# Proceedings of the 23rd Python in Science Conference

## Proceedings of the 23rd Python in Science Conference

Edited by Meghann Agarwal, Amey Ambade, Chris Calloway, Rowan Cockett, Sanhita Joshi, Charles Lindsey, and Hongsup Shin

SciPy 2024
Tacoma, Washington
July 8 - July 14, 2024

ISSN:2575-9752
https://doi.org/10.25080/DTVR3553

# Organization

## Conference Chairs

- Alexandre Chabot-Leclerc, *Enthought, Inc.*
- Julie Krugler Hollek, *Mozilla*

## Program Chairs

- Paul Ivanov, *Citadel*
- Madicken Munk, *University of Illinois at Urbana-Champaign*
- Guen Prawiroatmodjo, *MotherDuck*
- Matthew Feickert, *University of Wisconsin-Madison*
- Anna Haensch, *Tufts University*

## Communications

- Matt Davis, *Open Source Maintainer*
- Juanita Gomez, *University of California, Santa Cruz*
- Cam Gerlach, *Python Core/Spyder*

## Birds of a Feather

- Michael Akerman, *Novonesis*
- Mike Droettboom, *Microsoft Corporation*

## Proceedings

- Meghann Agarwal, *GDI*
- Amey Ambade, *SLB*
- Chris Calloway, *University of North Carolina*
- Sanhita Joshi, *Deloitte*
- Charles Lindsey, *Aptos*
- Hongsup Shin, *Arm*
- Rowan Cockett, *Curvenote*

## Financial Aid

- Scott Collis, *Argonne National Laboratory*
- Eric Ma, *Moderna*
- Nadia Tahiri, *University of Sherbrooke*

## Tutorials

- Tetsuo Koyama, *PyVista Community*
- Logan Thomas, *Pattern Bioscience*
- Benoit Hamelin, *Tutte Institute for Mathematics and Computing*
- Inessa Pawson, *Albus Code/OpenTeams*

## Sprints

- James Lamb, *NVIDIA*
- Brigitta Sipőcz, *Caltech/IPAC*

## Diversity

- Sarah Kaiser, *Microsoft*
- Meekail Zain, *Quantsight*

## Activities

- Paul Anzel, *Rev.com*
- Ed Rogers, *Majesco*
- Ana Comesana, *Lawrence Berkeley National Laboratory*

## Sponsors/Financial/Logistics

- Jim Weiss, *NumFOCUS*

## Hybrid

- David Nicholson, *Independent Researcher*
- Rebecca BurWei, *Mozilla*
- Allen Harvey Jr, *Applied Research Associates, Inc.*
- Neelima Pulagam, *Ford Motor Company*

## Proceedings Reviewers

- Abhay Dutt Paroha
- Amadi Gabriel Udu
- Andrei Paleyes
- Andrew James
- Andy Terrel
- Angus Hollands
- Ankur Ankan
- Ashwin Hegde
- Blaine Mooers
- Bobby Jackson
- Chong Shen Ng
- Chuchu Wu
- Cliff Kerr
- Conrad Koziol
- Drew Camron
- Dr. Kirtan Dave
- Franklin Koch
- James Lamb
- Jane Adams
- Jennifer E. Yoon
- John Drake
- Juan Cabanela
- Katelyn FitzGerald
- Katie Wetstone
- Kalyan Prasad
- Kevin Lacaille
- Kuntao Zhao
- Lindsey Heagy
- Marcus Hill
- Matt Craig
- Matthew Feickert
- Mihai Maruseac
- Mike Sarahan

- Nadia Tahiri
- Nate Jacobs
- Nicole Brewer
- Paul Wright
- Pranoy Ray
- Rithwik Tom
- Rohit Goswami
- Stefan van der Walt
- Steve Purves
- Sumit Kumar
- Suzana Șerboi
- Talha Irfan
- Tek Kshetri
- Tetsuo Koyama
- Tolulade Ademisoye
- Veronica Gathoni
- Victoria Adesoba
- Wu-Jung Lee

# Posters and Slides

## Keynote Presentations

### Explainable AI for Climate Science: Opening the Black Box to Reveal Planet Earth

Earth's climate is chaotic and noisy. Finding usable signals amidst all of the noise can be challenging: be it predicting if it will rain, knowing which direction a hurricane will go, understanding the implications of melting Arctic ice, or detecting the impacts of humans on the earth's surface. Here, I will demonstrate how explainable artificial intelligence (XAI) techniques can sift through vast amounts of climate data and push the bounds of scientific discovery: allowing scientists to ask "why?" but now with the power of machine learning.

*Elizabeth A. Barnes*

https://doi.org/10.25080/yhec5334

### Particles, People, and Pull Requests

I will tell the story of how the statistical challenges in the search for the Higgs boson and exotic new physics at the Large Hadron Collider led to new approaches to collaborative, open science. The story centers around computational and sociological challenges where software and cyberinfrastructure play a key role. I will highlight a few important changes in perspective that were critical for progress including embracing declarative specifications, pivoting from reproducibility to reuse, and the abstraction that led to the field of simulation-based inference.

*Kyle Cranmer*

https://doi.org/10.25080/rkcg9834

### The Right Tool for the Job

There are many programming languages that we might choose for scientific computing, and we each bring a complex set of preferences and experiences to such a decision. There are significant barriers to learning about other programming languages outside our comfort zone, and seeing another person or community make a different choice can be baffling. In this talk, hear about the costs that arise from exploring or using multiple programming languages, what we can gain by being open to different languages, and how curiosity and interest in other programming languages supports sharing across communities. We'll explore these three points with practical examples from software built for flexible storage and model deployment, as well as a brand new project for scientific computing.

*Julia Silge*

https://doi.org/10.25080/xwen4438

## Accepted Talks

### No-Code-Change GPU Acceleration for Your Pandas and NetworkX Workflows

This talk describes new open-source GPU accelerators from the NVIDIA RAPIDS project for Pandas and NetworkX and will demonstrate how you can enable them for your workflows to experience significant speedups without code changes.

*Rick Ratzel, Vyas Ramasubramani*

https://doi.org/10.25080/jyef9727

### Python for early-stage design of sustainable aviation fuels

We develop a multi-objective, multi-parameter optimization methodology applied to designing novel sustainable aviation fuels

*A.M. Martz, A.E. Comesana, V.H. Rapp, K.E. Niemeyer*

https://doi.org/10.25080/afjf2467

### Introduction to Causal Inference Using pgmpy

In the domain of data science, a significant number of questions are aimed at understanding and quantifying the effects of interventions, such as assessing the efficacy of a vaccine or the impact of price adjustments on the sales volume of a product. Traditional association based methods machine learning methods, predominantly utilized for predictive analytics, prove inadequate for answering these causal questions from observational data, necessitating the use of causal inference methodologies. This talk aims to introduce the audience to the Directed Acyclic Graph (DAG) framework for causal inference. The presentation has two main objectives: firstly, to provide an insight into the types of questions where causal inference methods can be applied; and secondly, to demonstrate a walkthrough of causal analysis on a real dataset, highlighting the various steps of causal analysis and showcasing the use of the pgmpy package.

*Ankur Ankan*

https://doi.org/10.25080/kvta3223

### Coming Online: Enabling Real-Time and AI-Ready Scientific Discovery

A framework for building real-time and AI enabled sensor processing applications

*Adam Thompson, Luigi Cruz*

https://doi.org/10.25080/juet4542

### Expanding the OME ecosystem for imaging data management

OMERO is an open-source solution for image data management which can be customized and hosted by individual institutions, based on the widely used OME data model for microscopy data. Multiple OMERO deployments might be used to provide core delivery, facilitate internal research, or serve as a public data repository. The omero-cli-transfer package facilitates data transfer between these OMERO instances and provides new methods for importing datasets. Another open-source package, ezomero, improves the usability of OMERO in a research environment by providing easier access to OMERO’s Python interface. Along with existing OMERO plugins built for other analysis and viewing software, this positions OMERO to be a hub for image storage, analysis, and sharing.

*Erick Ratamero*

https://doi.org/10.25080/wdya6338

### Free, public, standardized Zarr stores of geospatial data in the cloud for all! Now in Beta.

At the NASA Goddard Earth Sciences (GES) Data and Information Services Center (DISC), we’re doing the heavy lifting to make large geospatial datasets easily accessible from the cloud. No more downloading data. No more worrying about quirky metadata or missing dimensions. No more concatenating hundreds or thousands of files together. Just fire up your Jupyter notebook somewhere in Amazon Web Services (AWS)‘s US-West-2 region, get some free temporary AWS credentials, open our Zarr stores, and start doing your science.

*Christine Smit, Hailiang Zhang, Brianna Pagan, Dieu My Nguyen, James Acker, Ashley Heath, Mahabaleshwara Hegde, Long Pham*

https://doi.org/10.25080/majr5893

**My NumPy year: From no CPython C API experience to shipping a new DType in NumPy 2.0**

Support for string data in NumPy has long been a sore spot for the community. At the beginning of 2023 I was given the task to solve that problem by writing a new UTF-8 variable-length string DType leveraging the new NumPy DType API. I will offer my personal narrative of how I accomplished that goal over the course of 2023 and offer my experience as a model for others to take on difficult projects in the scientific python ecosystem, offering tips for how to get help when needed and contribute productively to an established open source community.

*Nathan Goldbaum*

https://doi.org/10.25080/wdug2226

**Introduction to Causal Inference with Machine Learning**

Causal inference has traditionally been used in fields such as economics, health studies, and social sciences. In recent years, algorithms combining causal inference and machine learning have been a hot topic. Libraries like EconML and CausalML, for instance, are good Python tools that facilitate the easy execution of causal analysis in areas like economics, human behavior, and marketing. In this talk, I will explain key concepts of causal inference with machine learning, show practical examples, and offer some practical tips. Attendees will learn how to apply machine learning to causal analysis effectively, boosting their research and decision-making.

*Hajime Takeda*

https://doi.org/10.25080/tufy8474

**HyperSpy: Your Multidimensional Data Analysis Toolbox**

HyperSpy is a community-developed open-source library providing a framework to facilitate interactive and reproducible analyses of multidimensional datasets. Born out of the electron microscopy scientific community and building on the extensive scientific Python environment, HyperSpy provides tools to efficiently explore, manipulate, and visualize complex datasets of arbitrary dimensionality, including those larger than a system's memory. After 14 years of development, HyperSpy recently celebrated its 2.0 version release. This presentation (re)introduces HyperSpy's features and community, with a focus on recent efforts paring the library into a domain-agnostic core and a robust ecosystem of extensions providing specific scientific functionality.

*Joshua Taillon*

https://doi.org/10.25080/gpna7794

**Ibis and interfaces**

This talk lays out the current database / data landscape as it relates to the SciPy stack, and explores how Ibis (an open-source, pure Python, dataframe interface library) can help decouple interfaces from engines, to improve both performance and portability.

*Gil Forsyth*

https://doi.org/10.25080/fdck7886

### Using Satellite Imagery to Identify Harmful Algal Blooms and Protect Public Health

This talk illustrates how machine learning models to detect harmful algal blooms from satellite imagery can help water quality managers make informed decisions around public health warnings for lakes and reservoirs. Rooted in the development of the open source package CyFi, this talk includes insights around identifying when your model is getting the right answer for the wrong reasons, the upsides of using decision tree models with satellite imagery, and how to help non-technical users build confidence in machine learning models.

*Emily Dorne*

https://doi.org/10.25080/ghpx3574

### An Introduction to Impact Charts

Impact charts, as implemented in the impactchart package, make it easy to take a data set and visualize the impact of one variable on another in ways that techniques like scatter plots and linear regression can't, especially when there are other variables involved. In this talk, we introduce impact charts, demonstrate how they find easter-egg impacts we embed in synthetic data, show how you can create your first impact chart with just a few lines of code, and show how impact charts can find hidden impacts in a real-world use case.

*Darren Vengroff, Ph.D.*

https://doi.org/10.25080/tfaj6588

### ITK-Wasm: Universal spatial analysis and visualization

How WebAssembly makes scientific computing accessible, sustainable, and reproducible

*Matthew McCormick*

https://doi.org/10.25080/pghc3745

### Making Research Data Flow with Python

Telescopes exist in remote environments, and yet produce huge amounts of data. This presentation is about building the data transfer tool Librarian for the Simons Observatory, which enables seamless shifting between internet-enabled transfers and hand-carrying disks down mountains.

*Josh Borrow*

https://doi.org/10.25080/ttdf6694

### Monte Carlo/Dynamic Code: Performant and Portable High-Performance Computing at Scale via Python and Numba

Monte Carlo / Dynamic Code (MC/DC) is a Monte Carlo neutron transport solver targeting high performance computing. MC/DC is acclerated using the Numba compiler and has the capibility to run on CPUs and GPUs. This talk describes the development of MC/DC

*Joanna Piper Morgan, Kyle E. Niemeyer*

https://doi.org/10.25080/cdrf9272

### Starsim: A flexible framework for agent-based modeling of health and disease

Starsim is an open-source agent-based modeling framework for simulating the spread of diseases among agents via dynamic transmission networks. This talk describes the Starsim package and gives an example of how it can be used to model HIV and syphilis.

*Cliff Kerr, Robyn Stuart, Romesh Abeysuriya, Paula Sanz-Leon, Jamie Cohen, Daniel Klein*

https://doi.org/10.25080/ukpu4584

### anywidget: custom Jupyter Widgets made easy

anywidget simplifies the creation and distribution of Jupyter Widgets by providing a portable and reusable specification and toolset. It ensures cross-platform compatibility with notebook platforms, lowers the barrier to entry, and improves reusability and interoperability in interactive computing environments. This talk highlights the motivation to bring the web and Python ecosystems closer together, showcasing community-driven anywidgets and new widgets that push the boundaries of these platforms.

*Trevor Manz*

https://doi.org/10.25080/wdhm9848

### Pooch: A friend to fetch your data files

Easily download and cache data files from the web.

*Santiago Soler*

https://doi.org/10.25080/frkj7844

### scikit-build-core: A modern build-backend for CPython C/C++/Fortran/Cython extensions

A presentation about scikit-build-core exploring how it modernizes Python extension building by integrating CMake with Python packaging standards, enabling seamless cross-compilation and multi-platform support. The slides highlight key features such as simplified configuration, support for multiple languages like C++, Fortran, and Cython, and the transition from the classic scikit-build. They also provide insights into how scikit-build-core enhances the development experience and streamlines Python module creation for diverse environments.

*Jean-Christophe Fillion-Robin, Henry Schreiner, Matt McCormick*

https://doi.org/10.25080/xjvg7399

### Sparse Arrays in scipy.sparse

The shift from sparse matrices to sparse arrays in SciPy. What’s coming, migration and API decisions.

*Dan Schult*

https://doi.org/10.25080/ejft5676

### Building a modular simulation platform for magnetic resonance force microscopy (MRFM) experiments (mrfmsim and mmodel)

We present mrfmsim, an open-source framework that not only facilitates the design, simulation, and signal validation of magnetic resonance force microscopy experiments, but also significantly speeds up the development process. In the talk, we present the challenges in building simulation packages for experiments undergoing continuous development in a graduate research setting. We show how we designed mrfmsim and its backend mmodel for modularity, extendibility, and readability, and how these design principles translate into practical benefits for researchers in the field.

*Peter Sun, John Marohn*

https://doi.org/10.25080/xpnp9684

**Vector space embeddings and data maps for cyber defense**

Cyber defense, and in particular threat detection, requires gaining insight from very large amounts of telemetry data, using unsupervised learning. This talk presents a method for embedding such data in vector spaces and exploring it through interactive visualization and labeling, using *data maps*. We demonstrate this method by looking at the baseline behaviours of hosts in the open ACME3 dataset of host-based telemetry.

*Benoit Hamelin, John Healy*

https://doi.org/10.25080/uykm3443

**Simplifying analysis of hierarchical HDF5 and NetCDF4 files with xarray-datatree**

Xarray-datatree, is a Python package that supports HDFs (Hierarchical Data Format) with hierarchical group structures by creating a tree-like hierarchical data structure in xarray.

*Eniola Awowale, Tom Nicholas, Lucas Sterzinger, Nick Lenssen*

https://doi.org/10.25080/xfex7842

## Accepted Posters

**Training a Supervised Cilia Segmentation Model from Self-Supervision**

Understanding cilia behavior is essential in diagnosing and treating such diseases. But, the tasks of automatically analysing cilia are often a labor and time-intensive since there is a lack of automated segmentation. In this work we overcome this bottleneck by developing a robust, self-supervised framework exploiting the visual similarity of normal and dysfunctional cilia. This framework generates pseudolabels from optical flow motion vectors, which serve as training data for a semi-supervised neural network. Our approach eliminates the need for manual annotations, enabling accurate and efficient segmentation of both motile and immotile cilia.

*Seyed Alireza Vaezi, Shannon Quinn*

https://doi.org/10.25080/hfew4757

**Domovyk: Multilingual Transliteration for Cyrillic Text**

The Domovyk package provides transliteration to and from Cyrillic alphabets in a way that addresses some limitations in existing packages, providing multilingual functionality, support for composite Unicode characters, and support for languages not addressed in other packages, such as Church Slavonic and Carpatho-Rusyn. Domovyk aims to increase the accessibility of transliteration technologies for users working in these languages, focusing on use cases that require thorough and accurate transliteration.

*Ian Goodale*

https://doi.org/10.25080/udkt5322

**ncompare: A Python Package for Comparing netCDF Structures**

As netCDF (Network Common Data Form) files are widely used in Earth science — with climate models, oceanographic or atmospheric reanalyses, and observational data — improved means of evaluating netCDF files can help enable a wide range of applications. We have developed a reusable open source approach through `ncompare`, which is a Python package for comparing netCDF structures. `ncompare` facilitates rapid comparisons by generating a formatted display of the matching and non-matching groups, variables, and associated metadata between two NetCDF datasets. The user has the option to colorize the terminal output for ease of viewing, and `ncompare` can optionally save comparison reports in text, comma-separated value (CSV), and/or Microsoft Excel formats.

*Daniel E. Kaufman, Walter E. Baskin, Julia S. Lowndes*

https://doi.org/10.25080/etnj4973

### Development and Application of CWGID: the California Wildfire GeoImaging Dataset for Deep Learning Driven Forest Wildfire Detection

This poster presents the development and application of the CWGID (California Wildfire GeoImaging Dataset), a comprehensive dataset for deep learning-driven forest wildfire detection. The study explores the dataset creation process, its application in wildfire detection using deep learning techniques, and the results obtained.

*Valeria Martin, K. Brent Venable, Derek Morgan*

https://doi.org/10.25080/kgpe7737

### Fast and Easy Graph Analytics with the NetworkX Ecosystem of Backends

Poster describing the function dispatching features of NetworkX and the various backends currently available.

*Rick Ratzel, Dan Schult*

https://doi.org/10.25080/ryga7653

### RoughPy: Streaming data is rarely smooth

RoughPy is a library that aims to connect data science with the mathematics of rough paths to provide a new perspective for working with streamed data. The Stream object provided by the library is an abstraction of streamed data so that it can be viewed through the lens of rough path theory. This makes the high order representation of the data (the signature) available as a tool to be used in data science and machine learning applications. This poster outlines the mathematics, the applications and data, and how RoughPy brings both sides together.

*Sam Morley*

https://doi.org/10.25080/yfnx8796

### Mamba Models:A Potential Replacement for Transformers?

Mamba models leverage State Space Models and the HiPPO framework to efficiently handle long-range dependencies, reducing computational complexity compared to traditional transformers.

*Suvrakamal Das*

https://doi.org/10.25080/txwe5647

### Employing the strengths of Generative AI supports the execution of time series analysis and forecasting

This poster explores the use of Generative AI models for time series analysis and forecasting, specifically in the context of energy consumption. It compares traditional statistical methods, such as ARIMA, with advanced AI-based techniques like AutoGluon-TimeSeries, xLSTM, and TimeGPT. The study aims to demonstrate the efficiency and accuracy of these methods using real-world energy data from the PJM Interconnection LLC. Results indicate that AI-based models, particularly xLSTM and AutoGluon-TimeSeries, outperform traditional models in forecasting accuracy, showcasing their potential for better resource management and decision-making in climate change mitigation.

*Ying-Jung Chen*

https://doi.org/10.25080/xtng2642

**Parallel Graph Algorithms and Building Backends with Entry Points**

Hi! Have you ever wished your pure Python libraries were faster? Or wanted to fundamentally improve a Python library by rewriting everything in a faster language like C or Rust? Well, wish no more... NetworkX's backend dispatching mechanism redirects your plain old NetworkX function calls to a FASTER implementation present in a separate backend package by leveraging Python's `entry_point` specification! NetworkX is a popular, pure Python library used for graph (aka network) analysis. But when the graph size increases (like a network of everyone in the world), NetworkX algorithms could take days to solve a simple graph analysis problem. To address these performance issues, this backend dispatching mechanism was recently developed. This poster explores NetworkX's parallel backend that utilizes Joblib to run graph algorithms on multiple CPU cores and how we can use it just by specifying a `backend` keyword argument or by passing the backend graph object (type-based dispatching). It also goes over some of the future ToDos for the nx-parallel backend, the speedups obtained, and some important notes to ponder about. Last but not at all the least, it depicts the ideal pipeline starting from networkx, going on to nx-parallel, then to joblib, and then towards the various parallel libraries. (nx-parallel GitHub repo - https://github.com/networkx/nx-parallel) Thank you :)

*Aditi Juneja*

https://doi.org/10.25080/ypkc2577

**Aeromancy: Towards More Reproducible AI and Machine Learning**

We present Aeromancy, an opinionated philosophy and open-sourced framework that closely tracks experimental runtime environments for more reproducible machine learning. In existing experiment trackers, it's easy to miss important details about how an experiment was run, e.g., which version of a dataset was used as input or the exact versions of library dependencies. Missing these details can make replicability more difficult. Aeromancy aims to make this process smoother by providing both new infrastructure (a more comprehensive versioning scheme including both system runtimes and external datasets) and a corresponding set of best practices to ensure experiments are maximally trackable.

*David McClosky*

https://doi.org/10.25080/yyvd5799

**Leveraging FAIR principles for efficient management of meteorological radar data**

Radars are crucial in meteorology for their precise spatio-temporal resolution, enabling early detection and tracking of severe weather. This capability aids meteorologists in issuing timely alerts, thus safeguarding lives and reducing property damage. Radar data also supports offline applications like cloud and precipitation analysis, climatology, and insurance risk assessment, all relying on its time-series nature. However, storing radar data traditionally involves proprietary formats with high I/O demands, leading to slow computations and resource-intensive requirements.

To address these challenges, a new data model is proposed using the CF format-based FM301 hierarchical tree structure and ARCO formats. This model efficiently organizes radar data into cloud-storage buckets using Python libraries like Xarray, Xradar, Wradlib, and Zarr. Demonstrated with Carimagua, Colombia radar data, the model shows faster processing times than legacy methods on standard hardware. Emphasizing FAIR principles (Findable, Accessible, Interoperable, Reusable), this approach enhances accessibility to radar data on cloud platforms, promoting open science and wider societal benefit.

*Alfonso Ladino, Maxwell Grover, Stephen Nesbitt, Kai Mühlbauer*

https://doi.org/10.25080/wnaf9823

**Building Quantum Bridges: Advancing Drug Discovery with QAOA and Explaining Quantum Computing with Building Blocks**

Using quantum computers to solve combinatorial optimization problems

*B. Maurice Benson*

https://doi.org/10.25080/mtdn2862

**Building sustainability and community in a small project: lessons from working on SaltProc**

SaltProc is an open source tool for simulating batch-wise reprocessing of fuel in nuclear reactors developed around an export controlled depedency. This limited the size of the userbase. I contributed features to SaltProc to support an open-source alternative, and found that this change attracted new users.

*Oleksandr R. Yardas, Madicken Munk*

https://doi.org/10.25080/ercj5799

**Open Source Farm to Open Science Table: Project Pythia's Cook-off Hackathons**

This poster describes Project Pythia's annual community hackathons, aka Cook-offs. These summer sprints blur the lines between scientist and software developer at an individual and group level, and seed excitement and commitment to the open source, open science community.

*M. Drew Camron, Kevin Tyle*

https://doi.org/10.25080/rrdv2565

**Accelerating the use of Lagrangian data with Clouddrift**

clouddrift is a python library built to accelerate and simplify the use of lagrangian datasets in science. To achieve this the library adapts datasets into cloud optimized ragged arrays, provides analysis and query methods to work with lagrangian datasets as ragged arrays.

*Santana, Kevin, Elipot, Shane, Miron, Philippe, Curcic, Milan*

https://doi.org/10.25080/ftyc2662

**Geist: a multimodal data transformation, query, and reporting language**

Geist is a new templating language for declarative data manipulation, query, and report generation. Building on the Jinja template engine, Geist is designed to support diverse data backends and query engines via predefined tags and filters, and may be extended with custom tags. A single Geist template may include multiple queries expressed in different languages, e.g. SQL and SPARQL, to leverage the strengths of each for clarity and ease of maintenance. Because Geist both can generate reports in diverse formats and perform inserts and updates on new or existing databases during template expansion, Geist templates may orchestrate data extraction, transformation, and load operations spanning multiple tools and data storage systems. Geist also enables modularity in query languages and eliminates messy procedural programs. Geist aims to enable developers to use whatever language or tools they like regardless of where the data is stored.

