~ 200 *km. Plot (b) shows the underlying AMR grid structure, where each grid consists of* 16^3 *cells.*

Discussions

`libyt` is free and open source, which does not depend on any non-free or non-open source software. Converting the post-processing script to inline script is a two-line change, which lowers the barrier of using this in situ analysis tool.

Though currently, only simulations that use AMR grid data structure are supported by `libyt`, we will extend to more data structure (e.g., octree, particle, unstructured mesh, etc) and hope to engage more simulations and data structures in the future.

Using `libyt` does not add time penalty to the analysis process, because using Python for in situ analysis and post-processing are exactly the same, except that the former one reads data from memory and the latter one reads data from disks. Fig 9 shows the strong scaling of `libyt`. The test compares the performance between in situ analysis with `libyt` and post-processing for computing 2D profiles on a `GAMER` dataset. The dataset contains seven adaptive mesh refinement levels with a total of 9.9×10^8 cells. `libyt` outperforms post-processing by $\sim 10 - 30\%$, since it avoids loading data from disk to memory. `libyt` and post-processing have similar deviation from the ideal scaling since `libyt` directly borrows the algorithm in `yt`. Some improvements have been made in `yt`, while some are still undergoing to eliminate the scaling bottleneck. But also, due to some parts cannot be parallelized, like the import of Python and the current data structure, the speed up is saturated at large number of processors and can be described by Amdahl's law.

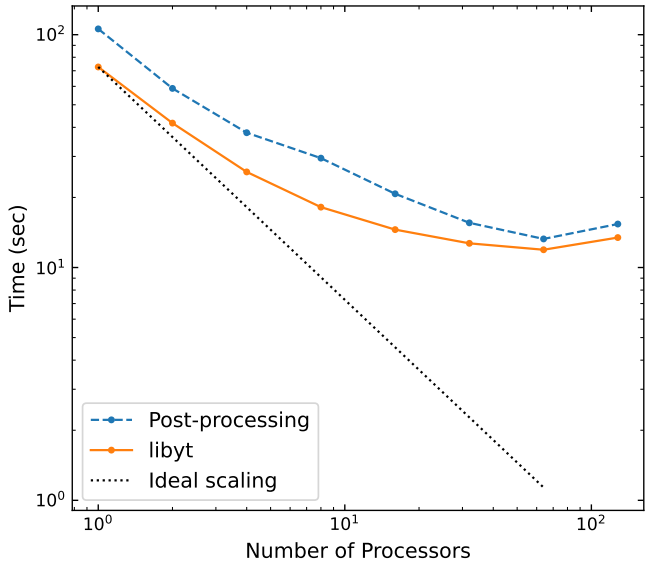

Fig. 9: *Strong scaling of* `libyt`*.* `libyt` *outperforms post-processing by* $\sim 10 - 30\%$ *since the former avoids loading data from disk to memory. The dotted line is the ideal scaling.* `libyt` *and post-processing show a similar deviation from the ideal scaling because it directly borrows the algorithm in* `yt`*. Improvements have been made and will be made in* `yt` *to eliminate the scaling bottleneck.*

`libyt` provides a promising solution that binds simulation to Python with minimal memory overhead and no additional time penalty. It makes analyzing large scale simulation feasible, and it can analyze the data with much higher frequency. It also reduces the barrier of heavy computational jobs written in C/C++ to use Python tools, which are normally well-developed. `libyt` focuses on using `yt` as its core analytic method, even though it can call other Python modules, and has the ability to enable back-communication of simulation information. A use case of this tool could be using `yt` to select data and then make it as an input source for further analysis. `libyt` provides us another way to interact with simulation and data.

References

[1] M. J. Turk, B. D. Smith, J. S. Oishi, S. Skory, S. W. Skillman, T. Abel, and M. L. Norman, "yt: A Multi-code Analysis Toolkit for Astrophysical Simulation Data," *The Astrophysical Journal Supplement Series*, vol. 192, p. 9, Jan. 2011, https://doi.org/10.1088/0067-0049/192/1/9.

[2] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant, "Array programming with NumPy," *Nature*, vol. 585, no. 7825, pp. 357–362, Sep. 2020, https://doi.org/10.1038/s41586-020-2649-2. [Online]. Available: https://doi.org/10.1038/s41586-020-2649-2

[3] G. Van Rossum and F. L. Drake, *Python 3 Reference Manual*. Scotts Valley, CA: CreateSpace, 2009.

[4] H.-Y. Schive, J. A. ZuHone, N. J. Goldbaum, M. J. Turk, M. Gaspari, and C.-Y. Cheng, "gamer-2: a GPU-accelerated adaptive mesh refinement code – accuracy, performance, and scalability," *Monthly Notices of the Royal Astronomical Society*, vol. 481, no. 4, pp. 4815–4840, 09 2018, https://doi.org/10.1093/mnras/sty2586. [Online]. Available: https://doi.org/10.1093/mnras/sty2586

[5] G. L. Bryan, M. L. Norman, B. W. O'Shea, T. Abel, J. H. Wise, M. J. Turk, D. R. Reynolds, D. C. Collins, P. Wang, S. W. Skillman, B. Smith, R. P. Harkness, J. Bordner, J.-h. Kim, M. Kuhlen, H. Xu, N. Goldbaum, C. Hummels, A. G. Kritsuk, E. Tasker, S. Skory, C. M. Simpson, O. Hahn, J. S. Oishi, G. C. So, F. Zhao, R. Cen, Y. Li, and The Enzo Collaboration, "ENZO: An Adaptive Mesh Refinement Code for Astrophysics," *The Astrophysical Journal Supplement Series*, vol. 211, p. 19, Apr. 2014, https://doi.org/10.1088/0067-0049/211/2/19.