*Meng Li, Timothy McPhillips, Bertram Ludäscher*

https://doi.org/10.25080/geaj2635

**zfit: scalable pythonic likelihood fitting**

zfit is a highly scalable and customizable model manipulation and likelihood fitting library. It uses the same computational backend as TensorFlow and is optimised for simple and direct manipulation of probability density functions.

*Jonas Eschle, Albert Puig Navarro, Rafael Silva Coutinho, Matthieu Marinangeli, Nicola Serra, Iason Krommydas*

https://doi.org/10.25080/xwwd2556

**Convolutional Autoencoders for Denoising Solar Images**

Using scientific packages in Python, we trained convolutional autoencoders to improve the quality of solar images taken from the Atmospheric Imaging Array (AIA).

*Jimmy Lynch*

https://doi.org/10.25080/fpju4745

**Facilitating scientific investigations from long-tail data with Python**

This presentation walks through our work on creating a non-nanosecond datetime for Pandas and the development of Python toolboxes to query databases and analyze datasets using Pandas objects for interoperability.

*Deborah Khider, Varun Ratnakar, Julien Emile-Geay, Kim Pevey, Marco Gorelli, Nicholas McKay, Alexander James, Jordan Landers*

https://doi.org/10.25080/jfcn6576

**Climatic and Geographic Influences on Cumacea Genetics in the Northern North Atlantic**

Cumacea crustaceans serve as vital indicators of benthic health in marine ecosystems. This study investigates the influence of environmental parameters on their genetic makeup in the North Atlantic, focusing on Icelandic waters. We analyze mitochondrial 16S rRNA gene sequences from 62 Cumacea specimens collected across varying depths.

*Justin Gagnon, Nadia Tahiri*

https://doi.org/10.25080/eage2654

**From concept to compute: accelerating research with workflow management in `pyiron`**

`pyiron_workflow` is a python framework for developing research workflows based on composing individual function nodes together into computational graphs. Each node class can be defined by simply applying a decorator to a regular python function, allowing user-developers to extend `pyiron` functionality even without deep knowledge of Object-Oriented Programming. Nodes can be grouped together into macro nodes using the same simple function-and-decorator approach, such that complex workflows can be built up and simply represented by composing and nesting these graphs. In contrast to user workflows being defined via a series of Jupyter notebook cells, this approach rigorously defines workflows by their graph topology and allows them to be easily shared – and incorporated into new contexts – by sharing/importing the workflow as a macro node. Interoperability can be controlled with (optional) type checking on data connections between nodes.

*Liam Huber, Joerg Neugebauer*

https://doi.org/10.25080/xwwj3999

### 3D plotting and mesh analysis through a streamlined interface for the Visualization Toolkit (VTK)

VTK implements an object-oriented approach to 3D visualization, and PyVista adheres to that underlying structure to provide an API that expands on VTK's data types. These expanded, wrapped types hold methods and attributes for quickly accessing scalar arrays, inspecting properties of the dataset, or using filtering algorithms to transform datasets. PyVista wrapped objects have a suite of common filters ready for immediate use directly on the objects. These filters are commonly used algorithms in the VTK library that have been made more accessible by binding a method to control that algorithm directly onto all PyVista datasets, providing a shared set of functionality. Through the use of these bound filtering methods, powerful VTK algorithms can be leveraged and controlled via keyword arguments designed to be intuitive for novice users.

*Tetsuo Koyama*

https://doi.org/10.25080/vxvf4964

### Vectorized Quadrature, Series Summation, Differentiation, Optimization, and Rootfinding in SciPy

Until recently, SciPy's best options for scalar quadrature, minimization, and root finding called compiled code, which could not take advantage of a vectorized Python integrand, objective function, or residual function; SciPy offered no functions for accurate numerical differentiation or series summation. These gaps are being filled with a family of pure-Python, array API compatible functions for dramatically faster vectorized calculation of scalar integrals, infinite sums, derivatives, minimizers, and roots.

*Matt Haberland, Albert Steppi, Pamphile T. Roy*

https://doi.org/10.25080/uyyk2727

### Spyder and the NumFOCUS SDG program: better UI/UX, improved code completion and lessons learned

Spyder is a free and open source scientific environment written in Python, for Python, and designed by and for scientists, engineers and data analysts. Thanks to the NumFOCUS Small Development Grants (SDG) program, from late 2022 through the beginning of 2024 we've made many improvements to Spyder's UI/UX and to its support for external code completion plugins.

We’d like to share the work we’ve done, and explain the ideas and execution behind two of our proposals, from both a technical and project management perspective.

*Daniel Althviz Moré, Juan Sebastian Bautista Rojas*

https://doi.org/10.25080/gdhp7664

**Scientific Publishing with MyST Markdown**

Supercharge your scientific writing with MyST Markdown! MyST is designed for technical communication and research publication. Sprinkling in all the typesetting power that you’d get from LaTeX but with the pleasure of lightweight content focussed writing that you get from Markdown, create interactive papers or documentation that link with data, have embedded computation and provide explorable notebooks to your readers. Get your work out to a templated PDF for traditional publication from exactly the same material, all from the command line or with continuous integration. In this talk, we’ll show how MyST works, how to get started and some examples of awesome MyST based publications.

*Steve Purves, Rowan Cockett*

https://doi.org/10.25080/yrmn7235

**gravitational lensing simulations made user friendly with Caustics’ three interface levels**

We present Caustics, a tool to accelerate the analysis of gravitational lensing systems for the next generation of astronomical data. Caustics will enable precision measurements of dark matter properties, the expansion rate of the Universe, lensed black holes, the first stars, and more. In this talk I will the benefits and challenges of how we used PyTorch (a differentiable and GPU accelerated scientific python package) to allow for fast development without sacrificing numerical performance. I will detail our development process as well as how we encourage users of all skill levels to engage with our documentation/tools.

*Connor Stone*

https://doi.org/10.25080/gwcm5548

**Seamless integration across developer ecosystems with python and wasm in VTK**

Visualizing large-scale simulation datasets is vital for research in scientific and engineering fields. To this end, VTK, an open-source C++ visualization library, and its Python counterparts VTK Python, PyVista and Vedo are streamlining 3D visualization, though shortcomings remain. We significantly improved VTK’s automated wrappers and enabled VTK WebAssembly thus simplifying creation of visualizations in Python or on the web using Trame. This also made it easier to execute VTK native code directly in a web browser. We discuss challenges and lessons learned in extending large C++ software to diverse languages through API wrapping and cross-compilation to ultimately benefit the Python community.

*Jaswant Panchumarti, Sebastien Jourdain, Berk Geveci, Aashish Chaudhary*

https://doi.org/10.25080/fyvp8946

**Venturial: Generating CFD Workflows in Python**

Venturial is an open-source suite of interactive Python tools for Computational Fluid Dynamics (CFD). Venturial, envisions to benefit the scientific python community by providing an easy-to-understand CFD application building workflow within the Python environment.

*Rajdeep Adak, Janani Srree Murallidharan, Prabhu Ramachandran* https://doi.org/10.25080/tpwg2365

## SciPy Tools Plenaries

### SciPy Tools Plenary on the Journal of Open Source Software (JOSS)

Updates from the Journal of Open Source Software (JOSS) on new work and improvements to the journal in 2023 and 2024.

*Matthew Feickert*

https://doi.org/10.25080/pcna4769

### SciPy Tools Plenary on Matplotlib

Matplotlib is a comprehensive library for creating static, animated, and interactive visualizations in Python. This presentation summarizes changes over the past year, new features, and future plans.

*Elliott Sales de Andrade*

https://doi.org/10.25080/guwd9846

### NumPy annual update

Comprehensive overview of the latest release and major milestones for the NumPy library and its contributor community

*Inessa Pawson*

https://doi.org/10.25080/dpha2486

### SciPy: not just a conference

Besides a conference, ‘SciPy’ is also a library that provides fundamental building blocks for modeling and solving scientific problems. SciPy includes algorithms for optimization, interpolation, statistics, fast Fourier transforms, and many other classes of problems; it also provides specialized data structures, such as k-dimensional trees and sparse matrices. This presentation summarizes the current status and future plans of the SciPy library.

*Matt Haberland*

https://doi.org/10.25080/mtvj6294

### Sphinx-gallery and Sphinx-tags

Sphinx-gallery and sphinx-tags features and updates

*Hannah Aizenman*

https://doi.org/10.25080/ddee5226

## Lightning Talks

### Deploying Python environments on top of Mt. Rainier

A process for building preparing artifacts that enable the deployment of a Python computing environment in a place where there is no Internet access.

*Benoit Hamelin*

https://doi.org/10.25080/pyyu4582

### Hello Project!

tips for onboarding into contributing to open source

*Hannah Aizenman*

https://doi.org/10.25080/hxxk4957

**swiftascmaps: A colour map library for swifties**

A hopefully comedic talk about the colour map library swiftascmaps.

*Josh Borrow*

https://doi.org/10.25080/uncn2995

**Sciris: Simplifying scientific Python**

Sciris aims to streamline the development of scientific Python code by making it easier to perform common tasks. This example illustrates how the same block of fairly typical scientific Python code – which performs tasks like collecting data from a function running in parallel, saving and loading files, and 3D plotting – looks like when written in ‘vanilla Python’ compared to using Sciris.

*Cliff Kerr, Paula Sanz-Leon, Romesh Abeysuriya*

https://doi.org/10.25080/knjj9332

**Plotting Slides in Matplotlib**

Matplotlib makes easy things easy and hard things possible, like this silly idea of making slides in it.

*Elliott Sales de Andrade*

https://doi.org/10.25080/wrvp6756

**For the python evangelist**

Because clearly python needs to make music.

*Christine Smit*

https://doi.org/10.25080/hcya9443

**Renovate: Automating Dependency Management**

A brief introduction to Renovate, a tool for automating dependency management in software projects. The slides can be accessed at https://paddyroddy.github.io/talks/renovate-automating-dependency-management. Two example parent configurations I maintain are available at https://github.com/paddyroddy/.github/tree/main/renovate and https://github.com/UCL-ARC/.github/tree/main/renovate.

*Patrick J. Roddy*

https://doi.org/10.25080/tkky5633

## Sponsored Students

### Scholarship Recipients

- Amadi Gabriel Udu, *University of Leicester*
- Ankur Ankan, *Radboud University*
- Ayush Nag, *University of Washington*
- C.A.M. Gerlach, *University of Alabama Huntsville*
- Elliot Salisbury, *University of Southampton*
- Erick Martins Ratamero, *The Jackson Laboratory*
- JT Thielen, *Colorado State University*
- Mohamed El Shorbagy, *Ain Shams University*
- Sheku Shafeie, *TechPoint*
- Tek Kshetri, *University of Calgary*
- Willy Menacho, *Universidad Técnica Federico Santa María*
- Ying-Jung Chen, *University of Washington eScience*

### NumFOCUS Diversity Scholarship Recipients

- Atharva Rasane, *DaSouk / GDG Belgaum*
- Hannah Aizenman, *City College of New York*
- Jacqui Levy, *Environment and Climate Change Canada*
- Juanita Gomez, *University of California Santa Cruz*
- Juliana Ferreira Alves, *Itaú Unibanco*
- Meng Li, *Rice University*
- Tetsuo Koyama, *ARK Information Systems*

## Table of Contents

Open Access | https://doi.org/10.25080/FGCJ9164

SciPy 2024
July 8 - July 14, 2024

Proceedings of the 23rd
Python in Science Conference
ISSN: 2575-9752

# multinterp

## A Unified Interface for Multivariate Interpolation in the Scientific Python Ecosystem

**Alan Lujan**[1,2]

[1]Johns Hopkins University, [2]Econ-ARK

**Abstract**

Multivariate interpolation is a fundamental tool in scientific computing used to approximate the values of a function between known data points in multiple dimensions. Despite its importance, the Python ecosystem offers a fragmented landscape of specialized tools for this task. This fragmentation hinders code reusability, experimentation, and efficient deployment across diverse hardware. The `multinterp` package was developed to address this challenge. It provides a unified interface for various grids used for interpolation (regular, irregular, and curvilinear), supports multiple backends (CPU, parallel, and GPU), and includes tools for multivalued interpolation and interpolation of derivatives. This paper introduces `multinterp`, demonstrates its capabilities, and invites the community to contribute to its development.

**Keywords** multivariate, interpolation, gpu, rectilinear, curvilinear, scattered

**Published** Jul 10, 2024

**Correspondence to**
Alan Lujan
alanlujan91@gmail.com

## 1. Introduction

In scientific computing, interpolation is a fundamental method used to approximate new data points within a range of known functional data points. When this concept is applied to functions with multiple variables, it is known as multivariate interpolation. This technique is crucial for various scientific applications, including data analysis, numerical modeling, and visualizing complex datasets.

Over time, the scientific Python ecosystem has developed various specialized tools for multivariate interpolation. These powerful tools are spread across different packages, each designed for specific purposes and for distinct data types. This scattering of tools across multiple packages creates challenges for researchers and practitioners.

One major issue is the inconsistency of interfaces across different packages. The varying syntax and usage patterns make it difficult for users to switch between interpolation methods or compare their performance effectively. Additionally, many existing tools are designed for CPU-only execution, lacking GPU acceleration or parallel processing support. This limitation can significantly impact performance, especially when dealing with large datasets or complex interpolation tasks.

Another challenge stems from the restricted functionality of some packages. Certain tools focus solely on specific types of interpolation, such as those for structured data, while lacking support for advanced features like multivalued interpolation or derivative calculations. This specialization often forces researchers to use multiple packages to cover all their interpolation needs, leading to an inefficient workflow. The need to learn and integrate various packages increases development time and introduces potential inconsistencies in the codebase.

The `multinterp` package was developed to address these challenges to provide a comprehensive framework for multivariate interpolation in python with several key features. It offers a unified interface for various interpolation grids, including regular (rectilinear), irregular (unstructured), and curvilinear grids, allowing users to easily switch between different grid types without changing their code structure.

Furthermore, `multinterp` supports multiple backends, including CPU (using `numpy` and `scipy`), parallel processing (using `numba`), and GPU acceleration (using `cupy`, `pytorch`, and `jax`). This flexibility enables users to optimize performance based on their available computational resources. The package also includes tools for multivalued interpolation and interpolation of derivatives, expanding its utility for a wide range of scientific applications.

Perhaps most importantly, `multinterp` provides a consistent API that allows easy switching between different interpolation methods and backends. This feature simplifies code development and facilitates experimentation with various interpolation techniques and performance optimization strategies.

In the following sections, we will introduce the main concepts of grid interpolation, demonstrate the capabilities of `multinterp` for different types of grids, and compare its performance with existing tools. We will conclude by discussing the project's current state and inviting community contributions to further development.

## 2. Background and Concepts

Before diving into the specifics of `multinterp`, it is crucial to understand some fundamental concepts in multivariate interpolation and the challenges they present.

### *2.1. Interpolation Basics*

At its core, interpolation is about approximating functional values between known data points. This might involve drawing a line or curve between points in one dimension. The process becomes more complex in multiple dimensions, involving surfaces or hypersurfaces.

### *2.2. Grid Types*

The arrangement of known data points, called the grid, significantly influences the choice of interpolation method and its computational efficiency. In `multinterp`, we consider three main types of grids:

1. **Rectilinear Grids**: These are the simplest and most common. Data points form a regular pattern along each dimension, though the spacing between points may vary. Moreover, these grids can be represented by the cross product of 1-dimensional vectors, making them easy to work with.
2. **Curvilinear Grids**: These maintain a structured relationship between neighboring points, but the grid lines may be curved or warped. Their main advantage is that these grids are monotonic in each coordinate dimension, allowing for an easy and efficient transformation into a rectilinear grid.
3. **Unstructured Grids**: These have irregularly spaced data points with no inherent structure. They are often encountered in experimental data collection or adaptive numerical methods. Interpolation on unstructured grids is more challenging and requires more sophisticated algorithms.

### *2.3. Existing Interpolation Methods*

Various methods exist for interpolation, each with its strengths and limitations:

**Table 1**. *Grids and structures implemented in "multinterp".*

| **Grid** | **Structure** | **Geometry** |
|---|---|---|
| Rectilinear | Regular | Rectangular mesh |
| Curvilinear | Regular | Quadrilateral mesh |
| Unstructured | Irregular | Random |

- **Nearest Neighbor**: Assigns the value of the nearest known point or a weighted average of the nearest few points. Simple but can be inaccurate.
- **Linear/Multilinear**: Assumes a linear relationship between points. Fast but can miss complex patterns.
- **Polynomial**: Higher-order polynomials are used to fit the data. It can capture more complex relationships but may oscillate between points.
- **Spline**: Uses piecewise polynomials. Offers a balance between smoothness and accuracy.
- **Radial Basis Function (RBF)**: Uses a sum of radially symmetric functions. Useful for scattered data.

Each of these methods may be implemented differently for different grid types, leading to the fragmentation in the Python ecosystem that `multinterp` aims to address. `multinterp` currently mainly implements multilinear interpolation and aims to add more methods and grid types in the future.

## 3. THE `multinterp` PACKAGE

The `multinterp` package is a unified framework for multivariate interpolation in python, designed to address the challenges outlined in the previous sections. It offers several key features:

1. **Unified Interface**: A consistent API for interpolation, regardless of data structure or desired backend, reducing the learning curve and promoting code reusability. This consistency extends across different grid types (rectilinear, curvilinear, and unstructured), allowing users to switch between grid types with minimal code changes.
2. **Hardware Adaptability**: Seamless support for CPU (`numpy`, `scipy`), parallel (`numba`), and GPU (`cupy`, `pytorch`, `jax`) backends, empowering users to optimize performance based on their computational resources.
3. **Broad Functionality**: Tools for regular/rectilinear interpolation, multivalued interpolation, and derivative calculations, addressing a wide range of scientific problems.

The `multinterp` package offers several advantages over existing tools:

1. **Flexibility**: Unlike specialized packages that focus on specific grid types or interpolation methods, `multinterp` provides a unified framework that can handle various grid types and interpolation scenarios.
2. **Performance**: By supporting multiple backends, including GPU acceleration, `multinterp` can offer significant performance improvements over CPU-only implementations, especially for large datasets. Our benchmarks show up to 10x speedup for large 3D grids when using GPU backends.
3. **Ease of Use**: The consistent API across different grid types and backends simplifies the learning curve and makes it easier for users to experiment with different approaches. For instance, switching from CPU to GPU interpolation often requires changing a single parameter.

Currently, `multinterp` primarily implements multilinear interpolation, which balances simplicity, speed, and accuracy for many applications. This method is particularly efficient for rectilinear grids. The core implementation uses efficient `numpy` operations for fast computation.

However, we recognize the limitations of multilinear interpolation, such as its inability to capture complex, non-linear relationships in the data. Future development plans include implementing additional interpolation methods such as polynomial, spline, and radial basis function interpolation. These methods can offer improved accuracy for certain types of data, albeit at the cost of increased computational complexity.

The `multinterp` package is currently in its beta stage, but it offers a strong foundation and welcomes community contributions to reach its full potential. We invite collaboration to improve documentation, expand the test suite, and ensure the codebase aligns with the highest standards of Python package development.

## 4. Rectilinear Interpolation

Rectilinear grids are the foundation of many scientific computing applications. They are simple yet powerful, allowing for efficient data representation and manipulation. In a rectilinear grid, data points form a regular pattern along each dimension, though the spacing between points may vary.

### *4.1. The Basics of Rectilinear Grids*

A rectilinear grid in 2D can be thought of as a sheet of graph paper where the lines might not be evenly spaced. In 3D, you can imagine a stack of these sheets, again with potentially varying spacing between them. Mathematically, we can represent a 2D rectilinear grid using two 1D arrays of increasing values:

$$\begin{aligned} x &= [x_0, x_1, x_2, ..., x_n] \\ y &= [y_0, y_1, y_2, ..., y_m] \end{aligned} \tag{1}$$

where $x_i < x_j$ and $y_i < y_j$ for all $i < j$. The full grid is then represented by the tensor product $x \times y$, resulting in dimensions $n \times m$.

This structure allows for efficient interpolation algorithms, as we can quickly locate the nearest known values to predict the function's behavior in unknown spaces. This is because we only need to search each dimension once and independently of the other.

### *4.2. Multilinear Interpolation with `multinterp`*

The `multinterp` package provides a straightforward yet powerful implementation of multilinear interpolation. It supports various backends, including `numpy`, `scipy`, `numba`, `cupy`, `pytorch`, and `jax`, allowing users to choose the most suitable option for their computational environment.

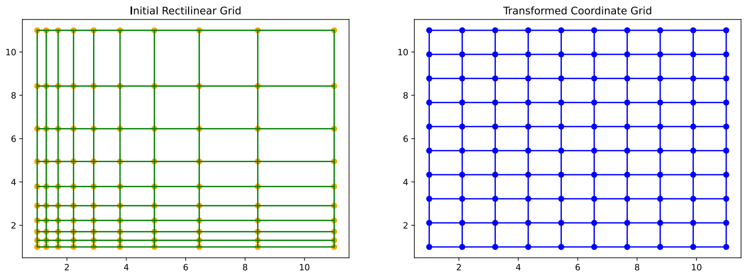


**Figure 1.** *Transformation between non-uniform and uniform rectilinear grids. This process is critical to efficient interpolation on rectilinear grids.*

At the heart of `multinterp`'s rectilinear interpolation is the `map_coordinates` function from `scipy.ndimage`. This function is versatile, taking an array of input values and an array of coordinates and returning interpolated values at those coordinates. Here is how it works:

- The `input` array contains known values on a coordinate (index) grid. For example, `input[i,j,k]` is the known value at coordinate `(i,j,k)`.
- The `coordinates` array contains fractional coordinates where we want to interpolate. For instance, `coordinates[0] = (1.5, 2.3, 3.1)` indicates we want to interpolate at a point between integer grid coordinates.

However, real-world functions are often not defined on a simple coordinate grid. This is where `multinterp`'s `get_coordinates` function comes in. It maps the functional input grid (defined on real numbers) and the coordinate grid (defined on non-negative integers).

Let us look at a practical example to see how this works:

```
import `numpy` as np
from multinterp import MultivariateInterp

# Define a function to interpolate
def squared_coords(x, y):
    return x**2 + y**2

# Create a non-uniform rectilinear grid
x_grid = np.geomspace(1, 11, 11) - 1
y_grid = np.geomspace(1, 11, 11) - 1
x_mat, y_mat = np.meshgrid(x_grid, y_grid, indexing="ij")

# Evaluate the function on this grid
z_mat = squared_coords(x_mat, y_mat)

# Create the interpolator
interp = MultivariateInterp(z_mat, [x_grid, y_grid])

# Define points for interpolation
x_new, y_new = np.meshgrid(
 np.linspace(0, 10, 11),
 np.linspace(0, 10, 11),
    indexing="ij",
)

# Perform interpolation
z_interp = interp(x_new, y_new)
```

In this example, we use a squared coordinates function (`x^2+y^2`) as our test case. This function is chosen because it produces a simple curved surface in 3D, making it easy to verify the interpolation results visually. We create a non-uniform grid using `geomspace`, which gives us more points near the origin and fewer points farther out. This non-uniform grid demonstrates `multinterp`'s ability to handle varying grid spacings effectively.

The `MultivariateInterp` class encapsulates the interpolation process, making it easy to create an interpolator and apply it to new points. This object-oriented approach allows for easy reuse and chaining of operations, as seen in the next section on derivatives.

### *4.3. Derivatives*

The `multinterp` package also allows calculating derivatives of the interpolated function defined on a rectilinear grid. This is done by using the function `get_grad`, which wraps `numpy`'s `gradient` function to calculate the gradient of the interpolated function at the given coordinates.

Consider the following function along with its analytical derivatives:

```
def trig_func(x, y):
    return y * np.sin(x) + x * np.cos(y)


def trig_func_dx(x, y):
    return y * np.cos(x) + np.cos(y)


def trig_func_dy(x, y):
    return np.sin(x) - x * np.sin(y)
```

First, we create a sample input gradient and evaluate the function at those points. Notice that we are not using the analytical derivatives to create the interpolation function. Instead, we will use these to compare the results of the numerical derivatives.

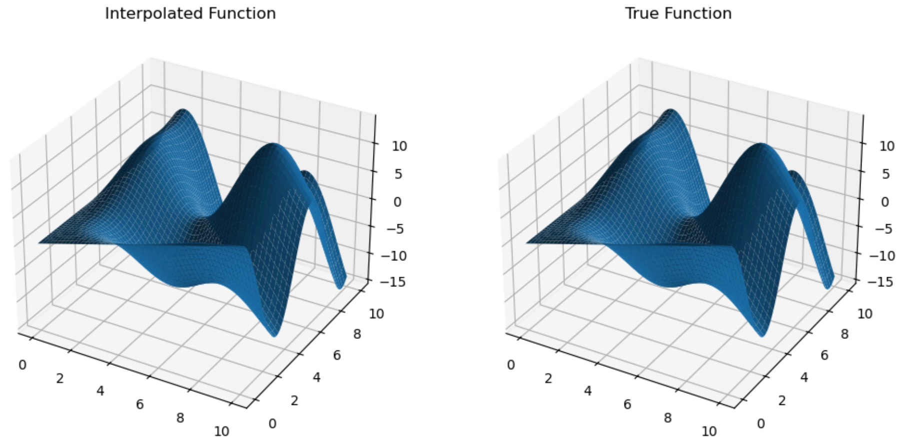
```
x_grid = np.geomspace(1, 11, 1000) - 1
y_grid = np.geomspace(1, 11, 1000) - 1
x_mat, y_mat = np.meshgrid(x_grid, y_grid, indexing="ij")

z_mat = trig_func(x_mat, y_mat)
```

Now, we generate a different grid, which will be used as our query points.

```
x_new, y_new = np.meshgrid(
 np.linspace(0, 10, 1000),
 np.linspace(0, 10, 1000),
    indexing="ij",
)
```

Then, we can compare our interpolation function with the analytical function and see that these are very close to each other.

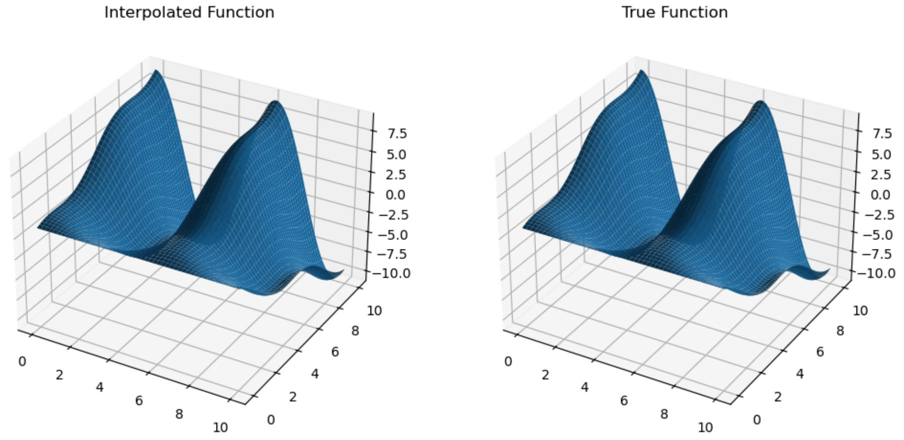
```
mult_interp = MultivariateInterp(z_mat, [x_grid, y_grid], backend="cupy")
z_mult_interp = mult_interp(x_new, y_new).get()
z_true = trig_func(x_new, y_new)
```



We can use the method `.diff(argnum)` of `MultivariateInterp` to evaluate numerical derivatives, which provides an object-oriented way to compute gradients. For example, calling `mult_interp.diff(0)` returns a `MultivariateInterp` object that represents the numerical derivative of the function with respect to the first argument on the same input grid.

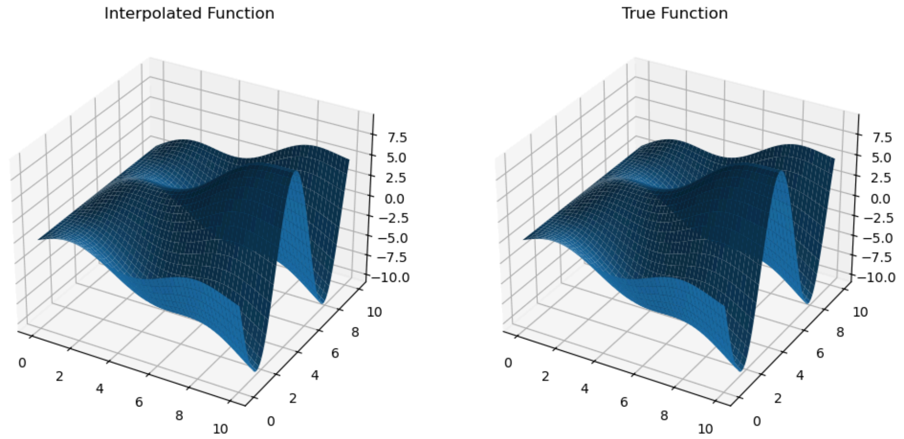
We can now compare the numerical and analytical derivatives and see that these are indeed very close to each other.

```
dfdx = mult_interp.diff(0)
z_dfdx = dfdx(x_new, y_new).get()
dfdx_true = trig_func_dx(x_new, y_new)
```



Similarly, we can compute the derivatives with respect to the second argument and see that they produce an accurate result.

```
dfdy = mult_interp.diff(1)
z_dfdy = dfdy(x_new, y_new).get()
dfdy_true = trig_func_dy(x_new, y_new)
```



The choice of returning object-oriented interpolation functions for the numerical derivatives is beneficial, as it allows for reusability without re-computation and easy chaining of operations. For example, we can compute the function's second derivative with respect to the first argument by calling `mult_interp.diff(0).diff(0)`.

### 4.4. *Multivalued Interpolation*

Finally, the `multinterp` package allows for multivalued interpolation on rectilinear grids via the `MultivaluedInterp` class.

Consider the following multivalued function:

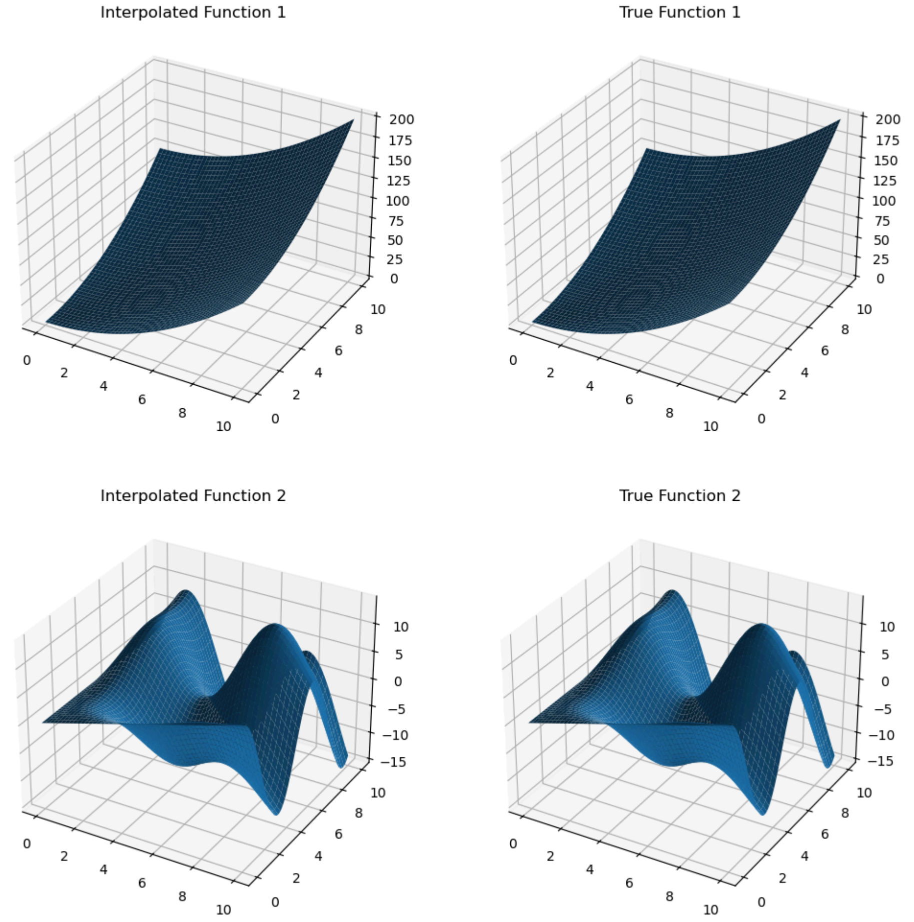
```
def squared_coords(x, y):
    return x**2 + y**2


def trig_func(x, y):
    return y * np.sin(x) + x * np.cos(y)


def multivalued_func(x, y):
    return np.array([squared_coords(x, y), trig_func(x, y)])
```

As before, we can generate values on a sample input grid and create a grid of query points.

```
x_grid = np.geomspace(1, 11, 1000) - 1
y_grid = np.geomspace(1, 11, 1000) - 1
x_mat, y_mat = np.meshgrid(x_grid, y_grid, indexing="ij")

z_mat = multivalued_func(x_mat, y_mat)

x_new, y_new = np.meshgrid(
 np.linspace(0, 10, 1000),
 np.linspace(0, 10, 1000),
    indexing="ij",
)
```

`MultivaluedInterp` can easily interpolate the function at the query points and avoid repeated calculations.

```
from multinterp.rectilinear._multi import MultivaluedInterp

mult_interp = MultivaluedInterp(z_mat, [x_grid, y_grid], backend="cupy")
z_mult_interp = mult_interp(x_new, y_new).get()
z_true = multivalued_func(x_new, y_new)
```



## 5. Curvilinear Interpolation

A *curvilinear* grid is a regular grid whose input coordinates are *curved* or *warped* in some regular way. However, simple transformations can nevertheless transform it into a regular grid. That is, every quadrangle in the grid can be transformed into a rectangle by remapping its vertices. There are two approaches to curvilinear interpolation in `multinterp`: the first requires a “point location” algorithm to determine which quadrangle the input point lies in, and the second requires a “dimensional reduction” algorithm to generate an interpolated value from the known values in the quadrangle.

Suppose we have a collection of values for an unknown function and their respective coordinate points. For illustration, assume the values come from the following function:

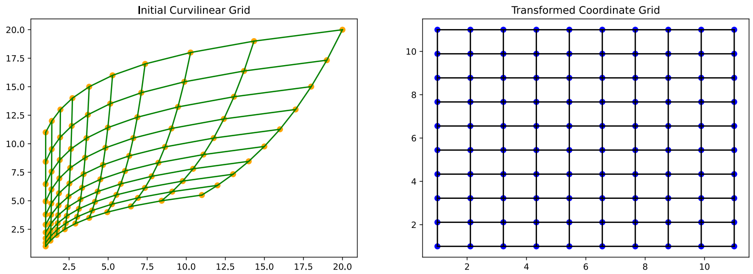


**Figure 2.** *A curvilinear grid can be transformed into a rectilinear grid by simply remapping its vertices.*

```
def function_1(x, y):
    return x * (1 - x) * np.cos(4 * np.pi * x) * np.sin(4 * np.pi * y**2) ** 2
```

The points are randomly scattered in the unit square and have no regular structure. This is achieved by randomly shifting a well-structured grid at every point.

```
rng = np.random.default_rng(0)
warp_factor = 0.01
x_list = np.linspace(0, 1, 20)
y_list = np.linspace(0, 1, 20)
x_temp, y_temp = np.meshgrid(x_list, y_list, indexing="ij")
rand_x = x_temp + warp_factor * (rng.random((x_list.size, y_list.size)) - 0.5)
rand_y = y_temp + warp_factor * (rng.random((x_list.size, y_list.size)) - 0.5)
values = function_1(rand_x, rand_y)
```

Suppose we want to interpolate this function on a rectilinear grid, a process called “regridding.”

```
grid_x, grid_y = np.meshgrid(
 np.linspace(0, 1, 100),
 np.linspace(0, 1, 100),
    indexing="ij",
)
```

We use `multinterp`’s `Warped2DInterp` and `Curvilinear2DInterp` classes to do this. The class takes the following arguments:

- `values`: an ND-array of values for the function at the points
- `grids`: a list of ND-arrays of coordinates for the points
- `backend`: the backend to use for interpolation currently only `scipy` and `numba` are supported for `Warped2DInterp`, and only `scipy` is supported for `Curvilinear2DInterp` for now.

```
from multinterp.curvilinear import Curvilinear2DInterp, Warped2DInterp

warped_interp = Warped2DInterp(values, (rand_x, rand_y), backend="numba")
warped_interp.warmup()
```

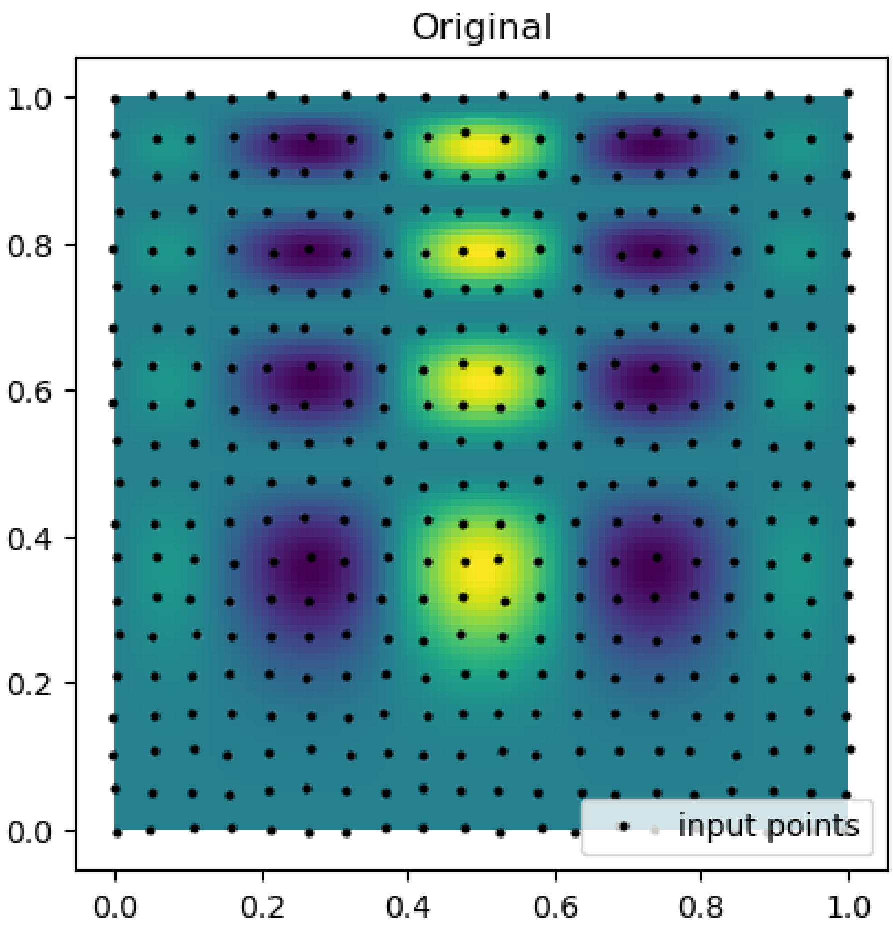
Once we create the interpolator objects, we can evaluate the functions on the query grids and compare their time performance.

```
start = time()
warped_grid = warped_interp(grid_x, grid_y)
print(f"Warped interpolation took {time() - start:.5f} seconds")
```

```
curvilinear_interp = Curvilinear2DInterp(values, (rand_x, rand_y))
start = time()
curvilinear_grid = curvilinear_interp(grid_x, grid_y)
print(f"Curvilinear interpolation took {time() - start:.5f} seconds")
```

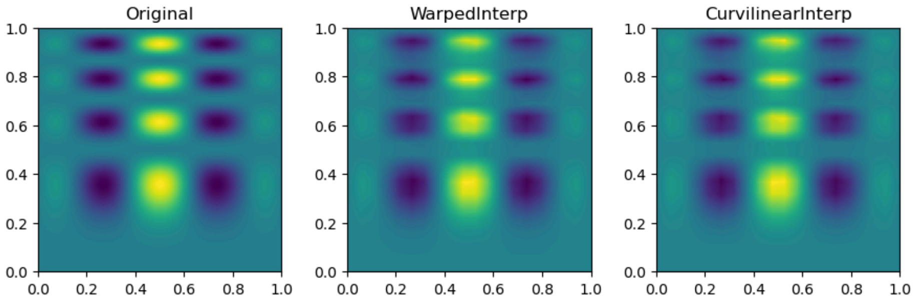
Now, we can compare the interpolation results with the original function. Below, we plot the original function and the known sample points. Notice that the points are almost rectilinear but have been randomly shifted to create a more challenging interpolation problem.

```
plt.imshow(function_1(grid_x, grid_y).T, extent=(0, 1, 0, 1), origin="lower")
plt.plot(rand_x.flat, rand_y.flat, "ok", ms=2, label="input points")
plt.title("Original")
plt.legend(loc="lower right")
```



Then, we can look at the result for each interpolation method and compare it to the original function.

```
fig, axs = plt.subplots(1, 3, figsize=(9, 6))
titles = ["Original", "WarpedInterp", "CurvilinearInterp"]
grids = [function_1(grid_x, grid_y), warped_grid, curvilinear_grid]

for ax, title, grid in zip(axs.flat, titles, grids):
    im = ax.imshow(grid.T, extent=(0, 1, 0, 1), origin="lower")
    ax.set_title(title)

plt.tight_layout()
plt.show()
```



In short, `multinterp`'s `Warped2DInterp` and `Curvilinear2DInterp` classes are helpful for interpolating functions on curvilinear grids with a quadrilateral structure but are not perfectly rectangular.

## 6. Unstructured Interpolation

Suppose we have a collection of values for an unknown function and their respective coordinate points. For illustration, assume the values come from the following function:

```
def function_1(u, v):
    return u * np.cos(u * v) + v * np.sin(u * v)
```

The points are randomly scattered within a square and have no regular structure.

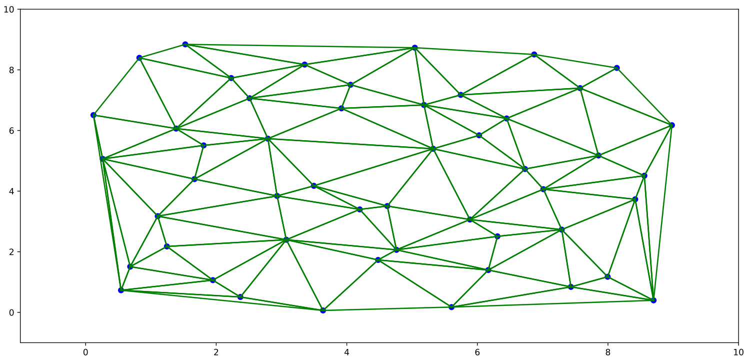


**Figure 3.** *Unstructured grids are irregular and often require a triangulation step, which might be computationally expensive and time-consuming.*

```
rng = np.random.default_rng(0)
rand_x, rand_y = rng.random((2, 1000)) * 3
values = function_1(rand_x, rand_y)
```

Suppose we want to interpolate this function on a rectilinear grid.

```
grid_x, grid_y = np.meshgrid(
 np.linspace(0, 3, 100),
 np.linspace(0, 3, 100),
    indexing="ij",
)
```

We use `multinterp`'s `UnstructuredInterp` class to do this. The class takes the following arguments:

- `values`: an ND-array of values for the function at the points
- `grids`: a list of ND-arrays of coordinates for the points
- `method`: the interpolation method to use, with options "nearest", "linear", "cubic" (for 2D only), and "rbf". The default is `'linear'`.

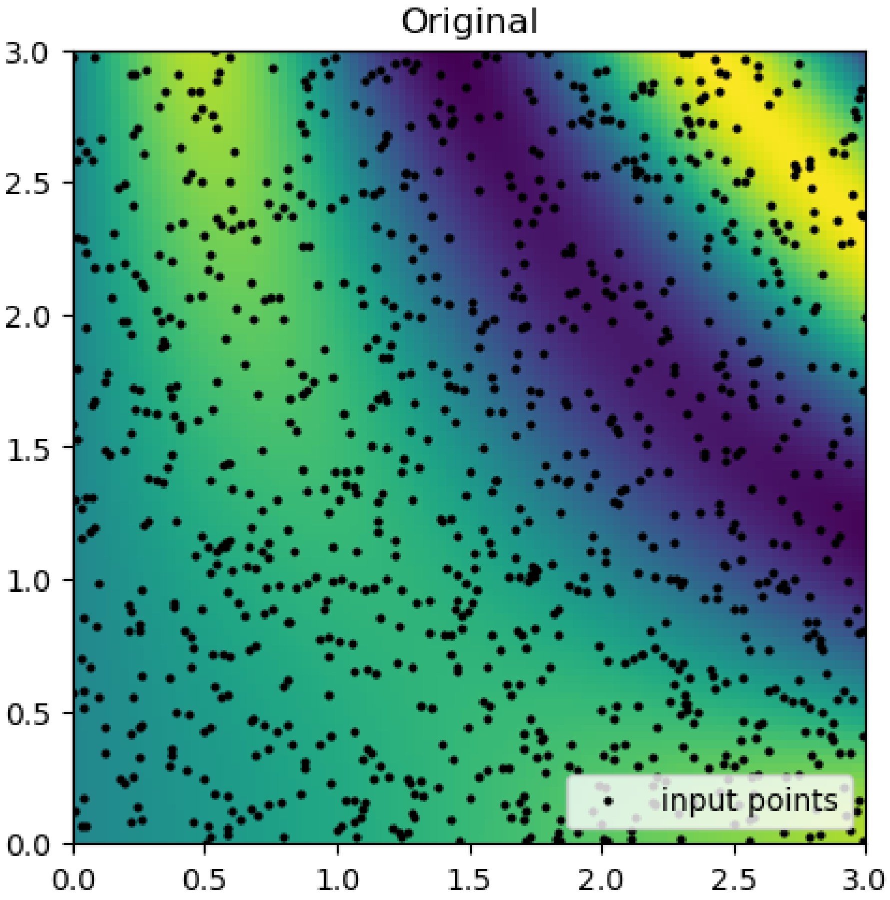
The `UnstructuredInterp` class is an object-oriented wrapper around `scipy.interpolate`'s functions for multivariate interpolation on unstructured data, which are `NearestNDInterpolator`, `LinearNDInterpolator`, `CloughTocher2DInterpolator`, and `RBFInterpolator`. The advantage of using `multinterp`'s `UnstructuredInterp` class is that it provides a consistent interface for all these methods, making it easier to switch between them and other interpolators in the `multinterp` package.

```
nearest_interp = UnstructuredInterp(values, (rand_x, rand_y), method="nearest")
linear_interp = UnstructuredInterp(values, (rand_x, rand_y), method="linear")
cubic_interp = UnstructuredInterp(values, (rand_x, rand_y), method="cubic")
rbf_interp = UnstructuredInterp(values, (rand_x, rand_y), method="rbf")
```

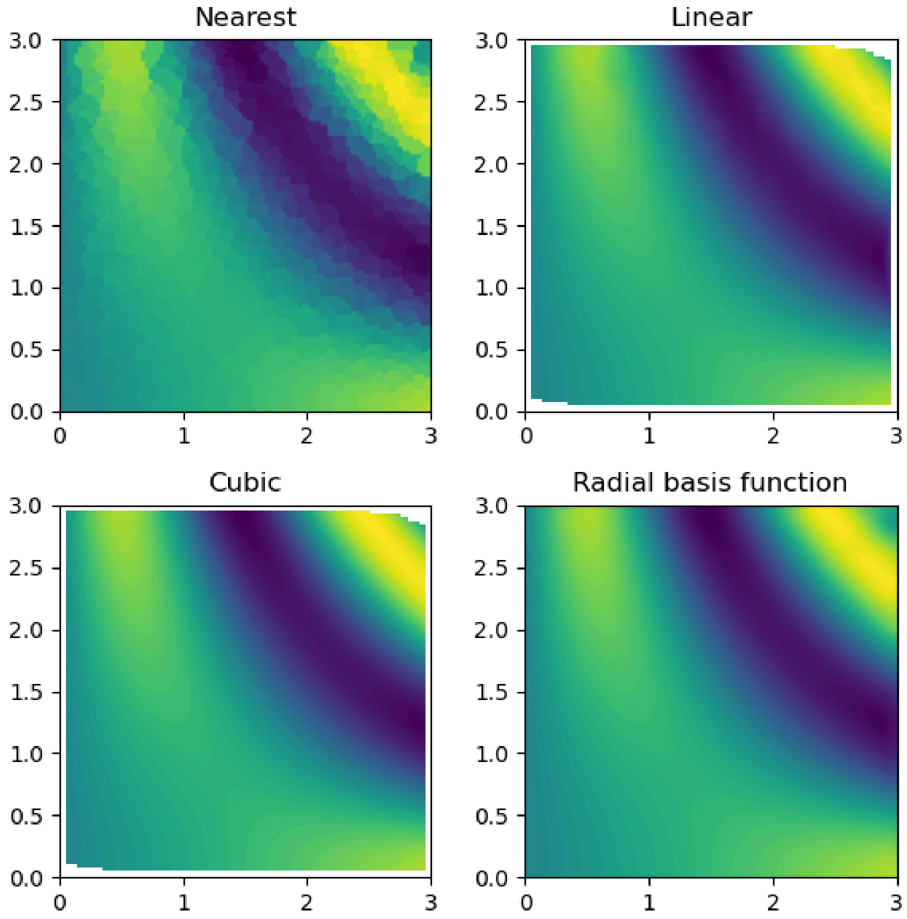
Once we create the interpolator objects, we can use them using the __call__ method, which takes as many arguments as there are dimensions.

```
nearest_grid = nearest_interp(grid_x, grid_y)
linear_grid = linear_interp(grid_x, grid_y)
cubic_grid = cubic_interp(grid_x, grid_y)
rbf_grid = rbf_interp(grid_x, grid_y)
```

Now, we can compare the interpolation results with the original function. Below, we plot the original function and the known sample points.

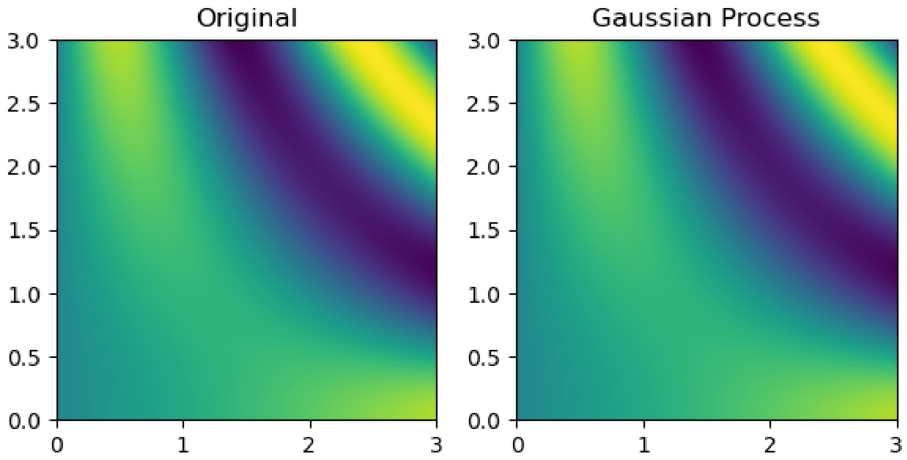


Then, we can look at the result for each interpolation method and compare it to the original function.