[6] H.-Y. Schive, T. Chiueh, and T. Broadhurst, "Cosmic structure as the quantum interference of a coherent dark wave," *Nature Physics*, vol. 10, pp. 496–499, Jul. 2014, https://doi.org/10.1038/nphys2996.

[7] T. Chiueh, "Dynamical quantum chaos as fluid turbulence," *Phys. Rev. E*, vol. 57, pp. 4150–4154, Apr 1998, https://doi.org/10.1103/PhysRevE.57.4150. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevE.57.4150

[8] L. Hui, A. Joyce, M. J. Landry, and X. Li, "Vortices and waves in light dark matter," *Journal of Cosmology and Astroparticle Physics*, vol. 2021, no. 1, p. 011, Jan. 2021, https://doi.org/10.1088/1475-7516/2021/01/011.

[9] J. Ahrens, B. Geveci, and C. Law, *Visualization Handbook*. Burlington, MA: Elsevier Butterworth–Heinemann, 2005, ch. ParaView: An End-User Tool for Large-Data Visualization, p. 717.

[10] S. M. Couch, "The Dependence of the Neutrino Mechanism of Core-collapse Supernovae on the Equation of State," *The Astrophysical Journal*, vol. 765, no. 1, p. 29, Mar. 2013, https://doi.org/10.1088/0004-637X/765/1/29.

[11] M. Liebendörfer, "A Simple Parameterization of the Consequences of Deleptonization for Simulations of Stellar Core Collapse," *The Astrophysical Journal*, vol. 633, no. 2, pp. 1042–1051, Nov. 2005, https://doi.org/10.1086/466517.

[12] E. P. O'Connor and S. M. Couch, "Two-dimensional Core-collapse Supernova Explosions Aided by General Relativity with Multidimensional Neutrino Transport," *The Astrophysical Journal*, vol. 854, no. 1, p. 63, Feb. 2018, https://doi.org/10.3847/1538-4357/aaa893.

[13] A. W. Steiner, M. Hempel, and T. Fischer, "Core-collapse Supernova Equations of State Based on Neutron Star Observations," *The Astrophysical Journal*, vol. 774, no. 1, p. 17, Sep. 2013, https://doi.org/10.1088/0004-637X/774/1/17.

[14] S. A. E. G. Falle, "Self-similar jets," *Monthly Notices of the Royal Astronomical Society*, vol. 250, no. 3, pp. 581–596, 1991, https://doi.org/10.1093/mnras/250.3.581. [Online]. Available: +http://dx.doi.org/10.1093/mnras/250.3.581

[15] B. van Leer, "Upwind and high-resolution methods for compressible flow: From donor cell to residual-distribution schemes," *Communications in Computational Physics*, vol. 1, pp. 192–206, 2006, https://doi.org/10.2514/6.2003-3559.

[16] P. Colella and P. R. Woodward, "The piecewise parabolic method (ppm) for gas-dynamical simulations," *Journal of Computational Physics*, vol. 54, no. 1, pp. 174–201, 1984, https://doi.org/10.1016/0021-9991(84)90143-8. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0021999184901438

[17] E. F. Toro, *Riemann Solvers and Numerical Methods for Fluid Dynamics. A Practical Introduction*, 3rd ed. Berlin: Springer, 2009.

Data Reduction Network

Haoyin Xu[‡§], Haw-minn Lu[‡*], José Unpingco[§]

Abstract—Multidimensional categorical data is widespread but not easily visualized using standard methods. For example, questionnaire (e.g. survey) data generally consists of questions with categorical responses (e.g., yes/no, hate/dislike/neutral/like/love). Thus, a questionnaire with 10 questions, each with five mutually exclusive responses, gives a dataset of 5^{10} possible observations, an amount of data that would be hard to reasonably collect. Hence, this type of dataset is necessarily sparse. Popular methods of handling categorical data include one-hot encoding (which exacerbates the dimensionality problem) and enumeration, which applies an unwarranted and potentially misleading notional order to the data. To address this, we introduce a novel visualization method named Data Reduction Network (DRN). Using a network-graph structure, the DRN denotes each categorical feature as a node with interrelationships between nodes denoted by weighted edges. The graph is statistically reduced to reveal the strongest or weakest path-wise relationships between features and to reduce visual clutter. A key advantage is that it does not "lose" features, but rather represents interrelationships across the entire categorical feature set without eliminating weaker relationships or features. Indeed, the graph representation can be inverted so that instead of visualizing the strongest interrelationships, the weakest can be surfaced. The DRN is a powerful visualization tool for multidimensional categorical data and in particular data derived from surveys and questionaires.

Index Terms—Data Visualization, Multidimensional Categorical Data

Introduction

The proliferation of Big Data has opened new frontiers in the analysis of information across numerous disciplines. This surge of data is most pronounced in areas such as large-scale surveys, which provide a wealth of multidimensional data capable of answering diverse research questions that may not be easily tested in experimental lab settings [1]. However, the size and complexity of this type of data introduce unique challenges, particularly in the domain of data visualization and downstream analysis.