Finally, `multinterp` also provides a set of interpolators organized around the concept of *regression*. As a demonstration, below, we use a `RegressionUnstructuredInterp` interpolator, which uses a Gaussian Process regression model from `scikit-learn` [1] to interpolate the function defined on the unstructured grid. The `RegressionUnstructuredInterp` class takes the same arguments as the `UnstructuredInterp` class but requires the user to specify the regression `model` to use.

```
from multinterp import RegressionUnstructuredInterp

gaussian_interp = RegressionUnstructuredInterp(
 values,
 (rand_x, rand_y),
    model="gaussian-process",
    std=True,
)

gaussian_grid = gaussian_interp(grid_x, grid_y)
```



## 7. Performance Comparisons

We conducted a series of benchmarks to assess the performance of `multinterp` and compare it with existing tools. The benchmarks evaluated the interpolation speed and accuracy of `multinterp` across different grid sizes, dimensions, and backends.

### 7.1. *Benchmark Setup*

The benchmarks were conducted using the following setup:

- **Test Function**: A multivariate function with a known analytical solution generated the input data.
- **Grid Sizes**: The input data was generated on grids of varying sizes, ranging from 10x10 to 1000x1000.
- **Dimensions**: The benchmarks were conducted for both 2D and 3D grids.
- **Backends**: The benchmarks were run using different backends, including CPU (`numpy` and `scipy`), parallel (`numba`), and GPU (`cupy`, `pytorch`, and `jax`).

### 7.2. *Comparison with `scipy.interpolate.RegularGridInterpolator`*

The first set of benchmarks compares the performance of `multinterp` with `scipy.interpolate.RegularGridInterpolator`, a widely-used interpolation library in the scientific Python ecosystem. The benchmarks were conducted for both 2D and 3D grids, and the results are presented in the following figures.

### 7.3. Backend Comparison

The second set of benchmarks compares the performance of different backends in `multinterp`. The benchmarks were conducted for both 2D and 3D grids, and the results are presented in the following figures.

### 7.4. Discussion

The benchmark results demonstrate that `multinterp` outperforms `scipy.interpolate.RegularGridInterpolator` for 2D and 3D grids, especially for larger grid sizes. This is likely due to the efficient implementation of `multinterp`'s core functions, such as `map_coordinates` and `get_coordinates`.

The backend comparison shows that GPU backends (`cupy`, `pytorch`, and `jax`) can provide significant performance improvements for large grid sizes, especially in 3D. However, data transfer overhead between the CPU and GPU can make GPU backends slower for smaller grid sizes. Parallel backends (`numba`) can also provide performance improvements for CPU-bound tasks, but the benefits are less pronounced than those of GPU backends.

Overall, the benchmark results highlight the advantages of `multinterp` regarding performance and flexibility. The package offers a unified interface for various types of interpolation, supports multiple backends, and includes tools for multivalued interpolation and interpolation of derivatives.

## 8. Future Work and Contributions

The `multinterp` package is currently in its beta stage, and there are several areas where future development and community contributions would be valuable.

### 8.1. Planned Features

1. **Additional Interpolation Methods**: Implement support for additional interpolation methods, such as spline, polynomial, inverse distance weight, and radial basis function interpolation. This will give users more options to handle various types of data and interpolation requirements.
2. **Optimization and Performance Improvements**: Continue to optimize the core functions of `multinterp` to improve performance and efficiency, particularly for large datasets and high-dimensional interpolation.
3. **Documentation and Examples**: Expand the documentation and provide more examples to demonstrate the capabilities of `multinterp`, including comparisons between different interpolation methods and grid types.
4. **Extended Grid Support**: Enhance support for complex grid types and improve interpolation efficiency on unstructured grids.

### 8.2. Community Contributions

We welcome contributions from the community to help improve the `multinterp` package. There are several ways to contribute:

1. **Code Contributions**: Submit pull requests to add new features, fix bugs, or improve the documentation.
2. **Issue Reporting**: Report any issues or bugs you encounter using `multinterp`.
3. **Testing**: Help expand the test suite to ensure the package works correctly across various scenarios.
4. **Documentation**: Contribute to the documentation by writing tutorials, examples, or improving the existing documentation.

To get started with contributing, please refer to our Contribution Guidelines for detailed information on our development process, coding standards, and how to submit pull requests.

### *8.3. Documentation and Resources*

For more information about `multinterp`, please refer to the following resources:

- **Source Code**: The source code is hosted on GitHub at https://github.com/alanlujan91/multinterp
- **Issue Tracker**: Report bugs or request features on our GitHub Issues page
- **Contribution Guidelines**: Learn how to contribute at CONTRIBUTING.md
- **Examples**: A collection of Jupyter notebooks with usage examples can be found in the examples directory of our GitHub repository

We encourage users to explore these resources and contact the community through our GitHub Discussions for any questions or feedback.

## 9. Conclusion

Multivariate interpolation is a cornerstone of scientific computing, yet the Python ecosystem [2] presents a fragmented landscape of tools. While individually powerful, these packages often lack a unified interface. This fragmentation makes it difficult for researchers to experiment with different interpolation methods, optimize performance across diverse hardware, and handle varying data structures (regular, rectilinear, curvilinear, unstructured).

The `multinterp` project seeks to change this. Its goal is to provide a unified, comprehensive, and flexible framework for multivariate interpolation in python. This framework will streamline workflows by offering:

- Unified Interface: A consistent API for interpolation, regardless of data structure or desired backend, reducing the learning curve and promoting code reusability.
- Hardware Adaptability: Seamless support for CPU (`numpy` C. R. Harris *et al.* [3], `scipy` P. Virtanen *et al.* [4]), parallel (`numba` S. K. Lam, A. Pitrou, and S. Seibert [5]), and GPU (`cupy` R. Okuta, Y. Unno, D. Nishino, S. Hido, and C. Loomis [6], `pytorch` A. Paszke *et al.* [7], `jax` J. Bradbury *et al.* [8]) backends, empowering users to optimize performance based on their computational resources.
- Broad Functionality: Tools for regular/rectilinear interpolation, multivalued interpolation, and derivative calculations, addressing a wide range of scientific problems.

The multinterp package (https://github.com/alanlujan91/multinterp) is currently in its beta stage. It offers a strong foundation but welcomes community contributions to reach its full potential. We invite collaboration to improve documentation, expand the test suite, and ensure the codebase aligns with the highest standards of Python package development.

## References

[1] F. Pedregosa *et al.*, “Scikit-learn: Machine Learning in Python,” *Journal of machine learning research: JMLR*, pp. 2825–2830, Feb. 2011.

[2] T. E. Oliphant, “Python for Scientific Computing,” *Computing in science & engineering*, vol. 9, no. 3, pp. 10–20, 2007, doi: 10.1109/MCSE.2007.58.

[3] C. R. Harris *et al.*, “Array programming with NumPy,” *Nature*, vol. 585, no. 7825, pp. 357–362, Sep. 2020, doi: 10.1038/s41586-020-2649-2.

[4] P. Virtanen *et al.*, “SciPy 1.0: fundamental algorithms for scientific computing in Python,” *Nature methods*, vol. 17, no. 3, pp. 261–272, 2020, doi: 10.1038/s41592-019-0686-2.

[5] 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*, in LLVM '15. New York, NY, USA, Nov. 2015, pp. 1–6. doi: 10.1145/2833157.2833162.

[6] R. Okuta, Y. Unno, D. Nishino, S. Hido, and C. Loomis, “CuPy: A NumPy-Compatible Library for NVIDIA GPU Calculations.” 2017.

[7] A. Paszke *et al.*, “PyTorch: An imperative style, high-performance deep learning library,” *arXiv [cs. LG]*, pp. 8026–8037, Dec. 2019.

[8] J. Bradbury *et al.*, “JAX: composable transformations of Python+NumPy programs.” 2018.

Open Access | https://doi.org/10.25080/JHPV7385

# Mandala: Compositional Memoization for Simple & Powerful Scientific Data Management

**Aleksandar Makelov**[1,2,3]

[1]MIT, [2]SERI MATS, [3]Independent Researcher

SciPy 2024

July 8 - July 14, 2024

Proceedings of the 23nd Python in Science Conference

ISSN: 2575-9752

**Published** Jul 10, 2024

**Correspondence to**
Aleksandar Makelov
aleksandar.makelov@gmail.com

## Abstract

We present `mandala`, a Python library that largely eliminates the accidental complexity of scientific data management and incremental computing. While most traditional and/or popular data management solutions are based on *logging*, `mandala` takes a fundamentally different approach, using *memoization* of function calls as the fundamental unit of saving, loading, querying and deleting computational artifacts.

It does so by implementing a *compositional* form of memoization, which keeps track of how memoized functions compose with one another. In this way: (1) complex computations are effectively memoized end-to-end, and become 'interfaces' to their own intermediate results by retracing the memoized calls; (2) all computations in a project form a single computational graph, which can be explored, queried and manipulated in high-level ways through a *computation frame*, which is a natural generalization of a dataframe that replaces columns by a computation graph, and rows by (partial) executions of this graph.

Several features implemented on top of the core memoization data structures — such as natively and transparently handling Python collections, in-memory caching of intermediate results, and a flexible versioning system with dynamic dependency tracking — turn `mandala` into a practical and simple tool for managing and interacting with computational data.

**Keywords** scientific data management, machine learning

## 1. Introduction

Numerical experiments and simulations are growing into a central part of many areas of science and engineering [1]. Recent trends in computation-intensive fields, such as machine learning, point towards (1) ever-increasing complexity of computational pipelines, and (2) adoption in more safety-critical domains, such as autonomous driving [2] and healthcare [3], [4].

These developments impose opposing constraints on the tools used to manage the resulting computational artifacts. On the one hand, they should be simple and easy to use by researchers, with a minimal learning curve and unobtrusive syntax and semantics. On the other hand, they should deliver a lot of added functionality, such as high-level operations [5], full data & code provenance auditing [6] and reproducibility [7] in complex projects. Rules and best practices that help with these requirements exist and are well-known [8], [9], but still require manual effort, attention to extraneous details, and discipline to follow. Researchers often operate under time pressure and/or the need to quickly iterate on code, which makes these best 'practices' a rather *impractical* time investment.

Thus, ideally we would like a system that (1) does not get in the way by imposing a complex new language/semantics/syntax, (2) provides powerful high-level data management operations over complex computational projects, and (3) incorporates best practices by design and without cognitive overhead.

```
# decorate any
# Python funcs
@op
def f(x):
  return x**2

@op
def g(x, y):
  return x + y

...
```

(a)

```
storage = Storage()

# memoizing context
with storage:
  for x in range(3):
    y = f(x)

In [1]: y # wrapped value
Out[1]: AtomRef(4,
          hid='628...',
          cid='a82...')
```

(b)

```
with storage:
  # just add more calls
  # & reuse old results
  for x in range(5):
    y = f(x)
    # unwrap for control flow
    if storage.unwrap(y) > 5:
      z = g(x, y)

# the "program" is now end-
to-end
# memoized & retraceable
```

(c)

**Figure 1**. *Basic imperative usage of* `mandala`*.* ***(a)****: add the* `@op` *decorator to any Python functions to make them memoizable.* ***(b)****: create a* `Storage`*, and use it as a context manager to automatically memoize any calls to* `@op`*-decorated functions in the block. Memoized functions return* `Ref` *objects, which wrap a value with two pieces of metadata: a* content ID*, which is a hash of the value of the object, and a* history ID*, which is a hash of the identity of the* `@op` *that produced the* `Ref` *(if any), and the history IDs of the* `@op`*'s inputs.* ***(c)****: the storage context allows simple incremental computation and recovery from failures. Here, we add more computations while automatically reusing already computed values.*

In this paper we present `mandala`, our proposal for such a system. It **integrates data management logic and best practices** such as

- Full data provenance tracking
- Idempotent & reproducible computation
- Content-addressable versioning of code and its dependencies
- Declarative high-level manipulation of persisted computational graphs

**into Python's already familiar syntax and semantics** (Figures Figure 1 and Figure 2). The integration aims to be maximally transparent and unobtrusive, so that the user can focus on the *essential complexity* (the scientific problem at hand), rather than on the *accidental complexity* (the data management tools necessary to implement the solution) [10].

The rest of this paper presents the design and main functionalities of `mandala`, and is organized as follows:

- In Section 2, we describe how memoization is designed, how this allows memoized calls to be composed and memoized results to be reused without storage duplication, and how this enables the *retracing* pattern of interacting with computational artifacts.
- In Section 3, we introduce the concept of a *computation frame*, which generalizes a dataframe by replacing columns with a computational graph, and rows with individual computations that (partially) follow this graph. Computation frames allow high-level exploration and manipulation of the stored computation graph, such as adding the calls that produced/used given values to the graph, deleting all computations that depend on the calls captured in the frame, and restricting the frame to a particular subgraph or subset of values with given properties.
- In Section 4, we describe some other features of `mandala` necessary to make it a practical tool, such as:
    - Representing Python collections in a way transparent to the storage, so that the membership relationships between a collection and its items are propagated through the saved computational graph;
    - Caching of intermediate results to speed up retracing and memoization;
    - A flexible versioning system with automatic dynamic dependency tracking.

Finally, we give an overview of related work in Section 5.

## 2. Core Concepts

### *2.1. Memoization and the Computational Graph*

Memoization is a technique that stores the results of expensive function calls to avoid redundant computation. `mandala` uses *automatic* memoization [11] which is applied via the combination of a decorator (`@op`) and a context manager which specifies the `Storage` object to use (Figure 1). The memoization can optionally be made persistent to disk, which is what you would typically want in a long-running project. Any Python function can be memoized (as long as its inputs and outputs are serializable by the `joblib` library; see the limitations section Section 6 for caveats); there is no restriction on the type, number or naming scheme (positional, keyword, variadic, or variable keyword) of the arguments or return values.

#### 2.1.1. *`Call` and `Ref` objects, and content/history IDs.:*

`Ref`s and `Call`s are the two atomic data structures in `mandala`'s model of computations. When a call to an `@op`-decorated function `f` is executed inside a storage context, this results in the creation of

- A `Ref` object for each input to the call. These wrap the 'raw' values passed as inputs together with content IDs (hashes of the Python objects) and history IDs (hashes of the memoized calls that produced these values, if any).
  - If an input to the call is already a `Ref` object, it is passed through as is;
  - If it is a 'raw' (i.e., non-`Ref`) value, a new `Ref` object is created with a 'empty' history ID that is simply a hash of the content ID itself.
- A `Call` object, which has pointers to the input and output (defined below) `Ref`s of the call to `f`, as well as a content ID for the call (a hash of the identity of `f` and the content IDs of the input `Ref`s) and a history ID (by analogy, a hash of the identity of `f` and the history IDs of the inputs). The version of `f` is also part of the identity of `f`; see the versioning section Section 4.2 for details.
- A `Ref` object for each return value of the call. These are again assigned content IDs by value, and history IDs by hashing the tuple (history ID of the call, corresponding output name[1]).

The `Ref`s and the `Call` are then stored in the storage backend, and the next time `f` is called on inputs that have the same *content* IDs, the stored `Call` is looked up to find the output `Ref`s, which are then returned (possibly with properly updated history IDs, if the call exists in storage by content ID only). The combination of all stored `Call`s and `Ref`s across memoized functions form the **computational graph** represented by the storage. Importantly, the 'interesting' structure of this graph is built up automatically by the way the user composes memoized calls.

### *2.2. Motivation for the Design of Memoization*

#### 2.2.1. *Why content and history IDs?:*

The simultaneous use of content and history IDs has a few subtle advantages. First, it allows for the *de-duplication* of storage, as the same content ID can be used to store the same value produced by different computations. For instance, there may be many computaitions all producing the value 42 (or a large all-zero array), but only one copy of 42 is stored in the

[1] Since Python functions don't have designated output names, we instead generate output names automatically using the order in the tuple of return values.

backend. At the same time, the history IDs allow us to distinguish between computations that produced the same value, but in different ways. This avoids ‘parasitic’ results in declarative queries. For example, a call to `f` may result in 42, and we may be interested in all calls that were ran on this particular 42 returned by `f` and not on any other `Ref` whose value happens to be 42. Without history IDs, it would be impossible to make this distinction in the stored computational graph.

### 2.2.2. *Why memoization?:*

Memoization is an unusual choice for data management systems, most of which are based on *logging*, i.e. explicitly pointing to the value to be saved and the address where it should be saved (whether this is some kind of name or a file path). Basing data management on memoization means that the ‘address’ of a value is now implicit in the code that produced it, and the value itself is stored in a shared storage backend. This has several advantages:

- It **eliminates the need to manually name artifacts**. This eliminates a major source of accidental complexity: names are arbitrary, ambiguous, and can drift away from the actual content of the value they point to over time. On the other hand, names are not strictly necessary, because the composition of memoized functions that produced a given value — which must be specified anyway for the computation to take place — is already an unambiguous pointer to it.
- Since the `@op` decorator encourages (and in a sense enforces) composition of `@op`s, it **automatically builds up a computational graph of the project**. Most data management tasks — e.g., a frequent use case is getting a table of relationships between some variables — are naturally expressed as queries over this graph, as we will see in Section 3.
- It **organizes storage functionality around a familiar and flexible interface: the function call**. This automatically enforces the good practice of partitioning code into functions, and eliminates extra ‘accidental’ code to save and load values explicitly. Furthermore, it synchronizes failures between computation and storage, as the memoized calls are the natural points to recover from.

On the other hand, memoization suffers from the following limitations:

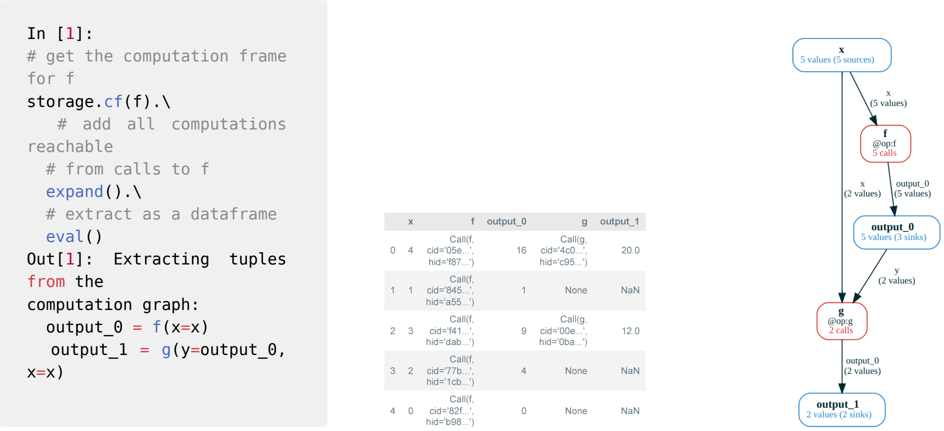
- Referring to values without reference to the code that produced or used them becomes difficult, because from the point of view of storage the ‘identity’ of a value is its place in the computational graph. We discuss practical ways to overcome this in Section 3.
- Modifying `@op` functions requires care, as changes may invalidate the stored computational graph. We discuss a versioning system that automates this process in Section 4.2.

## *2.3. Retracing as a Versatile Imperative Interface to the Stored Computation Graph*

The compositional nature of memoization makes it possible to build complex computations out of calls to memoized functions, turning the entire computation into an end-to-end-memoized interface to its own intermediate results. The main way to interact with such a persisted computation is through **retracing**, which means stepping through memoized code with the purpose of resuming from a failure, loading intermediate values, or continuing from a particular point with new computations. A small example of retracing is shown in Figure 1 (c).

This pattern is simple yet powerful, as it allows the user to interact with the stored computation graph in a way that is adapted to their use case, and to explore the graph in a way that is natural and familiar to them. It also simplifies the management of state in an interactive environment such as a Jupyter notebook, because it makes it very cheap to re-run cells in order to recreate the intended state of local variables.

# 3. Computation Frames



```
In [1]:
# get the computation frame
for f
storage.cf(f).\
   # add all computations
reachable
  # from calls to f
  expand().\
  # extract as a dataframe
  eval()
Out[1]: Extracting tuples
from the
computation graph:
  output_0 = f(x=x)
   output_1 = g(y=output_0,
x=x)
```

(a)

| | x | f | output_0 | g | output_1 |
|---|---|---|---|---|---|
| 0 | 4 | Call(f, cid='05e...', hid='f87...') | 16 | Call(g, cid='4c0...', hid='c95...') | 20.0 |
| 1 | 1 | Call(f, cid='845...', hid='a55...') | 1 | None | NaN |
| 2 | 3 | Call(f, cid='f41...', hid='dab...') | 9 | Call(g, cid='00e...', hid='0ba...') | 12.0 |
| 3 | 2 | Call(f, cid='77b...', hid='1cb...') | 4 | None | NaN |
| 4 | 0 | Call(f, cid='82f...', hid='b98...') | 0 | None | NaN |

(b)

(c)

**Figure 2.** *Basic declarative usage of* `mandala` *and an example of computation frames.*

In order to be able to explore and manipulate the full stored computation graph, patterns like retracing are insufficient, because they require the complete code producing part of the graph to be available. To complement retracing, we introduce the `ComputationFrame` data structure, which is a high-level declarative interface to the stored computation graph.

## 3.1. *Motivation and Intuition*

Intuitively, a computation frame is a way to organize a collection of saved `@op` calls into groups, where the calls in each group have an analogous role in the computation, and the groups form a high-level computational graph of variables (which represent groups of `Ref`s) and functions (groups of `Call`s). The illustration in Figure 2 (c) shows a visualization of a computation frame extracted from the computations in Figure 1.

This kind of organization is useful because it reflects how the user thinks about the computation, and allows them to tailor the exploration of the computation graph to a particular use case, much like a database view. For instance, sometimes it makes sense to group the outputs of several different `@op`s into a single variable because they are treated the same way by downstream computations.

From another point of view, computation frames are 'views' of the stored computation graph, analogous to database views. In particular, they may contain multiple references to the same `Ref` or `Call` object from different nodes of the graph, and do not necessarily contain all calls to a given `@op`.

## 3.2. *Formal Definition*

A computation frame (Figure 2) consists of the following data:

- **Computation graph**: a directed graph $G = (V, F, E)$ where $V$ are named variables and $F$ are named instances of `@op`-decorated functions. The edges $E$ are labeled with the input/output names of the adjacent functions. An example is shown in Figure 2 (c);
- **Groups of `Ref`s and `Call`s**: for each variable $v \in V$, a set of (history IDs of) `Ref`s $R_v$, and for each function $f \in F$ with underlying `@op` $o_f$, a set of (history IDs of) `Call`s $C_f$;

subject to the constraint that: for every call $c \in C_f$, if there's an input/output edge labeled $l$ connecting $f$ to some variable $v$, then if $c$ has a `Ref` $r_l$ corresponding to input/output name $l$, we have $r_l \in R_v$.

In other words, when we look at all calls in $f \in F$, their inputs/outputs must be present in the variables connected to $f$ under the respective input/output name.

### *3.3. Basic Usage*

The main advantage of computation frames is that they allow iterative exploration of the computation graph, and high-level grouped operations over computations with some shared structure. For example, we can use them for:

- **Iteratively expanding the frame with functions that generated or used existing variables**: this is useful for exploring the computation graph in a particular direction, or for adding more context to a particular computation. For example, in Figure 2 (a), we start with a computation frame containing only the calls to `f`, and then expand it to include all calls that can be reached from the memoized calls to `f` via their inputs/outputs, which adds the calls to `g` to the frame.
- **Converting the frame into a dataframe**: this is useful at the end of an exploration, when we want to get a convenient tabular representation of the captured computation graph. The table is obtained by collecting all terminal `Ref`s in the frame's computational graph (i.e., those that are not inputs to any function in the frame), computing their computational history in the frame (grouped by variable), and joining the resulting tables over the variables. This is shown in Figure 2 (right). In particular, as shown in the example, this step may produce nulls, as the computation frame can contain computations that only partially follow the graph.
- **Performing high-level storage manipulations**: such as deleting all calls captured in the frame as well as all calls that depend on them, available using the `.delete_calls()` method on the frame.

Computation frames are a powerful tool for exploring and manipulating the stored computation graph, and we're excited to explore their full potential in future work.

## 4. Main Extra Features

### *4.1. Data Structures*

Python's native collections — lists, dicts, sets — can be memoized transparently by `mandala`, using customized type annotations, e.g. `MList[int]` inheriting from `List[int]`, …. By applying this type annotation, individual elements as well as the collection itself are memoized as `Ref`s (with the collection merely pointing to the `Ref`s of its elements to avoid duplication).

This is implemented fully on top of the core memoization machinery, using 'internal' `@op`s like e.g. `__make_list__` which, given the elements of a list as variadic inputs, generates a `ListRef` (subclass of `Ref`) that points to the `Ref`s of the elements. In this way, collections are

```python
@op
def avg_items(xs: MList[int]) -> float:
  return sum(xs) / len(xs)

@op
def get_xs(n) -> MList[int]:
  return list(range(n))

with storage:
  xs = get_xs(10)
  for i in range(2, 10, 2):
    avg = avg_items(xs[:i])
```

**Figure 3**. *Illustration of native collection memoization in `mandala`. The custom type annotation `MList[int]` is used to memoize a list of integers as a list of pointers to element `Ref`s.*

naturally incorporated in the computation graph. These internal `@ops` are applied automatically when a collection is passed as an argument to a memoized function, or when a collection is returned from a memoized function (Figure 3).

### *4.2. Versioning*

It is crucial to have a flexible and powerful code versioning system in a data management tool, as it allows the user to keep track of the evolution of their computations, and to easily recover from mistakes. `mandala` provides the option to use a versioning system with three main features:

- **Per-function content-addressed versioning** [12], [13], where the version of a function is a hash of its source code and the hashes of the functions it calls. The storage can determine, based only on the state of the codebase, whether a given call is up-to-date or not.
- **Dynamic dependency tracking**, where each function call traces the dependencies it calls. This avoids the need for static analysis of the code to find dependencies, which can result in many false positives and negatives, especially in a dynamic language like Python. Moreover, it provides a stronger notion of reusability, as certain changes of the codebase may invalidate only a part of all memoized calls to an `@op`.
- **The flexibility to mark changes as backward-compatible or not**, which allows the user to maintain a stable interface to computations when performing routine refactoring or adding logging/debugging code.

## 5. Related Work

`mandala` combines ideas from several existing projects, but is unique in the Pythonic way it makes complex memoized computations easy to query, manipulate and version.

**Memoization.** There are several memoization solutions for Python that lack the compositional nature of `mandala`, as well as the versioning and querying tools: the builtin `functools` module provides decorators such as `lru_cache` for memoization; the `incpy` project [14] enables automatic persistent memoization of Python functions directly on the interpreter level; the `funsies` project [15] is a memoization-based distributed workflow executor that uses a similar hashing approach to keep track of which computations have already been done; `koji` [16] is a design for an incremental computation data processing framework that unifies over different resource types (files or services), and uses an analogous notion of hashing to keep track of computations.

**Computation Frames.** Computation frames are closely related to the relational model [17], to graph databases, and to certain versatile in-memory data structures based on functors $\mathcal{F} : \mathcal{C} \to \underline{\boldsymbol{Set}}$ where $\mathcal{C}$ is a finite category [18].

**Versioning.** The `unison` programming language [13] uses a content-addressed system for code storage, where a function is identified by the hash of its syntax tree. The language shares many other features and goals with `mandala`, such as use of serializable values and pure functions to ensure reproducibility.

The dynamic tracing mechanism used to capture dependencies is somewhat similar to the `@tf.function` decorator in the TensorFlow library [19], which traces the function calls to other `@tf.function`-decorated functions made during execution and builds a computation graph out of them. Unlike `@tf.function`, `mandala`'s versioner uses content (code) hashes to automatically detect changes in dependencies, and does not build a fine-grained model of a function's execution, but rather only tracks the set of its dependencies.

The revision history of each function in the codebase is organized in a bare-bones `git` repository [12]: it is a content-addressed tree, where each edge tracks a diff from the content at one endpoint to that at the other. Additional metadata indicates equivalence classes of semantically equivalent contents.

## 6. Limitations

**Computing deterministic content IDs of any Python object is difficult.** `mandala` uses the `joblib` library to serialize Python objects into byte strings, and then hashes these strings to get the content ID. This approach is not perfect, as it is not always possible to serialize Python objects, and even when it is, the serialization may not be unique. For example, two Python objects `x, y` which satisfy `x == y` may not have the same content ID (e.g., `True` and `1`). Furthermore, hashing is sensitive to small changes in the input, such as numerical precision in floating point numbers. Finally, complex Python objects may contain state that is not intrinsically part of the object's identity, such as resource utilization data (e.g., memory addresses). This can lead to different content IDs before and after a round trip through the storage backend. These issues don't come up often as long as all initial `Ref`s are created from simple Python objects: complex objects are hashed and saved once when returned from an `@op`, and then referred to by their content ID.

**Non-breaking versioning is difficult.** The ability to mark code changes as backward-compatible or not may lead to situations where the storage 'believes' that a call is up-to-date, but in reality it is not. For example, a function `f` can be changed by extracting a subroutine `g` out of it. The semantics of `f` is unchanged, so past calls are still valid, but `g` is now an invisible (to the storage) dependency of `f`. Care should be taken to avoid such situations until an automatic solution is implemented.

## 7. Conclusion

`mandala` is being actively developed, and has the potential to considerably simplify the way scientific data is managed and interacted with in Python. The author has already used it extensively to manage several multi-month machine learning projects, and has found it to be a very powerful tool for managing complex computations. We hope that this paper has given a good overview of the core concepts of `mandala`, and that the reader will be interested in exploring the library further.

### Acknowledgements

First and foremost, I would like to thank my friend Stefan Krastanov for many valuable conversations throughout the evolution and development of `mandala`. Nobody could ask for a more enthusiastic collaborator and champion of their work. Second, I would also like to thank Nicholas Schiefer for some helpful feedback on an earlier version of the library, as well as suggestions and implementations for features to make it work in a distributed setting, and for advertising `mandala` at his workplace.

There have been far too many people over the years who have listened patiently to me talk about this project in its earlier stages; in particular, I'm grateful to Petar Maymounkov from Protocol Labs for inviting me to give a talk about `mandala` at their research seminar, as well as for creating the `escher` language, which truly changed the way I think about programming. I'm also grateful to many friends from grad school and beyond who have listened to me talk about this project, and who have given me valuable feedback and encouragement. Finally, I would like to thank my reviewers at the SciPy conference, especially Andrei Paleyes, for their helpful feedback on this paper.

## References

[1] T. Hey, S. Tansley, K. M. Tolle, and others, *The fourth paradigm: data-intensive scientific discovery*, vol. 1. Microsoft research Redmond, WA, 2009. doi: 10.1007/978-3-642-33299-9_1.

[2] M. Bojarski *et al.*, "End to end learning for self-driving cars," *arXiv preprint arXiv:1604.07316*, 2016.

[3] D. Ravı *et al.*, "Deep learning for health informatics," *IEEE journal of biomedical and health informatics*, vol. 21, no. 1, pp. 4–21, 2016, doi: 10.1109/JBHI.2016.2636665.

[4] J. Abramson *et al.*, "Accurate structure prediction of biomolecular interactions with AlphaFold 3," *Nature*, pp. 1–3, 2024, doi: 10.1038/s41586-024-07487-w.

[5] H. Wickham, "Tidy data," *Journal of statistical software*, vol. 59, pp. 1–23, 2014, doi: 10.18637/jss.v059.i10.

[6] S. B. Davidson and J. Freire, "Provenance and scientific workflows: challenges and opportunities," in *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, 2008, pp. 1345–1350. doi: 10.1145/1376616.1376772.

[7] P. Ivie and D. Thain, "Reproducibility in scientific computing," *ACM Computing Surveys (CSUR)*, vol. 51, no. 3, pp. 1–36, 2018, doi: 10.1145/3186266.

[8] G. K. Sandve, A. Nekrutenko, J. Taylor, and E. Hovig, "Ten simple rules for reproducible computational research," *PLoS computational biology*, vol. 9, no. 10, p. e1003285, 2013, doi: 10.1371/journal.pcbi.1003285.

[9] M. D. Wilkinson *et al.*, "The FAIR Guiding Principles for scientific data management and stewardship," *Scientific data*, vol. 3, no. 1, pp. 1–9, 2016, doi: 10.1038/sdata.2016.18.

[10] F. P. Brooks, "No Silver Bullet: Essence and Accidents of Software Engineering," 1987. doi: 10.1109/MC.1987.1663532.

[11] P. Norvig, "Techniques for automatic memoization with applications to context-free parsing," *Computational Linguistics*, vol. 17, no. 1, pp. 91–98, 1991.

[12] L. Torvalds and others, "Git," 2005. [Online]. Available: https://git-scm.com/

[13] R. C. Lozano, "The Unison manual." 2017.

[14] P. J. Guo and D. Engler, "Using automatic persistent memoization to facilitate data analysis scripting," in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, 2011, pp. 287–297. doi: 10.1145/2001420.2001455.

[15] C. Lavigne and A. Aspuru-Guzik, "funsies: A minimalist, distributed and dynamic workflow engine," *Journal of Open Source Software*, vol. 6, no. 66, p. 3274, 2021, doi: 10.21105/joss.03274.

[16] P. Maymounkov, "Koji: Automating pipelines with mixed-semantics data sources," *arXiv preprint arXiv:1901.01908*, 2018.

[17] E. F. Codd, "A relational model of data for large shared data banks," *Communications of the ACM*, vol. 13, no. 6, pp. 377–387, 1970, doi: 10.1145/362384.362685.

[18] E. Patterson, O. Lynch, and J. Fairbanks, "Categorical data structures for technical computing," *Compositionality*, vol. 4, 2022, doi: 10.32408/compositionality-4-5.

[19] M. Abadi *et al.*, "TensorFlow: a system for Large-Scale machine learning," in *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, 2016, pp. 265–283. doi: 10.5281/zenodo.4724125.

Open Access | https://doi.org/10.25080/HXCJ6205

# Training a Supervised Cilia Segmentation Model from Self-Supervision

**Seyed Alireza Vaezi**[1], and **Shannon Quinn**[1]

[1] University of Georgia

SciPy 2024

July 8 - July 14, 2024

Proceedings of the 23rd
Python in Science Conference
ISSN: 2575-9752

**Published** Jul 10, 2024

**Correspondence to**
Seyed Alireza Vaezi
sv22900@uga.edu

## Abstract

Cilia are organelles found on the surface of some cells in the human body that sweep rhythmically to transport substances. Dysfunctional cilia are indicative of diseases that can disrupt organs such as the lungs and kidneys. Understanding cilia behavior is essential in diagnosing and treating such diseases. But, the tasks of automatically analyzing cilia are often a labor and time-intensive since there is a lack of automated segmentation. In this work we overcome this bottleneck by developing a robust, self-supervised framework exploiting the visual similarity of normal and dysfunctional cilia. This framework generates pseudolabels from optical flow motion vectors, which serve as training data for a semi-supervised neural network. Our approach eliminates the need for manual annotations, enabling accurate and efficient segmentation of both motile and immotile cilia.

**Keywords** Cilia, Unsupervised biomedical Image Segmentation, Optical Flow, Autoregressive, Deep Learning

## 1. Introduction

Cilia are hair-like membranes that extend out from the surface of the cells and are present on a variety of cell types such as lungs and brain ventricles and can be found in the majority of vertebrate cells. Categorized into motile and primary, motile cilia can help the cell to propel, move the flow of fluid, or fulfill sensory functions, while primary cilia act as signal receivers, translating extracellular signals into cellular responses [1]. Ciliopathies is the term commonly used to describe diseases caused by ciliary dysfunction. These disorders can result in serious issues such as blindness, neurodevelopmental defects, or obesity [2]. Motile cilia beat in a coordinated manner with a specific frequency and pattern [3]. Stationary, dyskinetic, or slow ciliary beating indicates ciliary defects. Ciliary beating is a fundamental biological process that is essential for the proper functioning of various organs, which makes understanding the ciliary phenotypes a crucial step towards understanding ciliopathies and the conditions stemming from it [4].

Identifying and categorizing the motion of cilia is an essential step towards understanding ciliopathies. However, this is generally an expert-intensive process. Studies have proposed methods that automate the ciliary motion assessment [5]. These methods rely on large amounts of labeled data that are annotated manually which is a costly, time-consuming, and error-prone task. Consequently, a significant bottleneck to automating cilia analysis is a lack of automated segmentation. Segmentation has remained a bottleneck of the pipeline due to the poor performance of even state-of-the-art models on some datasets. These datasets tend to exhibit significant spatial artifacts (light diffraction, out-of-focus cells, etc.) which confuse traditional image segmentation models [6].

Video segmentation techniques tend to be more robust to such noise, but still struggle due to the wild inconsistencies in cilia behavior: while healthy cilia have regular and predictable movements, unhealthy cilia display a wide range of motion, including a lack of motion altogether [7]. This lack of motion especially confounds movement-based methods which otherwise have no way of discerning the cilia from other non-cilia parts of the video. Both image and video segmentation techniques tend to require expert-labeled ground truth segmentation masks. Image segmentation requires the masks in order to effectively train neural segmentation models to recognize cilia, rather than other spurious textures. Video segmentation, by contrast, requires these masks in order to properly recognize both healthy and diseased cilia as a single cilia category, especially when the cilia show no movement.

To address this challenge, we propose a two-stage image segmentation model designed to obviate the need for expert-drawn masks. We first build a corpus of segmentation masks based on optical flow (OF) thresholding over a subset of healthy training data with guaranteed motility. We then train a semi-supervised neural segmentation model to identify both motile and immotile data as a single segmentation category, using the flow-generated masks as “pseudolabels”. These pseudolabels operate as “ground truth” for the model while acknowledging the intrinsic uncertainty of the labels. The fact that motile and immotile cilia tend to be visually similar in snapshot allows us to generalize the domain of the model from motile cilia to all cilia. Combining these stages results in a semi-supervised framework that does not rely on any expert-drawn ground-truth segmentation masks, paving the way for full automation of a general cilia analysis pipeline.

The rest of this article is structured as follows: The Section 2 enumerates the studies relevant to our methodology, followed by a detailed description of our approach in the Section 3. Finally, the Section 4 delineates our experiment and provides a discussion of the results obtained.

## 2. Background

Dysfunction in ciliary motion indicates diseases known as ciliopathies, which can disrupt the functionality of critical organs like the lungs and kidneys. Understanding ciliary motion is crucial for diagnosing and understanding these conditions. The development of diagnosis and treatment requires the measurement of different cell properties including size, shape, and motility [8].

Accurate analysis of ciliary motion is essential but challenging due to the limitations of manual analysis, which is labor-intensive, subjective, and prone to error. [5] proposed a modular generative pipeline that automates ciliary motion analysis by segmenting, representing, and modeling the dynamic behavior of cilia, thereby reducing the need for expert intervention and improving diagnostic consistency. [9] developed a computational pipeline using dynamic texture analysis and machine learning to objectively and quantitatively assess ciliary motion, achieving over 90% classification accuracy in identifying abnormal ciliary motion associated with diseases like primary ciliary dyskinesia (PCD). Additionally, [4] explored advanced feature extraction techniques like Zero-phase PCA Sphering (ZCA) and Sparse Autoencoders (SAE) to enhance cilia segmentation accuracy. These methods address challenges posed by noisy, partially occluded, and out-of-phase imagery, ultimately improving the overall performance of ciliary motion analysis pipelines. Collectively, these approaches aim to enhance diagnostic accuracy and efficiency, making ciliary motion analysis more accessible and reliable, thereby improving patient outcomes through early and accurate detection of ciliopathies. However, these studies rely on manually labeled data. The segmentation masks and ground-truth annotations, which are essential for training the models and validating their performance, are generated by expert reviewers. This dependence on manually labeled data is a significant limitation making automated cilia segmentation the bottleneck to automating cilia analysis.

In the biomedical field, where labeled data is often scarce and costly to obtain, several solutions have been proposed to augment and utilize available data effectively. These include semi-supervised learning [10], which utilizes both labeled and unlabeled data to enhance learning accuracy by leveraging the data's underlying distribution. Active learning [11] focuses on selectively querying the most informative data points for expert labeling, optimizing the training process by using the most valuable examples. Data augmentation techniques [12], [13], [14], [15], [16], [17], [18], [19], such as image transformations and synthetic data generation through Generative Adversarial Networks [20], [21], increase the diversity and volume of training data, enhancing model robustness and reducing overfitting. Transfer learning [16], [22], [23], [24] transfers knowledge from one task to another, minimizing the need for extensive labeled data in new tasks. Self-supervised learning [25], [26], [27] creates its labels by defining a pretext task, like predicting the position of a randomly cropped image patch, aiding in the learning of useful data representations. Additionally, few-shot, one-shot, and zero-shot learning techniques [28], [29] are designed to operate with minimal or no labeled examples, relying on generalization capabilities or metadata for making predictions about unseen classes.

A promising approach to overcome the dependency on manually labeled data is the use of unsupervised methods to generate ground truth masks. Unsupervised methods do not require prior knowledge of the data [30]. Using domain-specific cues unsupervised learning techniques can automatically discover patterns and structures in the data without the need for labeled examples, potentially simplifying the process of generating accurate segmentation masks for cilia. Inspired by advances in unsupervised methods for image segmentation, in this work, we firstly compute the motion vectors using optical flow of the ciliary regions and then apply autoregressive modelling to capture their temporal dynamics. Autoregressive modelling is advantageous since the labels are features themselves. By analyzing the OF vectors, we can identify the characteristic motion of cilia, which allows us to generate pseudolabels as ground truth segmentation masks. These pseudolabels are then used to train a robust semi-supervised neural network, enabling accurate and automated segmentation of both motile and immotile cilia.

## 3. Methodology

Dynamic textures, such as sea waves, smoke, and foliage, are sequences of images of moving scenes that exhibit certain stationarity properties in time [31]. Similarly, ciliary motion can be considered as dynamic textures for their orderly rhythmic beating. Taking advantage of this temporal regularity in ciliary motion, OF can be used to compute the flow vectors of each pixel of high-speed videos of cilia. In conjunction with OF, autoregressive (AR) parameterization of the OF property of the video yields a manifold that quantifies the characteristic motion in the cilia. The low dimension of this manifold contains the majority of variations within the data, which can then be used to segment the motile ciliary regions.

### 3.1. Optical Flow Properties

Taking advantage of this temporal regularity in ciliary motion, we use OF to capture the motion vectors of ciliary regions in high-speed videos. OF provides the horizontal $(u)$ and vertical $(v)$ components of the motion for each pixel. From these motion vectors, several components can be derived such as the magnitude, direction, divergence, and importantly, the curl (rotation). The curl, in this context, represents the rotational motion of the cilia, which is indicative of their rhythmic beating patterns. We extract flow vectors of the video recording of cilia, under the assumption that pixel intensity remains constant throughout the video.

$$I(x, y, t) = I(x + u\delta t, y + v\delta t, t + \delta t) \tag{1}$$

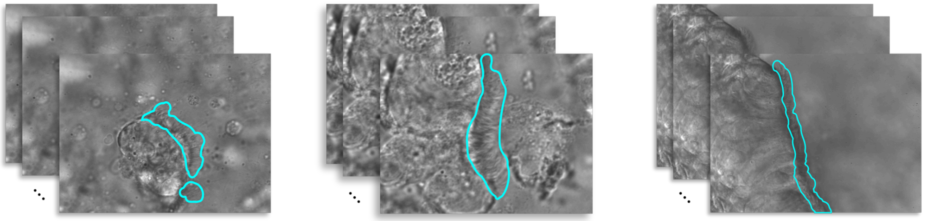


**Figure 1.** *A sample of three videos in our cilia dataset with their manually annotated ground truth masks.*

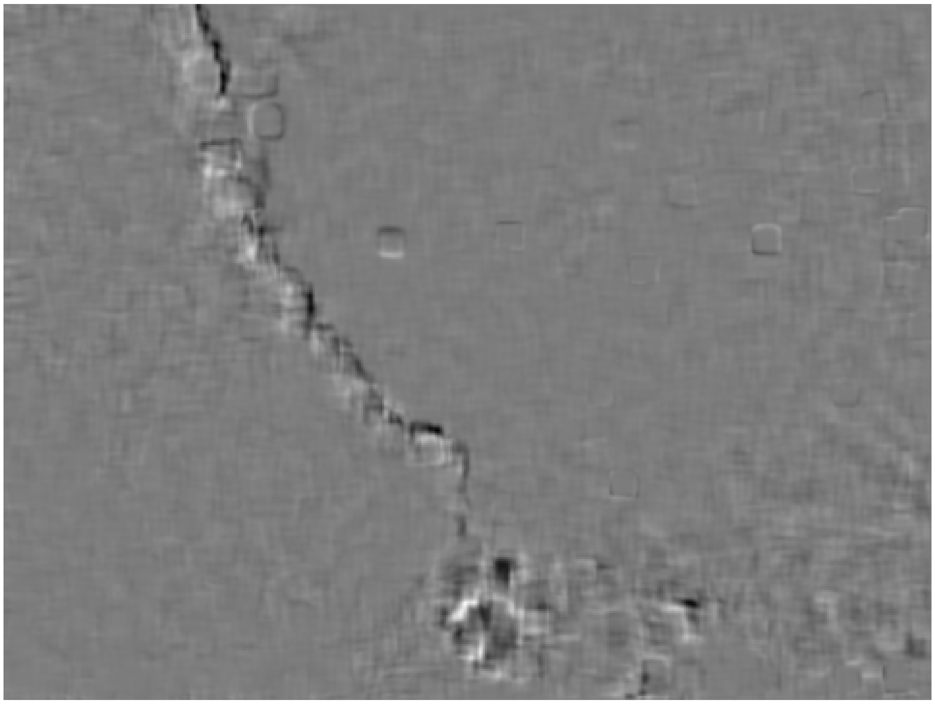


**Figure 2.** *Representation of rotation (curl) component of OF at a random time*

Where $I(x, y, t)$ is the pixel intensity at position $(x, y)$ a time $t$. Here, $(u\delta t, v\delta t)$ are small changes in the next frame taken after $\delta t$ time, and $(u, v)$, respectively, are the OF components that represent the displacement in pixel positions between consecutive frames in the horizontal and vertical directions at pixel location $(x, y)$.

### *3.2. Autoregressive Modeling*

Figure 2 shows a sample of the OF component at a random time. From OF vectors, elemental components such as rotation are derived, which highlights the ciliary motion by capturing twisting and turning movements. To model the temporal evolution of these motion vectors, we employ an autoregressive (AR) model [32]. This model captures the dynamics of the flow vectors over time, allowing us to understand how the motion evolves frame by frame. The AR model helps in decomposing the motion into a low-dimensional subspace, which simplifies the complex ciliary motion into more manageable analyses.

$$y_t = C\vec{x}_t + \vec{u} \tag{2}$$

$$\vec{x}_t = A_1\vec{x}_{t-1} + A_2\vec{x}_{t-2} + \ldots + A_d\vec{x}_{t-d} + \vec{v}_t \tag{3}$$

In equation Equation 2, $\vec{y}_t$ represents the appearance of cilia at time $t$ influenced by noise $\vec{u}$. Equation Equation 3 represents the state $\vec{x}$ of the ciliary motion in a low-dimensional subspace defined by an orthogonal basis $C$ at time $t$, plus a noise term $\vec{v}_t$ and how the state changes from $t$ to $t + 1$.

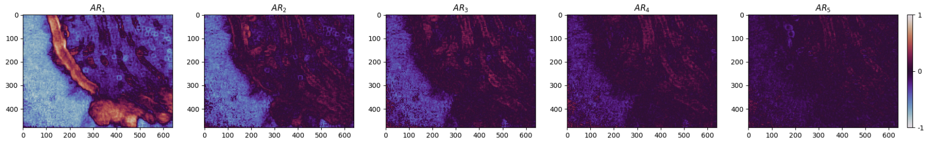


**Figure 3**. *The pixel representation of the 5-order AR model of the OF component of a sample video. The $x$ and $y$ axes correspond to the width and height of the video.*

Equation Equation 3 is a decomposition of each frame of a ciliary motion video $\vec{y}_t$ into a low-dimensional state vector $\vec{x}_t$ using an orthogonal basis $C$. This equation at position $x_t$ is a function of the sum of $d$ of its previous positions $\vec{x}_{t-1}, \vec{x}_{t-2}, \vec{x}_{t-d}$, each multiplied by its corresponding coefficients $A = A_1, A_2, ..., A_d$. The noise terms $\vec{u}$ and $\vec{v}$ are used to represent the residual difference between the observed data and the solutions to the linear equations. The variance in the data is predominantly captured by a few dimensions of $C$, simplifying the complex motion into manageable analyses.

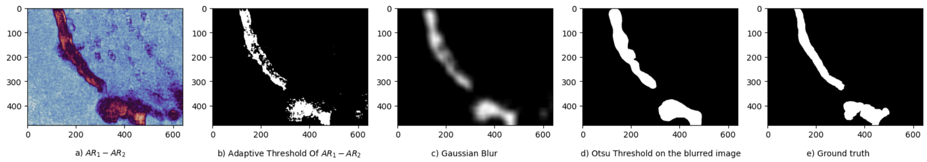
Each order of the autoregressive model roughly aligns with different frequencies within the data, therefore, in our experiments, we chose $d = 5$ as the order of our autoregressive model. This choice allows us to capture a broader temporal context, providing a more comprehensive understanding of the system's dynamics. We then created raw masks from this lower-dimensional subspace, and further enhanced them with adaptive thresholding to remove the remaining noise.

In Figure 3, the first-order AR parameter is showing the most variance in the video, which corresponds to the frequency of motion that cilia exhibit. The remaining orders have correspondence with other different frequencies in the data caused by, for instance, camera shaking. Evidently, simply thresholding the first-order AR parameter is adequate to produce an accurate mask, however, in order to get a more refined result we subtracted the second order from the first one, followed by a Min-Max normalization of pixel intensities and scaling to an 8-bit unsigned integer range. We used adaptive thresholding to extract the mask on all videos of our dataset. The generated masks exhibited under-segmentation in the ciliary region, and sparse over-segmentation in other regions of the image. To overcome this, we adapted a Gaussian blur filter followed by an Otsu thresholding to restore the under-segmentation and remove the sparse over-segmentation. Figure 4 illustrates the steps of the process.