For example, questionnaire (e.g., survey) data are generally categorical with just a few alternatives for each question, as with the Likert scale that typically consists of five alternatives for each question (e.g., `strongly agree`, `agree`, `neutral`, `disagree`, `strongly disagree`). Consider a small survey with ten questions, each with five alternatives, which produces $5^{10}(9,765,625)$ possible distinct completed questionnaires. In practice, visualizing such a large amount of data in an elegant format is rarely feasible. Rather, the standard approach is to consider a few questions and marginalize (i.e., sum) over the remaining responses. A summary headline result could be that "90% of respondents strongly agree with Question 1". Another could be "75% of respondents strongly agree with Question 2" and so on. However, this approach does not provide information on the respondents who selected `strongly agree` for both Questions 1 and 2. Even more troubling, there could be a third question that is strongly related to both questions but has been marginalized over, potentially leading to issues like Simpson's paradox[2], a phenomenon where a trend or relationship that appears within different groups reverses or disappears when the groups are combined, thus misleading the overall analysis.

As with our survey example above, categorical data often exists in a high-dimensional space. The challenge lies in accurately representing multiple dimensions on a two or three-dimensional plane, which carries potential risks, from oversimplification to cognitive overload [3]. This complexity is amplified by the heterogeneous nature of survey data, which may comprise multiple choice and open-ended responses to Likert scales, each necessitating different visualization techniques [4]. Current common methods to generate informative visualizations for high dimensional categorical data include manually pre-selecting questions or dimensional reduction techniques such as principal component analysis (PCA) and t-distributed stochastic neighbor embedding [5]. These techniques may help to inform some conclusions, yet they may still not be fully comprehensible to non-expert audiences or require excessive human effort to filter the information. In the filtering process, some important relationships, such as joint respondents between two questions, may be omitted due to data oversimplification. Furthermore, such oversimplification may even lead to bias in the downstream analysis.

One popular method for addressing high-dimensional categorical data is through one-hot encoding. In this method, each level of the data element is mapped into a string of n-bits where 1 marks the specific response for that row. For example, using the Likert scale, one-hot encoding produces a string of five bits where `10000` corresponds to `strongly agree` for that particular response and `01000` for the `agree` response. Suppose you have a data frame with rows corresponding to survey respondents and columns corresponding to questions. For each question, one-hot encoding will generate five bits. As a result, for n original columns, there will now be $5n$ one-hot encoded columns. Although it is relatively simple to see this two-dimensional data as a binary array, the relationships between the respondents (rows) and the survey questions (columns) may not be apparent from this representation.

Another common technique is label-encoding where each categorical level is mapped to an integer. For example, if our category is `favorite fruit` with levels `apple`, `banana`, and `strawberry`, then with label encoding, we have `apple`

‡ *Gary and Mary West Health Institute*
§ *University of California, San Diego*
* *Corresponding author: hlu@westhealth.org*

Unit	Question 1	Question 2	...	Question N
1	yes	sometimes	...	responseN
2	no	never	...	responseK
3	yes	never	...	responseN
...	...	...	...	...

TABLE 1: *Sample Survey Data Table. Each row represents a participant's response record and each column corresponds to questions in the survey.*

Unit	Question 1	Question 2	...	Question N
1	1	3	...	6
2	2	4	...	2
3	1	4	...	6
...	...	...	...	...

TABLE 2: *Survey data table where each response is converted to an integer number.*

$\rightarrow 1$, `banana` $\rightarrow 2$, and `strawberry` $\rightarrow 3$. The problem with this approach lies in the inherent ordering of integers, which may not reflect the true nature of the categories. For example, because $2 > 1$, does that imply that `banana` is somehow more than `apple`? The downstream numerical analysis relies on numeric properties and is oblivious to the nuances expressed by the categorical variable. This lack of sensitivity can lead to spurious correlations or nonsensical results.

To distill insightful and actionable visualizations from large survey data, it is essential to balance the trade-off between simplicity and completeness, highlighting the most critical variables for downstream data processing [6]. One previous attempt involves using the cobweb diagram to represent the inter-relationships between nodes [7]. This method effectively distills the complexity in high dimensional contingency tables, but may not be easily comprehensible to those without expertise. In this paper, we propose the Data Reduction Network (DRN) method, a straightforward visualization for representing multidimensional categorical data. The DRN method generates a condensed network graph, emphasizing the strong interrelationships among the variables. By employing a maximum spanning tree, the network not only avoids the risk of oversimplification but also retains the most significant insights from the survey.

Method

The DRN is a graph composed of nodes and edges. Each node denotes the answer to a particular survey question, whereas the

Unit	Question 1	Question 2	...	Question N
1	Q1-1	Q2-3	...	QN-6
2	Q1-2	Q2-4	...	QN-2
3	Q1-1	Q2-4	...	QN-6
...	...	...	...	...

TABLE 3: *Survey table with labeled response and corresponding questions.*

Unit	Question 1	Question 2	Question 3
1	Q1-1	Q2-3	Q3-6
2	Q1-1	Q2-4	Q3-6
3	Q1-2	Q2-4	Q3-2

TABLE 4: *Survey table with further transformed responses label. Now each label will have the question number in them.*

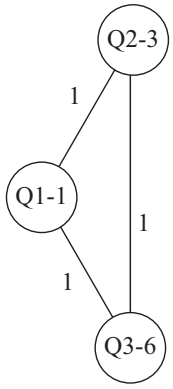

Fig. 1: *A clique representation of the first respondent in the hypothetical example.*

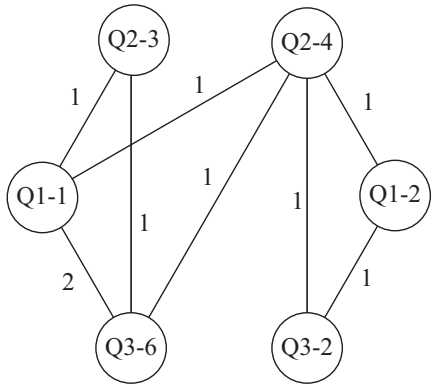

Fig. 2: *Network illustrating responses from the first 3 rows of the response table.*

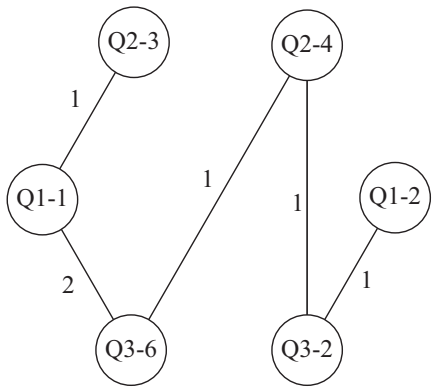

Fig. 3: *Network after maximum spanning tree.*

edges between nodes indicate the co-occurrence of respective answers. The DRN maintains a running tally of the number of times an edge occurs in the dataset, as well as the frequency of a particular response to a given question.

Suppose we have a set of survey data shown in Table 1. For the sake of convenience and visual simplicity, we map each response to an integer (e.g., yes → 1, no → 2, sometimes → 3) as demonstrated in Table 2. It is worth noting that while this is technically a label encoding process, it is an optional process. The node `Q1_1` could have been labeled `Q1_yes` but it is clear that visual clutter would soon ensue for more complex responses like `Q3_responseK`. By employing this encoding technique, we achieve a concise representation of each question-response pair, as demonstrated in Table 3. This gives every entry in the table a code that indicates the question-response pair. Each row in the table corresponds to a respondent. It is possible for two respondents to answer the survey in exactly the same way, leading to duplicated rows in the table.

Once the data is properly encoded, the subsequent step is to construct the DRN where each entry from the table represents a node, and the edges symbolize the connections between these nodes. This means that each row generates a *clique*, a network of mutually connected nodes, and these cliques are merged to create a comprehensive clique whose edges count all joint responses for the entire table.

The process of constructing a master graph from the table is a two step process:

(i) Form a clique for every row
(ii) Merge the cliques by accumulating the edge counts

Following a hypothetical example with three questions shown in Table 4, we start with the first row, each question-response term becomes a node and there is an edge between every two terms in this row with weight 1. The clique formed from the first row is depicted in Figure 1. After completing cliques for all rows, three in this case, they are merged into a master graph where the edge counts are accumulated as shown in Figure 2. Please note that in this case, the edge between `Q1_1` and `Q3_6` occurs in two of the cliques so it is weighted 2.

For large datasets, these master graphs can become extremely dense and impossible to interpret. To simplify the graph and reveal the key statistically significant relationships, we employ `networkx`'s `maximum_spanning_tree` function, which applies Prim's algorithm [8]. This algorithm calculates the maximum spanning tree where the weights are the edge counts. It's worth mentioning that Prim's algorithm is a greedy algorithm that finds a maximum spanning tree for a weighted undirected graph. Furthermore, it's worth noting that any maximum spanning tree algorithm could be used. For instance, Kruskal's algorithm [9], also supported by `networkx`[10], is a valid alternative.

The resulting tree maintains the largest edge-weights, which are the strongest co-occurring responses. It is worth noting that the maximum spanning tree might not be unique; hence, in the event of a tie, any competing tree could be chosen. With large combinatorial complexity associated with large data sets such ties are unlikely. In the present example, one maximum-spanning tree is shown in Figure 3

Given that the edges in the DRN represent the tallies of co-occurring survey responses, these values align with those found in a contingency table that encompasses the variables represented by the nodes. Therefore, the DRN effectively *unrolls* a high-dimensional contingency table onto a network graph, thereby facilitating the identification of the strongest inter-relationships. DRNs facilitate inferential statistics by translating topological features, such as clusters and connections in the graph, into statistical quantities of interest.

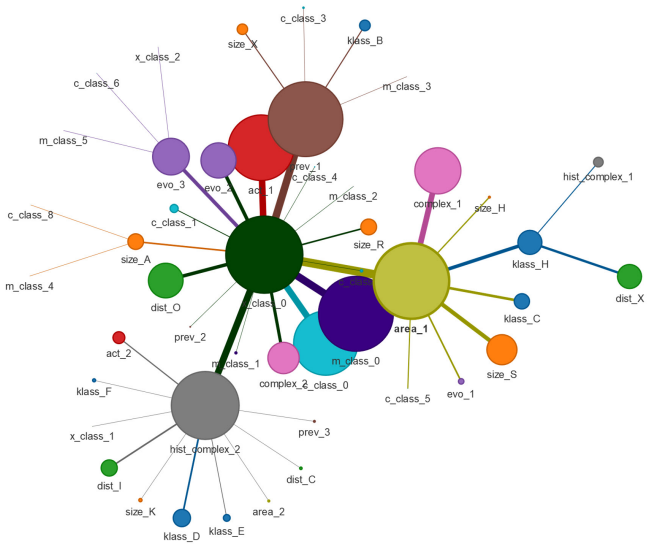

Fig. 4: *An overview of DRN for the solar flare dataset*

Example

The Solar Flare Dataset

As an example, a simple dataset is presented. The Solar Flare Dataset was adopted from University of California at Irvine (UCI) Machine Learning Repository[1]. Each entry of the dataset describes multiple features of an active region on the sun as well as the number of flares events that occurred within the past 24 hours in the region. The features were described in Table 5. We've also provided this in an example Jupyter notebook, the link to which can be found in the conclusion section of this document.