### 3.3. Training the model

Our dataset includes 512 videos, with 437 videos of dyskinetic cilia and 75 videos of healthy motile cilia, referred to as the control group. The control group is split into %85 and %15 for training and validation respectively. 108 videos in the dyskinetic group are manually annotated which are used in the testing step. Figure 1 shows annotated samples of our dataset.

In our study, we employed a Feature Pyramid Network (FPN) [33] architecture with a ResNet-34 encoder. The model was configured to handle grayscale images with a single



a) $AR_1 - AR_2$
b) Adaptive Threshold Of $AR_1 - AR_2$
c) Gaussian Blur
d) Otsu Threshold on the blurred image
e) Ground truth

**Figure 4**. *The process of computing the masks. a) Subtracting the second-order AR parameter from the first-order, followed by b) Adaptive thresholding, which suffers from under/over-segmentation. c) A Gaussian blur filter, followed by d) An Otsu thresholding eliminates the under/over-segmentation.*

**Table 1**. *Summary of model architecture, training setup, and dataset distribution*

| Aspect | Details |
|---|---|
| **Architecture** | FPN with ResNet-34 encoder |
| **Input** | Grayscale images with a single input channel |
| **Batch Size** | 2 |
| **Training Samples** | 28,869 |
| **Validation Samples** | 5,095 |
| **Test Samples** | 108 |
| **Loss Function** | Binary Cross-Entropy Loss |
| **Optimizer** | Adam optimizer with a learning rate of $10^{-3}$ |
| **Evaluation Metric** | Dice score during training, validation, and testing |
| **Data Augmentation Techniques** | Resizing, random cropping, and rotation |
| **Implementation** | Using a Python library with Neural Networks for Image Segmentation based on PyTorch [34] |

input channel and produce binary segmentation masks. For the training input, one mask is generated per video using our methodology, and we use all of the frames from each video in the control group making a total of 33,964 input images. We utilized Binary Cross-Entropy Loss for training and the Adam optimizer with a learning rate of $10^{-3}$. To evaluate the model's performance, we calculated the Dice score during training and validation. Data augmentation techniques, including resizing, random cropping, and rotation, were applied to enhance the model's generalization capability. The implementation was done using a library [34] based on PyTorch Lightning to facilitate efficient training and evaluation. Table 1 contains a summary of the model parameters and specifications.

The next section discusses the results of the experiment and the performance of the model in detail.

## 4. Results and Discussion

The model's performance metrics, including IoU, Dice score, sensitivity, and specificity, are summarized in Table 2. The validation phase achieved an IoU of 0.398 and a Dice score of 0.569, which indicates a moderate overlap between the predicted and ground truth masks. The high sensitivity (0.997) observed during validation suggests that the model is proficient in identifying ciliary regions, albeit with a specificity of 0.882, indicating some degree of false positives. In the testing phase, the IoU and Dice scores decreased to 0.132 and 0.233, respectively, reflecting the challenges posed by the dyskinetic cilia data, which were not included in the training or validation sets. Despite this, the model maintained a sensitivity of 0.479 and specificity of 0.806.

Figure 5 provides visual examples of the model's predictions on dyskinetic cilia samples, alongside the manually labeled ground truth and thresholded predictions. The dyskinetic samples were not used in the training or validation phases. These predictions were generated after only 15 epochs of training with a small training data. The visual comparison reveals that, while the model captures the general structure of ciliary regions, there are instances of under-segmentation and over-segmentation, which are more pronounced in the dyskinetic samples. This observation is consistent with the quantitative metrics, suggesting that further refinement of the pseudolabel generation process or model architecture could enhance segmentation accuracy.

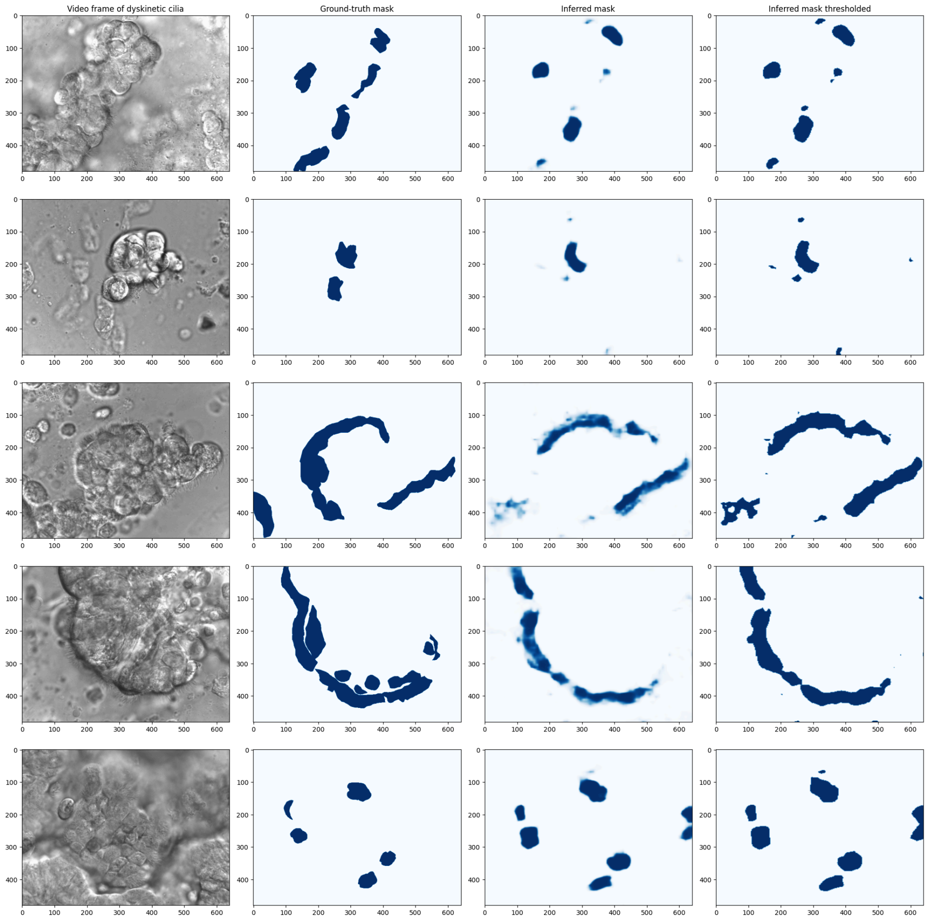


**Figure 5**. *The model predictions on 5 dyskinetic cilia samples. The first column shows a frame of the video, the second column shows the manually labeled ground truth, the third column is the model's prediction, and the last column is a thresholded version of the prediction.*

## *4.1. Training the model using control and dyskinetic data*

Since dyskinetic videos contain cilia that show some degree of movement we generated pseudo-labels for 283 dyskinetic videos and used them along with the 76 control videos from the previous experiment in another experiment. Training the model again for 15 epochs over healthy and dyskinetic videos exhibited a loss of performance in the validation phase, however, in the testing phase all of the metrics improved except for the specificity. Since in this experiment the model was trained on an additional subset of the dyskinetic videos, improved performance in detecting and masking dyskinetic ciliary regions is expected. The results are depicted in Table 3.

After using the model to infer more samples we detected a pattern for how the model performs in regions with specific visual properties. We observed that the model can more successfully and more accurately detect ciliary regions in images where they appear sharp

**Table 2**. *The performance of the model in validation and testing phases after 15 epochs of training.*

| **Phases** | **Metrics** | | | |
|---|---|---|---|---|
| | IoU over dataset | Dice Score | Sensitivity | Specificity |
| Validation | 0.398 | 0.569 | 0.997 | 0.882 |
| Testing | 0.132 | 0.233 | 0.479 | 0.806 |

**Table 3**. *The performance of the model after retraining with an addition of 283 videos of dyskinetic cilia to the training dataset.*

| **Phases** | **Metrics** | | | |
|---|---|---|---|---|
| | IoU over dataset | Dice Score | Sensitivity | Specificity |
| Validation | 0.202 | 0.337 | 0.999 | 0.765 |
| Testing | 0.139 | 0.245 | 0.732 | 0.696 |

and in focus, and do not overlap other cellular structures. On the other hand, as shown in all samples in Figure 5, the most number of false negatives occur where the ciliary regions are in close proximity to other cellular structures or overlapping them. Furthermore, the most false positives occur along sharp cellular borders. Cell borders are most where cilia can be found the most in the videos, and the model may have learnt to look for or prioritize sharp cell borders and boundaries as ciliary regions. More investigation is required to further examine whether the model's attention mechanism or feature extraction layers are overly biased towards sharp edges and boundaries, potentially leading to incorrect predictions. This investigation could involve analyzing the model's learned features, adjusting training strategies, or incorporating additional data augmentation techniques to improve its performance in complex regions.

The results show the potential of our approach to reduce the reliance on manually labeled data for cilia segmentation. The use of this unsupervised learning framework allows the model to generalize from the motile cilia domain to the more variable dyskinetic cilia, although with some limitations in accuracy. Future work could focus on expanding the dataset and improving the process of generating pseudolabels to enhance the model's accuracy.

## 5. Conclusions

In this paper, we introduced a self-supervised framework for cilia segmentation that eliminates the need for expert-labeled ground truth segmentation masks. Our approach takes advantage of the inherent visual similarities between healthy and unhealthy cilia to generate pseudolabels from optical flow-based motion segmentation of motile cilia. These pseudolabels are then used as ground truth for training a semi-supervised neural network capable of identifying regions containing dyskinetic cilia. Our results indicate that the self-supervised framework is a promising step towards automated cilia analysis. The model's ability to generalize from motile to dyskinetic cilia demonstrates its potential applicability in clinical settings. Although there are areas for improvement, such as enhancing segmentation accuracy and expanding the dataset, the framework sets the foundation for more efficient and reliable cilia analysis pipelines.

## References

[1] S. Hoyer-Fender, "Primary and Motile Cilia: Their Ultrastructure and Ciliogenesis," in *Cilia and Nervous System Development and Function*, K. L. Tucker and T. Caspary, Eds., Dordrecht: Springer Netherlands, 2013, pp. 1–53. doi: 10.1007/978-94-007-5808-7_1.

[2] J. N. Hansen, S. Rassmann, B. Stüven, N. Jurisch-Yaksi, and D. Wachten, "CiliaQ: a simple, open-source software for automated quantification of ciliary morphology and fluorescence in 2D, 3D, and 4D images," *The European Physical Journal E*, vol. 44, no. 2, p. 18, 2021, doi: https://doi.org/10.1140/epje/s10189-021-00031-y.

[3] W. Lee, P. Jayathilake, Z. Tan, D. Le, H. Lee, and B. Khoo, "Muco-ciliary transport: effect of mucus viscosity, cilia beat frequency and cilia density," *Computers & Fluids*, vol. 49, no. 1, pp. 214–221, 2011, doi: https://doi.org/10.1016/j.compfluid.2011.05.016.

[4] M. Zain, E. Miller, S. Quinn, and C. Lo, "Low Level Feature Extraction for Cilia Segmentation," in *Proceedings of the Python in Science Conference*, 2022. doi: https://doi.org/10.25080/majora-212e5952-026.

[5] M. Zain *et al.*, “Towards an unsupervised spatiotemporal representation of cilia video using a modular generative pipeline,” in *Proceedings of the Python in Science Conference*, 2020. doi: http://dx.doi.org/10.25080/Majora-342d178e-017.

[6] C. Lu, M. Marx, M. Zahid, C. W. Lo, C. Chennubhotla, and S. P. Quinn, “Stacked Neural Networks for end-to-end ciliary motion analysis,” *arXiv preprint arXiv:1803.07534*, 2018, doi: https://doi.org/10.48550/arXiv.1803.07534.

[7] C. Kempeneers and M. A. Chilvers, “To beat, or not to beat, that is question! The spectrum of ciliopathies,” *Pediatric Pulmonology*, vol. 53, no. 8, pp. 1122–1129, 2018, doi: https://doi.org/10.1002/ppul.24078.

[8] S. A. Vaezi, G. Orlando, M. S. Fazli, G. E. Ward, S. N. Moreno, and S. Quinn, “A Novel Pipeline for Cell Instance Segmentation, Tracking and Motility Classification of Toxoplasma Gondii in 3D Space.,” in *SciPy*, 2022, pp. 60–63. doi: https://doi.org/10.25080/majora-212e5952-009.

[9] S. P. Quinn, M. J. Zahid, J. R. Durkin, R. J. Francis, C. W. Lo, and S. C. Chennubhotla, “Automated identification of abnormal respiratory ciliary motion in nasal biopsies,” *Science translational medicine*, vol. 7, no. 299, p. 299, 2015, doi: http://dx.doi.org/10.1126/scitranslmed.aaa1233.

[10] J. E. Van Engelen and H. H. Hoos, “A survey on semi-supervised learning,” *Machine learning*, vol. 109, no. 2, pp. 373–440, 2020, doi: http://dx.doi.org/10.1007/s10994-019-05855-6.

[11] B. Settles, “Active learning literature survey,” 2009.

[12] C. Chen *et al.*, “Improving the Generalizability of Convolutional Neural Network-Based Segmentation on CMR Images,” *Frontiers in Cardiovascular Medicine*, vol. 7, 2020, doi: https://doi.org/10.3389/fcvm.2020.00105.

[13] J. Krois *et al.*, “Generalizability of deep learning models for dental image analysis,” *Scientific Reports*, vol. 11, no. 1, p. 6102, 2021, doi: http://dx.doi.org/10.1038/s41598-021-85454-5.

[14] W. Yan *et al.*, “MRI Manufacturer Shift and Adaptation: Increasing the Generalizability of Deep Learning Segmentation for MR Images Acquired with Different Scanners,” *Radiology: Artificial Intelligence*, vol. 2, no. 4, p. e190195, 2020, doi: 10.1148/ryai.2020190195.

[15] V. Sandfort, K. Yan, P. J. Pickhardt, and R. M. Summers, “Data augmentation using generative adversarial networks (CycleGAN) to improve generalizability in CT segmentation tasks,” *Scientific Reports*, vol. 9, no. 1, p. 16884, 2019, doi: https://doi.org/10.1016/j.imu.2021.100779.

[16] A. Yakimovich, A. Beaugnon, Y. Huang, and E. Ozkirimli, “Labels in a haystack: Approaches beyond supervised learning in biomedical applications,” *Patterns*, vol. 2, no. 12, p. 100383, 2021, doi: https://doi.org/10.1016/j.patter.2021.100383.

[17] D. A. Van Dyk and X.-L. Meng, “The art of data augmentation,” *Journal of Computational and Graphical Statistics*, vol. 10, no. 1, pp. 1–50, 2001, doi: http://dx.doi.org/10.1198/10618600152418584.

[18] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “Imagenet classification with deep convolutional neural networks,” *Advances in neural information processing systems*, vol. 25, 2012, doi: 10.1145/3065386.

[19] O. Ronneberger, P. Fischer, and T. Brox, “U-net: Convolutional networks for biomedical image segmentation,” in *Medical image computing and computer-assisted intervention–MICCAI 2015: 18th international conference, Munich, Germany, October 5-9, 2015, proceedings, part III 18*, 2015, pp. 234–241. doi: 10.48550/arXiv.1505.04597.

[20] I. Goodfellow *et al.*, “Generative adversarial nets,” *Advances in neural information processing systems*, vol. 27, 2014, doi: 10.48550/arXiv.1406.2661.

[21] X. Yi, E. Walia, and P. Babyn, “Generative adversarial network in medical imaging: A review,” *Medical image analysis*, vol. 58, p. 101552, 2019, doi: https://doi.org/10.48550/arXiv.1809.07294.

[22] T. H. Sanford *et al.*, “Data Augmentation and Transfer Learning to Improve Generalizability of an Automated Prostate Segmentation Model,” *AJR Am J Roentgenol*, vol. 215, no. 6, pp. 1403–1410, 2020, doi: http://dx.doi.org/10.2214/AJR.19.22347.

[23] M. Raghu, C. Zhang, J. Kleinberg, and S. Bengio, “Transfusion: Understanding Transfer Learning for Medical Imaging,” in *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché Buc, E. Fox, and R. Garnett, Eds., 2019. doi: https://doi.org/10.48550/arXiv.1902.07208.

[24] M. L. Hutchinson, E. Antono, B. M. Gibbons, S. Paradiso, J. Ling, and B. Meredig, “Overcoming data scarcity with transfer learning.” 2017. doi: https://doi.org/10.48550/arXiv.1711.05099.

[25] D. Kim, D. Cho, and I. S. Kweon, “Self-supervised video representation learning with space-time cubic puzzles,” in *Proceedings of the AAAI conference on artificial intelligence*, 2019, pp. 8545–8552. doi: https://doi.org/10.48550/arXiv.1811.09795.

[26] A. Kolesnikov, X. Zhai, and L. Beyer, “Revisiting self-supervised visual representation learning,” in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2019, pp. 1920–1929. doi: https://doi.org/10.48550/arXiv.1901.09005.

[27] A. Mahendran, J. Thewlis, and A. Vedaldi, “Cross pixel optical-flow similarity for self-supervised learning,” in *Computer Vision–ACCV 2018: 14th Asian Conference on Computer Vision, Perth, Australia, December 2–6, 2018, Revised Selected Papers, Part V 14*, 2019, pp. 99–116. doi: https://doi.org/10.48550/arXiv.1807.05636.

[28] F.-F. Li, R. Fergus, P. Perona, and others, “One-shot learning of object categories,” *IEEE Trans. Pattern Anal. Mach. Intell*, vol. 28, no. 4, pp. 594–611, 2006, doi: http://dx.doi.org/10.1109/TPAMI.2006.79.

[29] E. G. Miller, N. E. Matsakis, and P. A. Viola, “Learning from one example through shared densities on transforms,” in *Proceedings IEEE Conference on Computer Vision and Pattern Recognition. CVPR 2000 (Cat. No. PR00662)*, 2000, pp. 464–471. doi: https://doi.org/10.1109/CVPR.2000.855856.

[30] T. Khatibi, N. Rezaei, L. Ataei Fashtami, and M. Totonchi, “Proposing a novel unsupervised stack ensemble of deep and conventional image segmentation (SEDCIS) method for localizing vitiligo lesions in skin images,” *Skin Research and Technology*, vol. 27, no. 2, pp. 126–137, 2021, doi: http://dx.doi.org/10.1111/srt.12920.

[31] G. Doretto, A. Chiuso, Y. N. Wu, and S. Soatto, “Dynamic textures,” *International journal of computer vision*, vol. 51, pp. 91–109, 2003, doi: https://doi.org/10.1023/A:1021669406132.

[32] M. Hyndman, A. D. Jepson, and D. J. Fleet, “Higher-order Autoregressive Models for Dynamic Textures,” in *British Machine Vision Conference*, 2007. doi: http://dx.doi.org/10.5244/C.21.76.

[33] A. Kirillov, K. He, R. Girshick, and P. Dollár, “A unified architecture for instance and semantic segmentation,” in *Computer Vision and Pattern Recognition Conference*, 2017. doi: https://doi.org/10.48550/arXiv.2112.04603.

[34] P. Iakubovskii, “Segmentation Models Pytorch.” [Online]. Available: https://github.com/qubvel/segmentation_models.pytorch

Open Access | https://doi.org/10.25080/TPGN6857

SciPy 2024

July 8 - July 14, 2024

Proceedings of the 23nd Python in Science Conference

ISSN: 2575-9752

# Computational Resource Optimisation in Feature Selection under Class Imbalance Conditions

**Amadi Gabriel Udu**[1,2], **Andrea Lecchini-Visintini**[3], **Steve R. Gunn**[3], **Norman Osa-uwagboe**[4], **Maryam Khaksar Ghalati**[1], and **Hongbiao Dong**[1]

[1]School of Engineering, University of Leicester, LE1 7RH, Leicester, UK., [2]Air Force Institute of Technology, Air Force Base, PMB 2104, Kaduna, Nigeria., [3]School of Electronics and Computer Science, University of Southampton, SO17 1BJ, Southampton, UK., [4]Wolfson School of Mechanical, Electrical, and Manufacturing Engineering, Loughborough University, LE11 3TU, Loughborough, UK.

## Abstract

Feature selection is crucial for reducing data dimensionality as well as enhancing model interpretability and performance in machine learning tasks. However, selecting the most informative features in large dataset often incurs high computational costs. This study explores the possibility of performing feature selection on a subset of data to reduce the computational burden. The study uses five real-life datasets with substantial sample sizes and severe class imbalance ratios between 0.09 – 0.18. The results illustrate the variability of feature importance with smaller sample fractions in different models. In this cases considered, light gradient-boosting machine exhibited the least variability, even with reduced sample fractions, while also incurring the least computational resource.

**Keywords** feature selection, class imbalance, machine learning

**Published** Jul 10, 2024

**Correspondence to**
Amadi Gabriel Udu
agu1@le.ac.uk

## 1. Introduction

In the development of prediction models for real-world applications, two key challenges often arise: high-dimensionality resulting from the numerous features, and class-imbalance due to the rarity of samples in the positive class. Feature selection methods are utilized to address issues of high-dimensionality by selecting a smaller subset of relevant features, thus reducing noise, increasing interpretability, and enhancing model performance [1], [2], [3].

Studies [4], [5], [6], [7] on the performance of feature selection methods with class imbalance data have been undertaken on using synthetic and real-life datasets. A significant drawback noted was the computational cost of their approach on large sample sizes. While experimental investigations of feature selection amid class imbalance conditions have been studied in the literature, there is a need to further understand the effect of sample size on performance degradation of feature selection methods. This would offer valuable insights into tackling the associated resource expense involved in undertaking feature selection with respect to large sample sizes where class-imbalance exists, for a wide range of applications.

This study investigates the impact of performing feature selection on a reduced dataset on feature importance and model performance, using five real-life datasets characterised by large sample sizes and severe class imbalance structures. We employ a feature selection process that utilises permutation feature importance (PFI) and evaluate the feature importance on three selected models; namely light gradient-boosting machine (Light GBM), random forest (RF) and support vector machines (SVM). These models are popular in real-world machine learning (ML) studies and also serve as a benchmark for comparing novel models [8], [9], [10], [11], [12]. Feature importance was evaluated using the area under the Receiver Operator Characteristics (ROC) curve, commonly referred to as AUC owing to its suitability in class imbalance problems [13], [14]. The development of the ML framework and data visualisation in this study was facilitated by several key Python libraries. Pandas [15] and NumPy [16] were used for data loading and numerical computations, respectively. Scikit-learn [17] provided tools for data preprocessing, model development, and evaluation. Matplotlib [18] was employed for visualising data structures. Additionally, the SciPy [19] library's cluster, spatial, and stats modules were crucial for hierarchical clustering, Spearman rank correlation, and distance matrix computations.

The rest of the paper is organised as follows: Section 2 briefly outlines the methodology adopted, while Section 3 presents the results and discussion. The conclusion of the study is provided in Section 4.

## 2. Methodology

### 2.1. *Description of datasets*

Five real-life datasets from different subject areas were considered in this study. Four of the datasets were obtained from the UC Irvine ML repository, including CDC Diabetes Health Indicator [20], Census Income [21], Bank Marketing [22], and Statlog (Shuttle) [23]. The fifth dataset is Moisture Absorbed Composite [24] from a damage morphology study. The datasets are presented in Table 1. Notably, all datasets exhibited high class imbalance ratios from 0.09 - 0.18 (i.e., the ratio of the number of samples in the minority class over that of the majority class).

Building data-driven models in the presence of high dimensionality includes several steps such as data preprocessing, feature selection, model training and evaluation. To address class imbalance issues during model training, an additional resampling step may be performed to adjust the uneven distribution of class samples [25], [26], [27]. This paper, however, focuses on the feature selection method, model training, and the evaluation metrics adopted.

### 2.2. *Feature selection and model training*

To maintain a model-agnostic approach that is not confined to any specific ML algorithm, this study employed PFI for feature selection. PFI assesses how each feature affects the model's performance by randomly shuffling the values of a feature and noting the resulting change in performance. In essence, if a feature is important, shuffling its values should significantly reduce the model's performance since the model relies on that feature to make predictions. A positive importance score suggests that a feature is useful for the

**Table 1**. *Summary of datasets used in the study*

| Dataset | Features | Instances | Subject Area | Imbalance Ratio |
|---|---|---|---|---|
| Diabetes health indicator | 20 | 253,680 | Health and Medicine | 0.16 |
| Census income | 14 | 48,842 | Social Science | 0.09 |
| Bank marketing | 17 | 45,211 | Business | 0.13 |
| Statlog (shuttle) | 7 | 58,000 | Physics and Chemistry | 0.18 |
| Moisture absorbed composite | 9 | 295,461 | Mechanics of Materials | 0.11 |

model's prediction as permuting the values of the feature led to a decrease in the model's performance. Conversely, a negative importance score suggests that a feature might be introducing noise and the model might perform better without it. Thus, PFI interrupts the link between a feature and its predicted outcome, enabling us determine the extent to which a model relies on a particular feature [17], [28], [29]. It is noteworthy that the effect of permuting one feature could be negligible when features are collinear. Hence, an important feature may report a low score. To tackle this, a hierarchical cluster on a Spearman rank-order correlation can be adopted, with a threshold taking from visual inspection of the dendrograms in grouping features into clusters and selecting the feature to retain.