Figure 4 shows the DRN representation of the Solar Flare Dataset. In addition to the methods mentioned in the previous section, for clarity edge weights are not explicitly shown but translated into edge thickness with the most common occurrence of responses having heavier lines. Additionally, the size of the nodes is proportional to the number of responses for the particular question-response. Finally for added clarity, the responses associated with each question are encoded with the same color making it easy to discern which responses belong to which question. Each feature is encoded as a single label. For example, for the feature `act`, there are two options: 1 (reduced), and 2 (unchanged). If a region has reduced solar activity, the feature will be encoded as `act_1`. This code will appear in the DRN as nodes and the size of the node is proportional to the number of entries that contains the corresponding feature coding. In other words, greater node size indicates the most prevalent features across all entries.

In Figure 4, the largest node in the center was `x_class_0`, which indicates that for most active regions on the sun, 0 x-class

1. http://archive.ics.uci.edu/ml/datasets/solar+flare

Feature	Description
klass	Code for class (modified Zurich class) (A, B, C, D, E, F, H)
size	Code for largest spot size (X, R, S, A, H, K)
dist	Code for spot distribution (X, O, I, C)
act	Activity (1 = reduced, 2 = unchanged)
evo	Evolution (1 = decay, 2 = no growth, 3 = growth)
prev	Previous 24 hour flare activity code (1 = nothing as big as an M1,2 = one M1,3 = more activity than one M1)
complex	Historically-complex (1 = Yes, 2 = No)
hist_complex	Did region become historically complex on this pass across the sun's disk (1 = yes, 2 = no)
area	Area (1 = small, 2 = large)
c_class	Small with few noticeable consequences on Earth
m_class	Medium-sized; cause brief radio blackouts that affect Earth's polar regions
x_class	Big; major events that can trigger planet-wide radio blackouts and long-lasting radiation storms

TABLE 5: *Table for the features described in the solar flare dataset.*

solar flare activity was observed. While it shows the specific feature encoding for that feature category, the DRN could also reveal joint encoding relationships through the edge connection and the thickness of edges. Thicker edges indicate stronger correlations between two features. For example, regions that have the feature label of `x_class_0` will also most likely have the features of `area_1`, `act_1`, `m_class_0`, `prev_1`, `hist_complex_2`, `c_class_0`, `evo_3`, as indicated by the thickness of the edge. This indicates that regions that do not have x-class solar flare activity, they would most likely be 1) small area, or 2) regions with reduced activity, or 3) no m-class solar flare activity, or 4) not as big as M1, or 5) wasn't considered as historically complex, or 6) no c-class solar flare activity, or 7) actively growing. Notice that each edge represents an independent joint relationship with each other.

Use Case

Telehealth Questionnaire

In this section, we demonstrate an application of the DRN, using it to analyze a complex survey dataset. This dataset was obtained from a survey of the perceptions and uses of telehealth in the care of older adults [6]. The survey used a questionnaire with 29 close-ended questions, spanning multiple-choice, agreement scale, dichotomous true/false, and rating scale formats. These statements presented potential challenges associated with using telehealth to serve older adults, touching on areas like relationship building, high medical complexity, physical and cognitive impairment, and fragmentation of care. The survey captured responses from 7246 U.S. clinicians across a range of specialties.[2]

Figure 5 depicts an overview of the DRN for the entire survey. We've followed a specific convention for labeling each node. When questions include subquestions, the corresponding response is represented with a hyphenated label connecting the question number, subquestion number, and numeric representation of the response. For instance, a **TRUE** response for question 11 subquestion 4 is labeled as `Q11-4_90`. The numeric representation of the response can be arbitrary but to avoid confusion the coding used in the survey results itself was used where **TRUE** is represented as 90. For questions without subquestions, we simply omit the hyphen and subquestion number, e.g., `Q2_91`.

2. The exact questionnaire can be found at: https://pubmed.ncbi.nlm.nih.gov/36493377/

Each node's size is proportionate to the number of respondents who chose the associated answer, with larger nodes representing the most common question-answer combinations (topline results). For ease of discussion, the most prominent nodes in Figure 5 are labeled with uppercase letters, and two other nodes of interest with an uppercase-lowercase pair. Table 6 complements the figure by detailing the annotation, DRN label, and corresponding question-answer pairs for these nodes.

The prominent nodes, labeled as A, B, Ba, Bb, C, and D in the network diagram, are derived from the subquestions of survey Question 11. This question presented a list of reasons why providers might choose to exclude older adults from their telehealth services, and asked respondents to rate their agreement with each statement, based on their personal and professional opinions. The full text of Question 11 is as follows:

> Q11: The following is the list of reasons why some providers might choose to exclude older adults from their telehealth offerings (visits, program, system). Please read and indicate your personal and professional opinion about whether you believe each statement is true or false

The prominent nodes indicate the most frequently selected responses across all survey participants. Observations from these responses allow for immediate conclusions to be drawn across all U.S. clinical specialties. The most common reasons to exclude telehealth offerings for older patients include: 1) the belief that telehealth couldn't provide sufficient healthcare service to older patients, 2) concerns about whether older patients have enough resources to properly utilize the service, and 3) possible insufficiency of resources by the medical staff to offer the telehealth service. These conclusions serve as the top-line information extracted from the questionnaires.