Datasets were loaded using pandas, and categorical features were encoded using one-hot encoding. The Spearman correlation matrix was computed and then converted into a distance matrix. Hierarchical clustering was subsequently performed using Ward's linkage method, and a threshold for grouping features into clusters was determined through visual inspection of the dendrograms, allowing for the selection of features to retain. Subsequently, the investigation proceeded in two steps. In step 1, all samples of the respective dataset was used. The dataset was split into training and test sets based on a test-size of 0.25. The respective classifiers were initialised using their default hyper-parameter settings and fitted on the training data. Thereafter, PFI was computed on the fitted model with number of times a feature is permuted set to 30 repeats. Lastly, the change in AUC was evaluated on the test set.

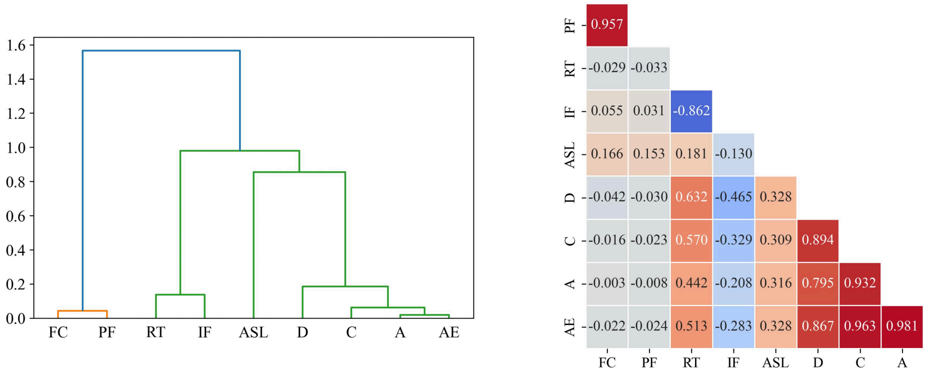
In the second step, we initiate three for-loops to handle the different features, fractions of samples, and repetition of the PFI process undertaken in step 1. Sample fraction sizes were taken from 10% – 100% in increments of 10%, with the entire process randomly repeated 10 times. This provided an array of 300 AUC scores for each sample fraction and respective feature of the PFI process. To ensure reproducibility, the random state for the classifiers, sample fractions, data split, and permutation importance were predefined. Computation processes were accelerated using the joblib parallel library on the Sulis High Performance Computing platform. A sample source code of step 2 is presented:

```
# Define the function for parallel execution
def process_feature(f_no, selected_features, df):
    for frac in np.round(np.arange(0.1, 1.1, 0.1), 1).tolist():  #loop for sample fractions
        for rand in range(10): #loop for 10 repeats of the process
            df_new = df.sample(frac=frac, random_state=rand)
...
            pfi = permutation_importance(model, X_val, y_val, n_repeats=30,
                                       random_state=rand, scoring='roc_auc', n_jobs=-1)
    return final_df
# Parallelise computation
results = Parallel(n_jobs=-1)(delayed(process_feature)(f_no, selected_features, df) for f_no in
range(len(selected_features)))
```

## 3. Results and Discussions

The hierarchical cluster and Spearman's ranking for moisture absorbed composite dataset is shown in Figure 1 and Figure 1 respectively (Frequency Centroid – FC, Peak Frequency – PF, Rise Time – RT, Initiation Frequency – IF, Average Signal Level – ASL, Duration – D, Counts – C, Amplitude – A and Absolute Energy – AE). Based on the visual inspection of the hierarchical cluster, a threshold of 0.8 was selected, thus, retaining features RT, C, ASL, and FC.

As observed in Figure 1, Frequency Centroid and Peak Frequency are in the same cluster with a highly correlated value of 0.957 shown in Figure 1. Similarly, Rise Time and Initia-



(a) (b)

**Figure 1**. *Feature relationship for moisture absorbed composite dataset; (a) hierarchical cluster, (b) Spearman correlation ranking.*

tion Frequency are clustered with a highly negative correlation of -0.862. Amplitude and Absolute Energy also exhibited a high positive correlation of 0.981.

Table 2 gives the median and interquartile (IQR) feature importance scores based on change in AUC for the LightGBM, RF and SVM models. These scores were obtained using all samples in the PFI process. Values emphasised in bold fonts represent the highest ranked feature for the respective models based on their median change in AUC.

From Table 2, SVM tended to be have very low scores in some datasets, possibly due to its reliance of support vectors in determining the decision boundaries. Thus, features with strong influence at the decision boundary but not directly affecting the support vectors may seem less important. For the Moisture Absorbed Composite dataset, the three classifiers reported similar scores for Frequency Centroid of 0.468, 0.466 and 0.422 respectively in Table 2.

However, in Bank Marketing dataset, LightGBM and RF identified Feature 1 as a relatively important feature, while SVM considered it insignificant. The mutability of importance scores for the classifiers considered underscores the need to explore multiple classifiers when undertaking a comprehensive investigation of feature importance for feature selection purposes.

Figure 2 shows the PFI process time and corresponding sample fractions for the Diabetes dataset, which has a substantial sample size of 253,680 instances. The results are based on one independent run, with PFI set at 30 feature-permuted repeats. For LightGBM and RF, the PFI process time increased linearly with larger sample fractions, whereas SVM experienced an exponential growth. LightGBM had the lowest computational cost, with CPU process times of 3.9 seconds and 28.8 seconds for 10% and 100% sample fractions, respectively. SVM required 21,263 seconds to process the entire dataset, reflecting a 9,345% increase in CPU computational cost compared to using a 10% sample fraction. SVM's poor performance relative to LightGBM and RF is likely due to its poor CPU parallelisability.

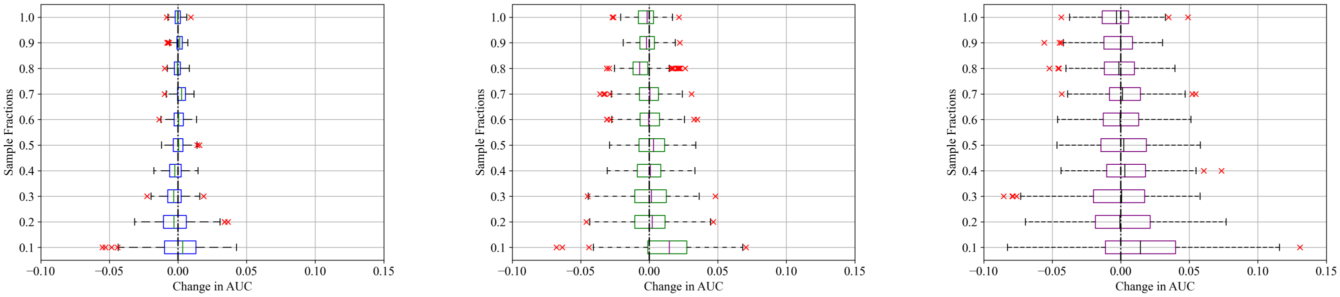
Figure 3a - Figure 3c present the PFI for Final Weight feature of Census Income dataset, evaluated across different sample fractions using LightGBM, RF, and SVM models, respectively. The change in AUC indicates the impact on model performance when Final Weight feature is permuted. Generally, for smaller sample fractions, there was a higher variability in AUC and prominence of outliers. This could be attributed to the increased influence of randomness, fewer data points, and sampling fluctuations for smaller sample fractions across the datasets.

**Table 2.** *Median and IQR feature importance scores based on change in AUC for LightGBM, RF and SVM models, (values in bold fonts represent the highest ranked feature for the respective models).*

| **Census Income** | | | | | | | | | | | **Bank Marketing** | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ID | Feature | LightGBM | | | RF | | | SVM | | | ID | Feature | LightGBM | | | RF | | | SVM | | |
| | | Med | IQR | | Med | IQR | | Med | IQR | | | | Med | IQR | | Med | IQR | | Med | IQR | |
| | | | 25th | 75th | | 25th | 75th | | 25th | 75th | | | | 25th | 75th | | 25th | 75th | | 25th | 75th |
| 0 | Age | 0.117 | 0.114 | 0.121 | 0.066 | 0.061 | 0.069 | $< 10^{-3}$ | $< 10^{-3}$ | $< 10^{-3}$ | 0 | Age | 0.016 | 0.015 | 0.017 | 0.026 | 0.024 | 0.027 | -0.001 | -0.001 | 0.002 |
| 1 | Final weight | -0.002 | -0.003 | -0.001 | $< 10^{-3}$ | -0.003 | 0.004 | 0.003 | $< 10^{-3}$ | 0.011 | 1 | Balance | 0.013 | 0.011 | 0.014 | 0.011 | 0.009 | 0.012 | 0.026 | 0.021 | 0.027 |
| 2 | Education-num | 0.085 | 0.080 | 0.087 | 0.063 | 0.061 | 0.068 | $< 10^{-3}$ | $< 10^{-3}$ | $< 10^{-3}$ | 2 | Day of week | 0.012 | 0.011 | 0.013 | 0.014 | 0.014 | 0.016 | 0.001 | 0.001> | 0.001 |
| 3 | Capital-gain | 0.049 | 0.048 | 0.052 | 0.047 | 0.046 | 0.050 | 0.029 | 0.026 | 0.030 | 3 | Duration | 0.256 | 0.253 | 0.261 | 0.211 | 0.209 | 0.215 | 0.154 | 0.148 | 0.157 |
| 4 | Workclass | $< 10^{-3}$ | $< 10^{-3}$ | 0.001 | 0.003 | 0.001 | 0.004 | $< 10^{-3}$ | $< 10^{-3}$ | $< 10^{-3}$ | 4 | PDays | 0.051 | 0.051 | 0.052 | 0.054 | 0.052 | 0.055 | 0.053 | 0.050 | 0.055 |
| 5 | Race | $< 10^{-3}$ | $< 10^{-3}$ | $< 10^{-3}$ | 0.001 | 0.001 | 0.002 | $< 10^{-3}$ | $< 10^{-3}$ | $< 10^{-3}$ | 5 | Job_b | 0.003 | 0.003 | 0.004 | 0.002 | 0.001 | 0.002 | $< 10^{-3}$ | $< 10^{-3}$ | $< 10^{-3}$ |
| | | | | | | | | | | | 6 | Job_m | $< 10^{-3}$ | $< 10^{-3}$ | 0.001 | 0.001 | 0.001 | 0.002 | $< 10^{-3}$ | $< 10^{-3}$ | $< 10^{-3}$ |
| | | | | | | | | | | | 7 | Housing | 0.026 | 0.025 | 0.027 | 0.032 | 0.031 | 0.034 | 0.001 | 0.001 | 0.001 |
| Statlog (Shuttle) | | | | | | | | | | | Diabetes | | | | | | | | | | |
| 0 | Rad Flow | 0.355 | 0.350 | 0.360 | 0.387 | 0.383 | 0.389 | 0.253 | 0.249 | 0.259 | 0 | HighBP | 0.128 | 0.127 | 0.129 | 0.128 | 0.128 | 0.130 | 0.066 | 0.065 | 0.067 |
| 1 | Fpv Close | 0.005 | 0.005 | 0.005 | 0.012 | 0.011 | 0.013 | $< 10^{-3}$ | $< 10^{-3}$ | 0.001 | 1 | CholCheck | 0.009 | 0.009 | 0.010 | 0.011 | 0.010 | 0.011 | -0.001 | -0.001 | -0.001 |
| 2 | Fpv Open | 0.241 | 0.239 | 0.244 | 0.274 | 0.270 | 0.277 | 0.319 | 0.316 | 0.322 | 2 | BMI | 0.080 | 0.078 | 0.081 | 0.079 | 0.077 | 0.080 | -0.073 | -0.074 | -0.072 |
| | | | | | | | | | | | 3 | Smoker | 0.004 | 0.004 | 0.005 | 0.004 | 0.004 | 0.005 | 0.026 | 0.025 | 0.027 |
| Moisture Absorbed Composites | | | | | | | | | | | | | | | | | | | | | |
| 0 | Risetime | 0.005 | 0.005 | 0.005 | 0.009 | 0.008 | 0.009 | 0.004 | 0.003 | 0.004 | | | | | | | | | | | |
| 1 | Counts | 0.037 | 0.037 | 0.037 | 0.075 | 0.073 | 0.075 | 0.009 | 0.009 | 0.009 | | | | | | | | | | | |
| 2 | ASL | 0.034 | 0.034 | 0.034 | 0.072 | 0.071 | 0.073 | $< 10^{-3}$ | $< 10^{-3}$ | $< 10^{-3}$ | | | | | | | | | | | |
| 3 | Freq. Centroid | 0.468 | 0.466 | 0.470 | 0.463 | 0.461 | 0.465 | 0.422 | 0.421 | 0.425 | | | | | | | | | | | |

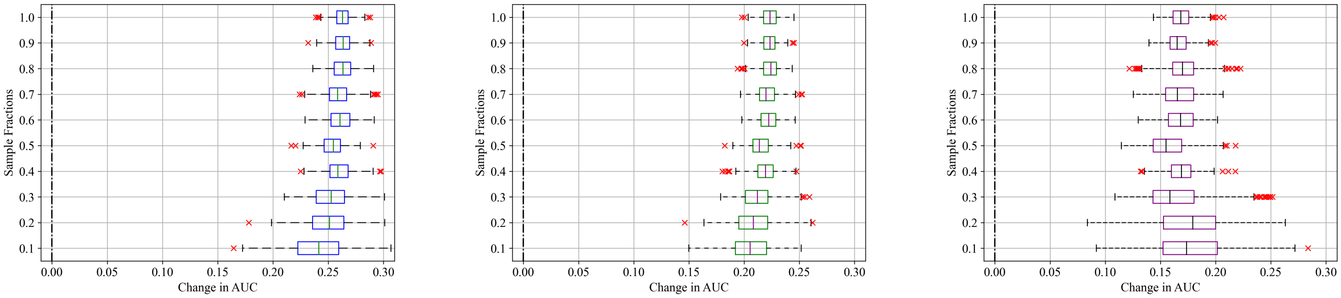
For LightGBM model in Figure 3a, the median change in AUC was close to zero, indicating that Final Weight had minimal impact on model performance, as noted in Table 2. Similar results were recorded in Figure 4a - Figure 4c for the Duration feature of Bank Marketing dataset, where all models exhibited similarly high feature importance scores. Even for sample fractions of 0.5, LightGBM and RF appeared to give similar importance scores to using the entire data sample. On the other hand, SVM exhibited a higher median change in AUC, indicating that the Final Weight feature had a more significant impact on its performance. Additionally, SVM showed the greatest variability and the most prominent outliers, particularly at lower sample fractions. This was noticeable in Figure 5a - Figure 5c, where all classifiers reported similar importance scores as noted in Table 2. This variability and the presence of outliers suggest that the model's performance is less stable when features are permuted.

PFI can provide insights into the importance of features, but it is susceptible to variability, especially with smaller sample sizes. Thus, complementary feature selection methods could be explored to validate feature importance. Future work could investigate the variability



(a) (b) (c)

**Figure 3.** *Sample fractions and corresponding change in AUC for Final Weight feature of Census Income dataset; (a) LightGBM, (b) RF, and (c) SVM.*



(a) (b) (c)

**Figure 4.** *Sample fractions and corresponding change in AUC for Duration feature of Bank Marketing dataset; (a) LightGBM, (b) RF, and (c) SVM.*

of features under particular models and sample sizes, with a view to evolving methods of providing a more stable information to the models.

## 4. Conclusion

Feature selection for large datasets incurs considerable computational cost in the model development process of various ML tasks. This study undertakes a preliminary investiga-

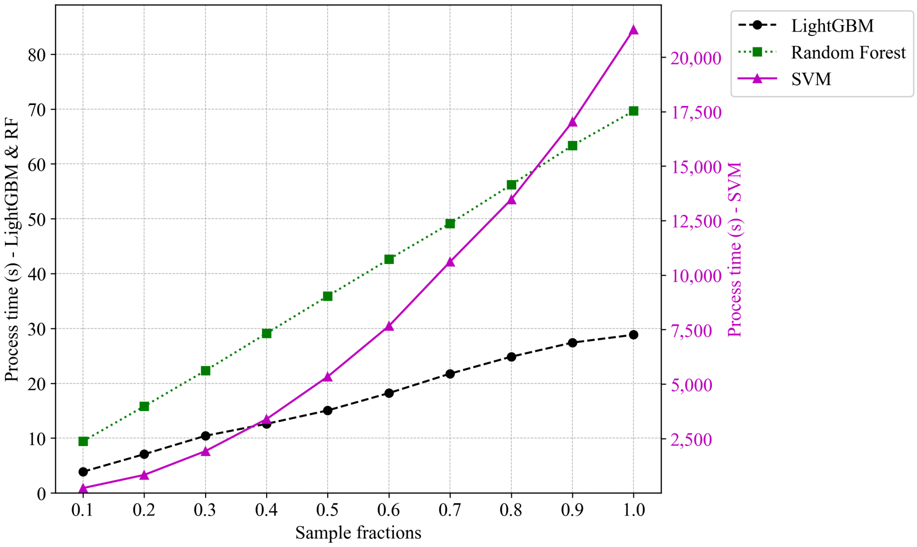
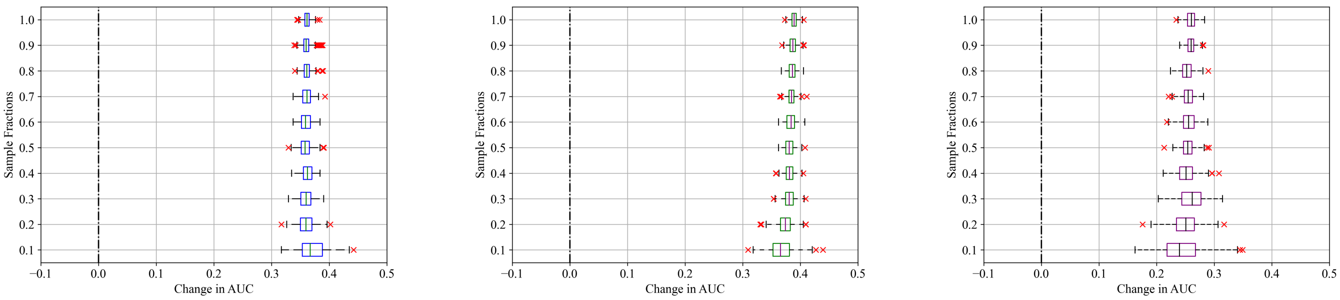


**Figure 2.** *PFI process time and corresponding sample fractions for the Diabetes dataset.*



(a)

(b)

(c)

**Figure 5.** *Sample fractions and corresponding change in AUC for Rad Flow feature of Statlog (Shuttle) dataset; (a) LightGBM, (b) RF, and (c) SVM.*

tion into the influence of sample fractions on feature importance and model performance in datasets characterised by class imbalance. Five real-life datasets with large sample sizes from different subject fields which exhibited high class imbalance ratios of 0.09 – 0.18 were utilised.

Due to its model-agnostic nature, PFI was adopted for feature selection process with feature importance evaluated on Light GBM, RF and SVM. The models were chosen due to their widespread use in real-world ML studies and their role as benchmarks for comparing new models. Cluster, spatial, and stats sub-packages of SciPy were instrumental in tackling the multicollinearity effects associated with PFI. Using a PFI approach, the study revealed the variability of feature importance with smaller sample fractions in LightGBM, random forest and SVM models. In the cases explored, LightGBM showed the lowest variability, while SVM exhibited the highest variability in feature importance. Also, Light GBM had the least CPU process time across the cases considered, while SVM showed the highest computational cost.

In future work, this investigation would be expanded to substantially larger datasets and introduce some quantitative measure of the variability of various models and feature selection methods. An understanding of the variability of feature importance can inform feature engineering efforts that provides means of alleviating the variability of feature importance in samples fractions under class imbalance conditions.

## Acknowledgement

This work was supported by the Petroleum Technology Development Fund under grant PTDF/ED/OSS/PHD/AGU/1076/17 and NISCO UK Research Centre. Computations were performed using the Sulis Tier 2 HPC platform hosted by the Scientific Computing Research Technology Platform at the University of Warwick. Sulis is funded by EPSRC Grant EP/T022108/1 and the HPC Midlands+ consortium.

## References

[1] J. Cai, J. Luo, S. Wang, and S. Yang, “Feature selection in machine learning: A new perspective,” *Neurocomputing*, vol. 300, pp. 70–79, 2018, doi: 10.1016/j.neucom.2017.11.077.

[2] P. Dhal and C. Azad, “A comprehensive survey on feature selection in the various fields of machine learning,” *Applied Intelligence*, 2022, doi: 10.1007/s10489-021-02550-9.

[3] A. Udu, A. Lecchini-Visintini, and H. Dong, *Feature Selection for Aero-Engine Fault Detection*, vol. 3. Cham, Switzerland: Springer Nature, 2023. doi: 10.1007/978-3-031-39847-6_42.

[4] L. Yin, Y. Ge, K. Xiao, X. Wang, and X. Quan, “Feature selection for high-dimensional imbalanced data,” *Neurocomputing*, vol. 105, pp. 3–11, 2013, doi: 10.1016/j.neucom.2012.04.039.

[5] C.-F. Tsai and Y.-T. Sung, “Ensemble feature selection in high dimension, low sample size datasets: Parallel and serial combination approaches,” *Knowledge-Based Systems*, vol. 203, p. 106097, 2020, doi: 10.1016/j.knosys.2020.106097.

[6] A. de Haro-García, G. Cerruela-García, and N. García-Pedrajas, “Ensembles of feature selectors for dealing with class-imbalanced datasets: A proposal and comparative study,” *Information Sciences*, vol. 540, pp. 89–116, 2020, doi: 10.1016/j.ins.2020.05.077.

[7] S. Matharaarachchi, M. Domaratzki, and S. Muthukumarana, “Assessing feature selection method performance with class imbalance data,” *Machine Learning with Applications*, vol. 6, p. 100170, 2021, doi: 10.1016/j.mlwa.2021.100170.

[8] G. Bonaccorso, *Machine Learning Algorithms: Popular algorithms for data science and machine learning*. Packt Publishing Ltd, 2018.

[9] S. Feng, H. Zhou, and H. Dong, “Using deep neural network with small dataset to predict material defects,” *Materials & Design*, vol. 162, pp. 300–310, 2019, doi: 10.1177/07316844241236696.

[10] I. H. Sarker, “Machine learning: Algorithms, real-world applications and research directions,” *SN computer science*, vol. 2, no. 3, p. 160, 2021, doi: 10.1007/s42979-021-00592-x.

[11] A. Paleyes, R.-G. Urma, and N. D. Lawrence, “Challenges in deploying machine learning: a survey of case studies,” *ACM computing surveys*, vol. 55, no. 6, pp. 1–29, 2022, doi: 10.1145/3533378.

[12] A. G. Udu, N. Osa-uwagboe, O. Adeniran, A. Aremu, M. G. Khaksar, and H. Dong, “A machine learning approach to characterise fabrication porosity effects on the mechanical properties of additively manufactured thermoplastic composites,” *Journal of Reinforced Plastics and Composites*, p. 07316844241236696, 2024, doi: 10.1177/07316844241236696.

[13] A. Luque, A. Carrasco, A. Martín, and A. de las Heras, “The impact of class imbalance in classification performance metrics based on the binary confusion matrix,” *Pattern Recognition*, vol. 91, pp. 216–231, 2019, doi: 10.1016/j.patcog.2019.02.023.

[14] M. Temraz and M. T. Keane, “Solving the class imbalance problem using a counterfactual method for data augmentation,” *Machine Learning with Applications*, vol. 9, p. 100375, 2022, doi: 10.1016/j.mlwa.2022.100375.

[15] The Pandas Development Team, “pandas-dev/pandas: Pandas.” Zenodo, 2020. doi: 10.5281/zenodo.3509134.

[16] C. R. Harris *et al.*, “Array programming with NumPy,” *Nature*, vol. 585, no. 7825, pp. 357–362, 2020, doi: 10.1038/s41586-020-2649-2.

[17] F. Pedregosa *et al.*, “Scikit-learn: Machine Learning in Python,” *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[18] J. D. Hunter, “Matplotlib: A 2D graphics environment,” *Computing in Science & Engineering*, vol. 9, no. 3, pp. 90–95, 2007, doi: 10.1109/MCSE.2007.55.

[19] P. Virtanen *et al.*, “SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python,” *Nature Methods*, vol. 17, pp. 261–272, 2020, doi: 10.1038/s41592-019-0686-2.

[20] M. Kelly, R. Longjohn, and K. Nottingham, “The UCI Machine Learning Repository [Data set].” [Online]. Available: https://archive.ics.uci.edu/

[21] R. Kohavi, “Census Income.” UCI Machine Learning Repository, 1996. doi: 10.24432/C5GP7S.

[22] Moro and P. Cortez, “Bank Marketing.” UCI Machine Learning Repository, 2012. doi: 10.24432/C5K306.

[23] “Statlog (Shuttle).” UCI Machine Learning Repository. doi: 10.24432/C5WS31.

[24] N. Osa-uwagboe, A. G. Udu, V. V. Silberschmidt, K. P. Baxevanakis, and E. Demirci, “Effects of seawater on mechanical performance of composite sandwich structures: A machine learning framework,” *Materials*, vol. 17, no. 11, p. 2549, 2024, doi: 10.3390/ma17112549.

[25] A. Udu, A. Lecchini-Visintini, M. Ghalati, and H. Dong, “Addressing Class Imbalance in Aeroengine Fault Detection,” in *2023 International Conference on Machine Learning and Applications (ICMLA)*, 2023, pp. 1072–1077. doi: 10.1109/ICMLA58977.2023.00159.

[26] S. Rezvani and X. Wang, “A broad review on class imbalance learning techniques,” *Applied Soft Computing*, vol. 143, p. 110415, 2023, doi: 10.1016/j.asoc.2023.110415.