Similar to the DRN analysis of the solar flare dataset, the strength of joint responses among all respondents is visually emphasized by the width of the lines (i.e., edges) connecting the nodes. Employing the maximum spanning tree method, the DRN selectively showcases the most significant joint relationships within the network. To illustrate, a thicker edge connecting nodes A and B indicates that respondents who selected response B also likely selected response A. In terms of raw data, the count of respondents who selected both responses A and B is notably higher than any other pairwise combination. In sum, the DRN not only indicates topline responses but also captures significant joint responses.

Label	Annotation	Response description	Respondent's Response
Q11-2	A	People over a certain age cannot be well cared for using telehealth	**TRUE**
Q11-3	B	The older adults I serve do not always have access to the resources needed to make a telehealth visit effective	**TRUE**
Q11-5	C	I have concerns about the impact of telehealth on fragmentation of care for older adults	**TRUE**
Q11-6	D	Relationship building via telehealth is more difficult than in person	**TRUE**
Q11-7	Ba	There is a lack of support from my health system leadership or support staff to make telehealth for my older adults an effective alternative	**TRUE**
Q11-8	Bb	Providing telehealth is dangerous to older adults because their care needs are so medically complex	**TRUE**

TABLE 6: *Table for response description and annotation used in figure 5.*

Label	Annotation	Response description	Respondent's Response
Q11-4	Da	(Subquestion 4) Older adults' significant physical or cognitive challenges make telehealth unrealistic	**TRUE**
Q2	Db	For demographic purposes only, select all that describe your ethnic background?	**White/Caucasian**
Q9	Dc	Which of the following best describes the top THREE reasons that you feel some older adults may not use telehealth? Please choose THREE:	**Older adults' physical and/or cognitive challenges**

TABLE 7: *Response description for DRN in figure 6*

The tree-like structure of the DRN further allows the isolation of individual trees within the network, thus providing supplementary visual information for a particular response. For example, Figure 6 shows a detailed view of the tree segment centered around node D for closer examination. For clarity, the most prominent nodes within this tree are annotated according to Table 7.

This tree allows us to see the interconnections between various survey responses. Notably, respondents who agreed that "relationship building via telehealth is more difficult than in person" (D), also agreed with the statements "older adults' significant physical or cognitive challenges make telehealth unrealistic" (Da) and "older adults' physical and/or cognitive challenges is one of the reasons why older patients may refuse to use telehealth" (Dc).

In addition, most respondents who agreed with the difficulties of relationship building via telehealth identified as White/Caucasian (Db). However, this trend might be reflective of the ethnic imbalance in our respondents.

Additionally, upon closer inspection of the Da node, we see that most respondents who expressed concerns about cognitive/physical decline disagreed with the statement "insufficient resources to effectively use telehealth service" (denoted by the light green node Q11-3_26 under Da).

A quick glance at the network offers substantial insights. It becomes clear that the majority of respondents in this survey shared opinions on the potential reasons why providers might exclude older adults from telehealth services (Q11). They largely agreed on three primary factors that make telehealth less suitable for people over a certain age: 1) difficulties in relationship building, 2) potential fragmentation of care, and 3) a lack of resources. Those who indicated a lack of resources as a potential issue further specified that there is a lack of support from the health system leadership or support staff to make telehealth effective. For respondents who indicated the difficulties of relationship building would be a potential issue, the majority of them further specify that older adults' significant physical or cognitive challenges would make telehealth unrealistic.

The majority of the participants expressing concerns about elder physical or cognitive challenges disagreed with the insufficient resource argument. It is also noteworthy that most participants who identified difficulties in relationship building as a major concern reported their ethnic background as white/Caucasian.

This example demonstrates how the DRN offers a clean, concise visualization that enables users to quickly identify relationships between different questions in a survey. It not only highlights the topline information but also allows users to explore additional relevant data associated with each topline piece of information through the examination of the edges. Compared to traditional methods, DRN provides a simpler, yet more informative, visualization of survey data.

Discussion

Data Reduction Networks (DRNs) provide a valuable solution in the field of data exploration tools, predominantly addressing the handling of categorical data, which has often been overlooked by tools focusing on continuous data. As previously noted, the DRN is adept at rapidly highlighting top-line information. It effectively highlights significant relationships, which can be challenging to discern when dealing with high-dimensional contingency tables. While contingency tables are useful for data analysis, they can become intractable beyond a few dimensions and may be too cumbersome for data exploration.

Despite their utility, DRNs share the same limitations common to data exploration tools. Specifically, a DRN's purpose isn't to act as a standalone data analysis tool, even though it can highlight interesting aspects of a data set. Instead, its strength lies in fostering hypotheses that prompt further investigation, either within the existing data set or via the acquisition of new, targeted data.

The utility of DRNs is not confined to survey data. In fact, categorical data can be seen as analogous to survey data, where features map to questions, and feature values correspond to answers. The example of solar flares showcases this equivalence.

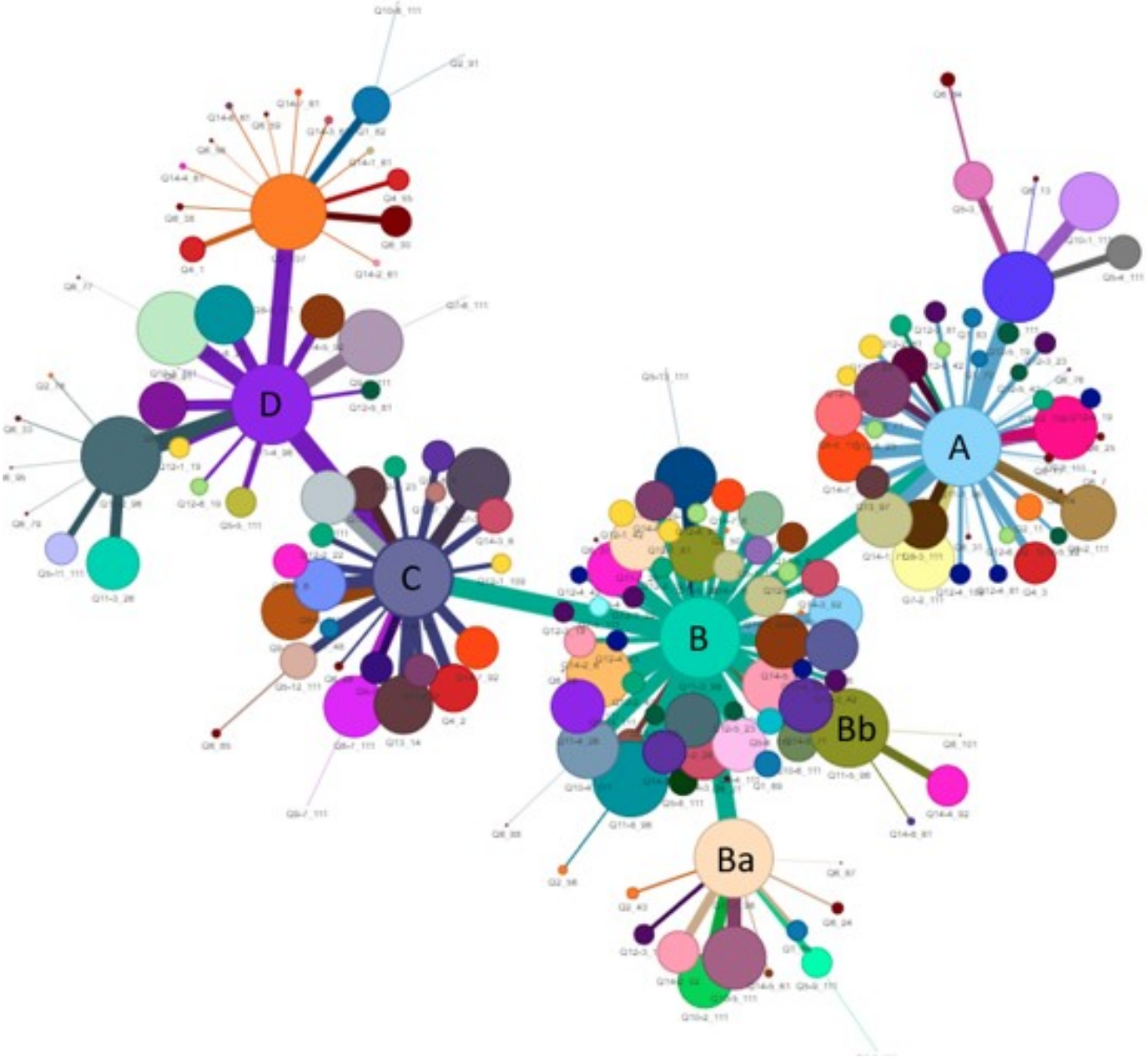

Fig. 5: *An overview of DRN for the telehealth questionnaire*

Although largely a semantic shift, it reframes the data's context. By contrast, our telehealth use case involves the use of actual survey data.

In the presence of continuous data in data sets, discretization or binning techniques akin to those utilized in decision trees can be employed to convert this data into a categorical form. However, this conversion introduces a new challenge - bin formulation. Still, this can be tackled by using existing techniques from decision tree methodologies[11].

A unique problem with survey data comes from questions where multiple responses are permitted. Even visualization methods that can accommodate categorical data struggle with this. The DRN elegantly handles this by generating cliques based on all responses from a respondent, irrespective of whether those responses are associated with different or the same question.

Regarding data imbalances, especially those of a demographic nature, traditional methods such as oversampling underrepresented populations may inadvertently amplify certain relationships in the DRN. As an alternative, removing the overrepresented node and its edges from the main graph, creating a new tree devoid of this node. This action can result in a graphical representation where the influence of the overrepresented node is neutralized, but relationships that may involve that overrepresented demographic are retained. This approach preserves the same weighting of the relationships without emphasizing the relationship to the over-represented node. It falls on the analyst to weigh the merits of this approach versus the traditional oversampling techniques to determine which approach best fits their problem.

The extraction of a maximum spanning tree from the master graph within a DRN serves to capture prominent features in the survey data. However, this method may inadvertently overlook relationships that, while strong, are not the most dominant, and

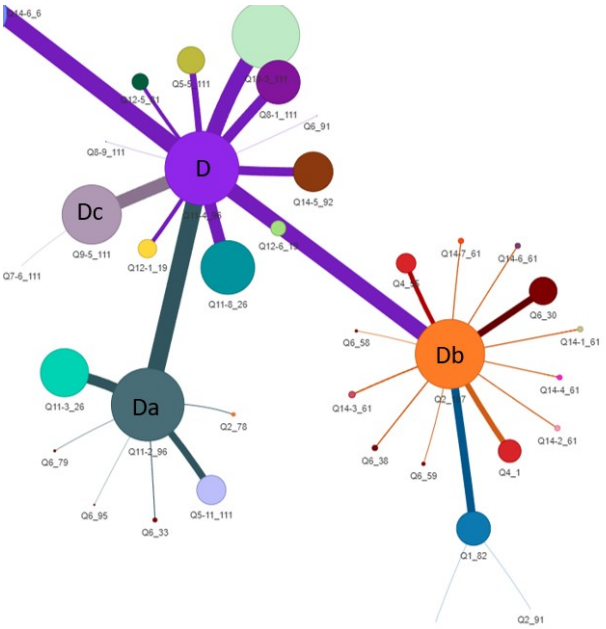

Fig. 6: *A closer look at tree D from the telehealth questionnaire*

it would miss subtler, hidden insights. As a potential future direction, the application of alternative subgraph extraction or graph partitioning methods to the master graph might address this concern.

For further enhancement of its utility and streamlining of data exploration, several features could be incorporated into a DRN's graphical interface. An advanced user interface could enable the omission or disabling of nodes upon a user's request, and subsequently generate new trees accordingly. This would enable users to remove nodes based on their relevance or preference. Another feature could enable a user to select a node, which would then prompt the UI to trigger a statistical function - for instance, constructing a contingency table based on the question associated with the selected node, and questions associated with all connected nodes. This feature could facilitate and streamline subsequent statistical analyses.

Considering the computational demands associated with processing large survey data, strategies such as parallel processing or employing graphics processing unit (GPU)-enhanced graph packages, like cugraph[12], could significantly enhance processing efficiency.

DRNs offer a powerful tool for exploring categorical data, especially in survey data sets. While they have certain limitations, they excel at revealing important relationships and fostering hypotheses for further investigation. The adaptability of DRNs to various data types, including the treatment of continuous data and handling of multiple responses, contributes to their effectiveness. Future improvements to DRN's graphical interface could further enhance their utility, while alternative methods for extracting subgraphs could reveal subtler insights that might be missed by the current approach. Further studies are needed to explore these areas and continue refining the DRN methodology.

DRNs serve as potent tools for categorical data exploration, excelling at revealing key relationships and fostering investigatory hypotheses. Their versatility to handle various data types and responses boosts their efficacy. Future enhancements to the DRN methodology could offer deeper insights, but further research and development is needed.

Conclusion

DRNs offer significant potential in the exploration of categorical data, as they can reveal crucial relationships and foster the generation of hypotheses for subsequent investigations. Their utility in data analysis is promising due to their flexible approach in accommodating diverse data types and responses. They are particularly effective in survey data analysis, where their unique ability to manage categorical data, including continuous data and multiple responses, is most apparent.

However, DRNs are not without their challenges. There is potential for future improvements to enhance their functionality and unlock more profound insights. Potential areas for improvement include the provision of a fully featured graphical interface, the development of alternative methods for subgraph extraction, and more sophisticated management of data imbalances and computational demands. Furthermore, more studies are needed to fully explore and refine the DRN methodology, ensuring that this powerful tool continues to evolve and contribute to the field of data exploration.

An example Jupyter notebook is available at our GitHub repository: https://github.com/Westhealth/drn-scipy2023/. The efforts to build the method to a library is documented in the `README` file.

References

[1] J. Manyika, "Big data: The next frontier for innovation, competition, and productivity," 2011.

[2] M. A. Hernán *et al.*, "The simpson's paradox unraveled," *International Journal of Epidemiology*, vol. 40, no. 3, pp. 780–785, 2011, https://doi.org/10.1093/ije/dyr041.

[3] U. Fayyad, G. G. Grinstein, and A. Wierse, *Information Visualization in Data Mining and Knowledge Discovery*. Elsevier, 2002.

[4] B. Shneiderman, "The eyes have it: a task by data type taxonomy for information visualizations," in *Proceedings 1996 IEEE Symposium on Visual Languages*, 1996, pp. 336–343, https://doi.org/10.1109/VL.1996.545307.

[5] P. C. Wong and R. D. Bergeron, "Multivariate visualization using metric scaling," in *Proceedings. Visualization '97 (Cat. No. 97CB36155)*, 1997, pp. 111–118, https://doi.org/10.1109/VISUAL.1997.663866.

[6] L. Wardlow, C. Roberts, L. Archbald-Pannone, C. for Telehealth, and Aging, "Perceptions and uses of telehealth in the care of older adults," *Telemedicine journal and e-health : the official journal of the American Telemedicine Association*, 2022, advance online publication. [Online]. Available: https://doi.org/10.1089/tmj.2022.0378

[7] G. J. G. Upton, "Cobweb diagrams for multiway contingency tables," *Journal of the Royal Statistical Society Series D: The Statistician*, vol. 49, no. 1, pp. 79–85, 2000, https://doi.org/10.1111/1467-9884.00221.

[8] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, no. 1, pp. 269–271, 1959, https://doi.org/10.1007/BF01386390.

[9] J. B. Kruskal, "On the shortest spanning subtree of a graph and the traveling salesman problem," *Proceedings of the American Mathematical society*, vol. 7, no. 1, pp. 48–50, 1956.

[10] A. A. Hagberg, D. A. Schult, and P. J. Swart, "Exploring network structure, dynamics, and function using networkx," in *Proceedings of the 7th Python in Science Conference*, G. Varoquaux, T. Vaught, and J. Millman, Eds., Pasadena, CA USA, 2008, pp. 11 – 15.

[11] S. García, J. Luengo, J. A. Sáez, V. López, and F. Herrera, "A survey of discretization techniques: Taxonomy and empirical analysis in supervised learning," *IEEE Transactions on Knowledge and Data Engineering*, vol. 25, no. 4, pp. 734–750, 2013, https://doi.org/10.1109/TKDE.2012.35.

[12] R. D. Team, "cuGraph: Gpu-accelerated graph analytics library," https://github.com/rapidsai/cugraph, 2023, accessed: May 23, 2023.