[27] A. G. Udu, A. Lecchini-Visintini, and H. Dong, “On chance performance in high-dimensional class-imbalance problems,” in *2024 UKACC 14th International Conference on Control (CONTROL)*, 2024, pp. 254–255. doi: 10.1109/CONTROL60310.2024.10531841.

[28] J. Li *et al.*, “Feature selection: A data perspective,” *ACM Computing Surveys*, vol. 50, no. 6, p. Article94, 2017, doi: 10.1145/3136625.

[29] H. Kaneko, “Cross-validated permutation feature importance considering correlation between features,” *Analytical Science Advances*, vol. 3, no. 9–10, pp. 278–287, 2022, doi: 10.1002/ansa.202200018.

Open Access | https://doi.org/10.25080/TWCF2755

SciPy 2024
July 8 - July 14, 2024

Proceedings of the 23rd
Python in Science Conference
ISSN: 2575-9752

# Algorithms to Determine Asteroid's Physical Properties using Sparse and Dense Photometry, Robotic Telescopes and Open Data

Arushi Nath[1]

[1] Founder, MonitorMyPlanet.com

**Abstract**

The rapid pace of discovering asteroids due to advancements in detection techniques outpaces current abilities to analyze them comprehensively. Understanding an asteroid's physical properties is crucial for effective deflection strategies and improves our understanding of the solar system's formation and evolution. Dense photometry provides continuous time-series measurements valuable for determining an asteroid's rotation period, yet is limited to a singular phase angle. Conversely, sparse photometry offers non-continuous measurements across multiple phase angles, essential for determining an asteroid's absolute magnitude, albedo (reflectivity), and size. This paper presents open-source algorithms that integrate dense photometry from citizen scientists with sparse photometry from space and ground-based all-sky surveys to determine asteroids' albedo, size, rotation, strength, and composition. Applying the algorithms to the Didymos binary asteroid, combined with data from GAIA, the Zwicky Transient Facility, and ATLAS photometric sky surveys, revealed Didymos to be 840 meters wide, with a 0.14 albedo, an 18.14 absolute magnitude, a 2.26-hour rotation period, rubble-pile strength, and an S-type composition. Didymos was the target of the 2022 NASA Double Asteroid Redirection Test (DART) mission. The algorithm successfully measured a 35-minute decrease in the mutual orbital period following the DART mission, equating to a 40-meter reduction in the mutual orbital radius, proving a successful deflection. Analysis of the broader asteroid population highlighted significant compositional diversity, with a predominance of carbonaceous (C-type) asteroids in the outer regions of the asteroid belt and siliceous (S-type) and metallic (M-type) asteroids more common in the inner regions. These findings provide insights into the diversity and distribution of asteroid compositions, reflecting the conditions and processes of the early solar system. This work empowers citizen scientists to become planetary defenders, contributing significantly to planetary defense and enhancing our understanding of solar system composition and evolution.

**Published** Jul 10, 2024

**Correspondence to**
Arushi Nath
Arushi@monitormyplanet.com

**Keywords** Near-Earth Asteroids, Python Algorithms, Open Data, Citizen Scientists, Asteroid Characterization

## 1. Introduction

### 1.1. Background

There are over 1.3 million known asteroids, and advanced detection techniques lead to the discovery of hundreds of new near-Earth and main-belt asteroids every month. Studying these asteroids provides valuable insights into the early solar system's formation and evolution. Phase curves, which illustrate the change in an asteroid's brightness as its phase angle (the angle between the observer, asteroid, and Sun) changes, are essential for asteroid characterization. Understanding near-Earth asteroids is crucial because it allows for the development of effective deflection strategies, which are vital for preventing potential collisions with Earth and safeguarding our planet from catastrophic impacts.

### *1.2. Research Problem*

Despite advancements in detection techniques, the need for observations spanning multiple years, limited telescope availability, and narrow observation windows hinder detailed characterization of asteroids. To date, phase curves have been generated for only a few thousand asteroids. This slow pace of analysis hinders our planetary defense capabilities for deflecting potentially hazardous asteroids and limits our understanding of the solar system’s evolution.

### *1.3. Related Work*

Recent efforts in the field have focused on various approaches to combine dense and sparse photometric datasets. For instance, the Pan-STARRS survey has used sparse photometry to estimate the absolute magnitudes and rotation periods of asteroids, while the Asteroid Terrestrial-impact Last Alert System (ATLAS) provides sparse photometry data for many asteroids observed across different phase angles. Studies by V. G. Shevchenko, E. F. Tedesco, L. O. Kovalchuk, A. M. Fiacconi, and V. A. Zubarev [1] have explored methods to derive phase integrals and geometric albedos from sparse data. On the dense photometry side, projects like the Zwicky Transient Facility (ZTF) and Gaia Data Release 3 (DR3) have contributed extensive datasets valuable for continuous observations of asteroid brightness variations. However, these efforts often face challenges in data integration due to differing observational cadences, filters, and coverage.

### *1.4. Research Objectives*

This paper presents an innovative methodology, PhAst, developed using Python algorithms to combine dense photometry from citizen scientists with sparse photometry from space and ground-based all-sky surveys to determine key physical characteristics of asteroids. The specific objectives of this research are to:

1. Develop Python algorithms to integrate serendipitous asteroid observations with citizen-contributed and open datasets.
2. Apply these algorithms to planetary defense tests, such as NASA’s DART mission.
3. Characterize large populations of asteroids to infer compositional diversity in our solar system.

### *1.5. Significance of the Study*

This study offers significant contributions to the field of asteroid characterization and planetary defense. By integrating dense and sparse photometry, the PhAst algorithm provides a comprehensive method for determining the physical properties of asteroids. The open-source nature of the algorithm encourages collaboration and improvements from a global community of researchers and citizen scientists, enhancing its robustness and accelerating advancements in the field. Furthermore, the study empowers citizen scientists to actively participate in planetary defense, contributing valuable data and insights that enhance our understanding and preparedness for potential asteroid impacts.

### *1.6. Application in Planetary Defense: NASA’s DART Mission*

The NASA Double Asteroid Redirection Test (DART) mission was designed to test and validate methods to protect Earth from hazardous asteroid impacts by demonstrating the kinetic impactor technique. It involved sending a spacecraft to collide with an asteroid to change its trajectory. PhAst provides a detailed pre- and post-impact analysis of the target asteroid, Didymos, and its moonlet, Dimorphos.

## 2. Methodology

### 2.1. Overview

The PhAst algorithm integrates dense and sparse photometry data to determine the physical properties of asteroids. Dense photometry provides continuous time-series measurements, crucial for determining rotation periods, while sparse photometry offers non-continuous measurements across multiple phase angles, essential for absolute magnitude and size determination. Integrating both methods can overcome their individual limitations.

### 2.2. Development of Novel Open-Source PhAst

PhAst integrates several years of sparse photometry from serendipitous asteroid observations with dense photometry from professional and citizen scientists Figure 1. The algorithm effectively combines continuous light data (dense photometry) and infrequent light data (sparse photometry) by creating phase curves whose linear components yield the asteroid's geometric albedo and composition, while the non-linear brightness surge at small angles determines the absolute magnitude. This methodology allows for the creation of folded light curves to measure the asteroid's rotation period and, for binary asteroids, their mutual orbital period. Being open-source, the PhAst algorithm allows for collaboration and improvements from a global community of researchers and citizen scientists, enhancing its robustness and accelerating advancements in asteroid characterization.

### 2.3. Data Sources and Integration

1. **Primary Asteroid Observations Using Robotic Telescopes:** Observation proposals were submitted to Alnitak Observatory, American Association of Variable Star Observers, Burke Gaffney Observatory, and Faulkes Telescope.
2. **Citizen Scientist Observations:** Observations submitted by backyard astronomers from locations such as Chile and the USA.
3. **Serendipitous Asteroid Observations in Sky Surveys:** Data from European Space Agency Gaia Data Release 3 and Zwicky Transient Facility (ZTF) Survey.
4. **Secondary Asteroid Databases:** Data from the Asteroid Lightcurve Database (ALCDEF) ALCDEF [2] and Asteroid Photometric Data Catalog (PDS) 3rd update.

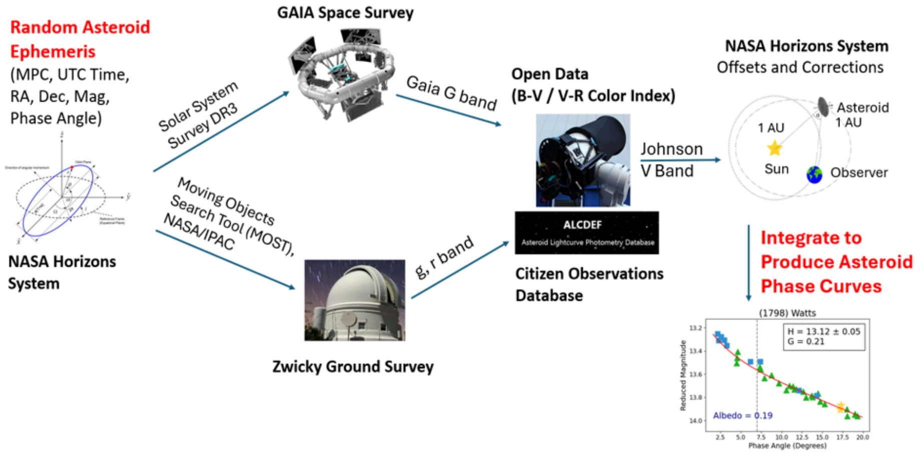


**Figure 1.** *Flowchart Showing Data Integration Process of PhAst*

For searching asteroids in the ZTF dataset, the FINKS portal was utilized, which allowed searching asteroids by their Minor Planet Center (MPC) number. Similarly, asteroids in the GAIA dataset were searched using the Solar System Objects database of Gaia DR3.

### 2.4. *Observational Process*

1. **Identify Known Stars and Asteroids:** Using the GAIA Star Catalog and HORIZONS Asteroid Catalog, known stars and asteroids are identified and centroided in images. This step ensures that the exact positions of celestial objects are accurately determined, which is crucial for subsequent analysis.
2. **Determine Optimal Aperture Size:** Differential photometry is used to calculate the asteroid's instrumental magnitude by determining the optimal aperture size that balances brightness measurement and noise. Too small an aperture may not capture the full brightness of the asteroid, while too large an aperture may include excessive background noise.
3. **Select Suitable Comparison Stars:** Comparison stars with stable brightness are selected to remove the effects of seeing conditions and determine the asteroid's computed magnitude. This step is important to ensure that variations in observed brightness are due to the asteroid itself and not due to atmospheric conditions or instrumental errors.
4. **Remove Distance Effects:** Effects of the asteroid's changing distance from the Sun and the observer are removed to find the reduced magnitude. This normalization allows for a more accurate comparison of observations taken at different times and distances.
5. **Generate Phase Curves:** Phase angle effects are removed to generate phase curves, which help find the absolute magnitude and linear slope. These phase curves provide insights into the reflectivity and surface properties of the asteroid.
6. **Determine Rotation and Orbital Periods:** Composite light curves are used to find the asteroid's rotation period and, for binary asteroids, the mutual orbital period. This analysis reveals the dynamic characteristics of the asteroid, including its spin state and orbital interactions with companion bodies.

### 2.5. *Python Tools and Libraries*

The development and implementation of PhAst heavily relied on various Python tools and libraries:

- **Python:** The primary programming language used for developing PhAst.
- **NumPy:** Used for numerical computations and handling large datasets efficiently.
- **Matplotlib:** Utilized for plotting phase curves and light curves, visualizing the data, and generating graphs for analysis.
- **AstroPy:** Employed for astronomical calculations and handling astronomical data, such as coordinate transformations and time conversions.

## 3. Case Study: Didymos Binary Asteroid

### 3.1. *Initial Observations*

The Didymos binary asteroid, targeted by NASA's 2022 Double Asteroid Redirection Test (DART) mission, was selected for a detailed case study. Initial observations determined Didymos to be 840 meters wide, with a 0.14 albedo, an 18.14 absolute magnitude (a measure of its intrinsic brightness), a 2.26-hour rotation period, rubble-pile strength (indicating it is a loose collection of rocks held together by gravity), and an S-type composition (indicating it is made of stony or siliceous minerals). These properties were derived by applying the PhAst algorithm to a combination of dense and sparse photometric data.

### 3.2. *Impact Analysis*

PhAst successfully measured a 35-minute decrease in the mutual orbital period following the DART mission's impact. External sources validated these findings, demonstrating the algorithm's accuracy and reliability. The change in the mutual orbital period provided critical data on the effectiveness of the DART mission in altering the asteroid's trajectory, a key goal of planetary defense strategies.

## 4. Results

PhAst was used to generate phase curves for over 2100 asteroids in 100 hours on a home computer, including data-retrieval time. The physical properties of various target asteroids of space missions and understudied asteroids were determined, including targets of the NASA LUCY Mission, UAE Mission, binary asteroids, and understudied asteroids Figure 2. The rapid analysis capability highlights PhAst's potential for large-scale asteroid characterization, enabling detailed studies of large populations of asteroids in a relatively short time.

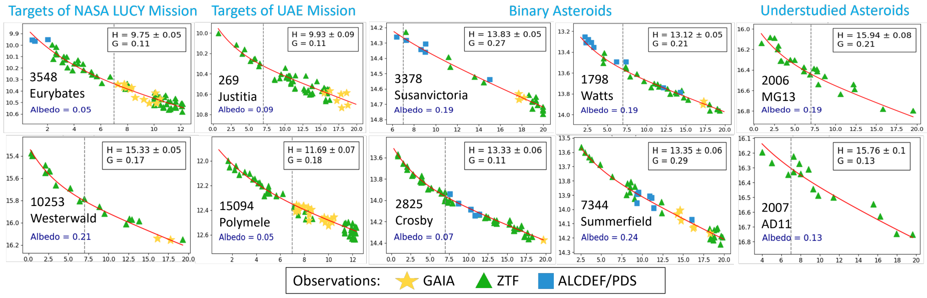
### 4.1. *Determining Physical Properties of Target Asteroids of Space Missions and Understudied Asteroids*

PhAst was used to generate phase curves and determine the physical properties of various target asteroids of space missions and understudied asteroids. The results include:

#### 4.1.1. *NASA LUCY Mission Targets:*

The NASA LUCY mission aims to explore Trojan asteroids, which share Jupiter's orbit around the Sun. Understanding these asteroids can provide insights into the early solar system since Trojans are considered remnants of the primordial material that formed the outer planets.

**3548 Eurybates** Absolute Magnitude (H) = 9.75 ± 0.05Slope Parameter (G) = 0.11Albedo = 0.05Relevance: Eurybates is the largest and presumably the most ancient member of the Eurybates family, offering a window into the conditions of the early solar system.



| Asteroid Physical Properties | 3548 Eurybates | 10253 Westerwald | 269 Justitia | 15094 Polymele | 3378 Susanvictoria | 2825 Crosby | 1798 Watts | 7344 Summerfield | 2006 MG13 | 2007 AD11 |
|---|---|---|---|---|---|---|---|---|---|---|
| Absolute Magnitude | 9.75 | 15.33 | 9.93 | 11.69 | 13.83 | 13.33 | 13.12 | 13.35 | 15.94 | 15.76 |
| Geometric Albedo | 0.05 | 0.21 | 0.09 | 0.05 | 0.19 | 0.07 | 0.19 | 0.24 | 0.19 | 0.13 |
| Size (kilometres) | 67 | 2.5 | 46 | 27 | 5.2 | 11 | 7.2 | 5.8 | 2 | 2.6 |
| Rotation Period (hrs) | 8.72 | - | 11.13 | 5.8 | 2.56 | 3.51 | 3.45 | 2.59 | No data | No data |
| Mutual Orbital Period (hrs) | These are not binary asteroids | | | | 17.2 | 25.56 | 26.21 | 14.43 | No data | No data |
| Strength | Rubble | No data | Rubble | Rubble | Rubble | Rubble | Rubble | Rubble | No data | No data |

**Figure 2.** *Physical Properties of Target Asteroids of Space Missions and Understudied Asteroids Determined*

**10253 Westerwald** Absolute Magnitude (H) = 15.33 ± 0.05Slope Parameter (G) = 0.17Albedo = 0.21Relevance: Westerwald's high albedo suggests it might be a fragment from a larger parent body, providing clues about collisional processes in the early solar system.

#### 4.1.2. *UAE Mission Targets:*

The UAE space mission to explore asteroids aims to study their composition, structure, and history, contributing to our understanding of asteroid formation and the evolution of the solar system.

**269 Justitia** Absolute Magnitude (H) = 9.93 ± 0.09Slope Parameter (G) = 0.11Albedo = 0.09Relevance: Justitia's relatively low albedo indicates a carbonaceous composition, which can help researchers understand the distribution of organic materials in the solar system.

**15094 Polymele** Absolute Magnitude (H) = 11.69 ± 0.07Slope Parameter (G) = 0.18Albedo = 0.05Relevance: Polymele's properties suggest it is a primitive body, providing valuable information about the early solar system's building blocks.

#### 4.1.3. *Binary Asteroids:*

Understanding binary asteroids, where two asteroids orbit each other, can offer insights into the formation and evolutionary history of these systems. The mutual orbital period and other physical properties provide data on their dynamics and interactions.

**3378 Susanvictoria** Absolute Magnitude (H) = 13.83 ± 0.05Slope Parameter (G) = 0.27Albedo = 0.19Relevance: Studying binary systems like Susanvictoria helps in understanding the processes that lead to the formation of binary asteroids and their subsequent evolution.

**2825 Crosby** Absolute Magnitude (H) = 13.33 ± 0.06Slope Parameter (G) = 0.11Albedo = 0.07Relevance: Crosby's characteristics can provide insights into the collisional history and mechanical properties of binary asteroid systems. The physical properties of the binary asteroids were submitted to the binary asteroid working group.

#### 4.1.4. *Understudied Asteroids:*

Characterizing understudied asteroids expands our knowledge of the diversity and distribution of asteroid properties in the solar system.

**2006 MG13** Absolute Magnitude (H) = 15.94 ± 0.08Slope Parameter (G) = 0.21Albedo = 0.19Relevance: Detailed study of asteroids like 2006 MG13 helps fill gaps in our understanding of the physical and compositional diversity of asteroids.

**2007 AD11** Absolute Magnitude (H) = 15.76 ± 0.11Slope Parameter (G) = 0.13Albedo = 0.13Relevance: Investigating such understudied bodies contributes to a more complete picture of asteroid population characteristics and their evolutionary paths.

## 5. Discussions

### 5.1. *Determining the Success of Asteroid Deflection*

The success of the DART mission was evaluated by analyzing the change in the orbital path of Dimorphos, the moonlet of Didymos, after deflection. Applying Kepler's Third Law, the pre-impact orbital period of 11.91 hours and post-impact orbital period of 11.34 hours were used to calculate an orbital radius change of 0.04 km. This change confirms the effectiveness of the DART mission in altering the asteroid's trajectory, a crucial component of planetary defense.

### *5.2. Determining Asteroid Strength*

Asteroid strength can be inferred from the rotation period. This inference is based on the fact that an asteroid's structural integrity must be sufficient to withstand the centrifugal forces generated by its rotation. If the rotation period is less than 2.2 hours, the asteroid must be a strength-bound single rock; otherwise, it would fly apart due to centrifugal forces exceeding the gravitational binding forces. This criterion is supported by studies such as those by P. Pravec and A. W. Harris [3], who observed that most asteroids with rotation periods shorter than 2.2 hours are smaller than 150 meters and are likely monolithic. For larger asteroids, the rubble-pile structure is held together by self-gravity rather than cohesive forces, making them prone to disaggregation at faster rotation rates. This information is vital for assessing the structural integrity of asteroids and planning deflection missions.

### *5.3. Determining Asteroid Taxonomy*

Asteroid taxonomy (chemical composition) can be determined from geometric albedo. C-type asteroids have lower albedo, S-type and M-type asteroids have moderate albedo, and rare E-type asteroids have the highest albedo. (S-type asteroids are made of stony or siliceous minerals, while C-type and M-type refer to carbonaceous and metallic compositions, respectively.) The taxonomic distribution provides insights into the conditions of the early solar system based on the spatial distribution of asteroid types. Understanding these compositions helps in determining the origins and evolutionary history of these asteroids.

### *5.4. Early Solar System Conditions*

The taxonomical distributions of carbonaceous, siliceous, and metallic asteroids in the main belt were compiled. Over 58% of the asteroids characterized by PhAst are carbonaceous, showing they are the most abundant type in our Solar System. Their abundance increases with distance from the Sun, reaching nearly 75% in the outer region of the main belt compared to over 45% in the inner region Figure 3. This finding is consistent with previous research in the field, such as studies by F. E. DeMeo and B. Carry [4], which indicate that carbonaceous asteroids are prevalent in the outer asteroid belt. Characterizing asteroid populations helps us better understand the diversity of compositions in the solar system by providing a detailed inventory of the different types of asteroids and their distribution. This information is crucial for several reasons:

- **Formation Conditions:** Different types of asteroids formed under varying conditions in the early solar system. For example, carbonaceous (C-type) asteroids, which are rich in organic compounds, are more prevalent in the outer regions of the asteroid belt, suggesting formation in cooler, volatile-rich environments. In contrast, siliceous (S-type) and metallic (M-type) asteroids are more common in the inner regions, indicating formation in hotter, more metal-rich conditions.
- **Evolutionary Processes:** By studying the physical and chemical properties of asteroids, we can infer the processes that have shaped their evolution. This includes understanding how collisions, thermal processes, and space weathering have affected their surfaces and internal structures.

### *5.5. Errors and Limitations*

Photometry was performed on images with a Signal-to-Noise Ratio (SNR) > 100, yielding a measurement uncertainty of 0.01. The average error in phase curve fitting was 0.10. Limited processing power restricted the preciseness of the best fit for rotation and mutual orbital periods to two significant digits. These limitations highlight the need for more powerful computational resources and more precise observational data to improve the accuracy of asteroid characterization.

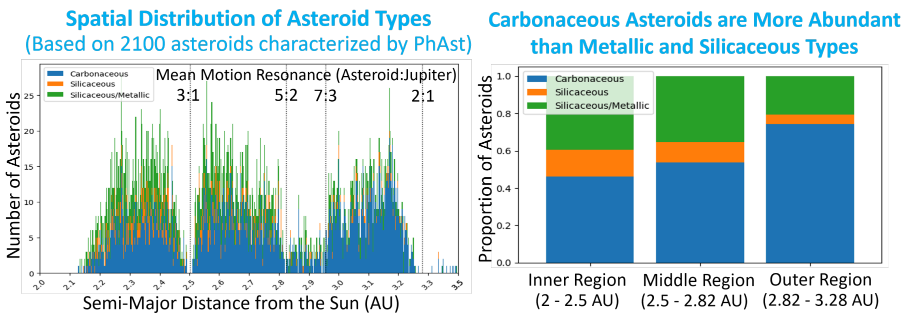


**Figure 3**. *Spatial Distribution of Asteroid Types*

## 6. Conclusions

PhAst represents a significant advancement in asteroid characterization, combining dense and sparse photometry to yield comprehensive insights into asteroid properties. The successful application of PhAst to the Didymos binary asteroid and over 2100 other asteroids demonstrates its potential for large-scale use. By engaging citizen scientists, we can accelerate asteroid analysis and enhance our planetary defense strategies.

## 7. Future Work

PhAst will serve as a powerful tool for accelerating the analysis of data produced by the Legacy Survey of Space and Time (LSST), set to begin in 2025. Over a decade, LSST aims to observe over 5 million asteroids across various filters, generating a nightly data volume of 20TB. The specific benefits and new opportunities that PhAst's applications might bring include:

**Enhanced Planetary Defense** By rapidly characterizing large populations of asteroids, including potentially hazardous asteroids (PHAs), PhAst can provide detailed analysis that are crucial for developing effective deflection strategies, thereby enhancing planetary defense capabilities.

**Comprehensive Asteroid Mapping** The integration of dense and sparse photometry allows for the creation of more accurate and comprehensive maps of asteroid distributions and compositions in the solar system. This can provide valuable insights into the formation and evolution of the solar system, aiding both scientific research and educational initiatives.

**Resource Identification and Utilization** PhAst's ability to determine the physical and compositional properties of asteroids can aid in identifying asteroids rich in valuable minerals or water. This opens up new opportunities for asteroid mining and resource utilization, which could support long-term space exploration and the development of space infrastructure.

**Support for Future Space Missions** PhAst can be used to provide detailed pre and post mission characterization of target asteroids for upcoming space missions including NASA's OSIRIS-APEX which will fly-by near-Earth asteroid Apophis on April 23, 2029, JAXA's Hayabusa2 SHARP to explore two asteroids, 2001 CC21 and 1998 KY26, and China's first kinetic impact deflection test mission would target the near-Earth asteroid 2015 XF261 with a launch in 2027.

**Exoplanetary Atmosphere Characterization** PhAst can be expanded to exoplanetary atmosphere characterization by adapting its methodology to analyze the light curves from transiting exoplanets in multiple filters. This expansion would allow researchers to study the atmospheres of distant planets, providing insights into their composition, climate, and potential habitability.

**Citizen Science and Public Engagement** By making PhAst open-source and developing training modules for citizen scientists, the project promotes public engagement in scientific research. This democratization of science enables a wider community to contribute to and benefit from cutting-edge research, fostering a culture of curiosity and collaboration.

## 8. Project Impact

The PhAst algorithm has been made open-source, and training modules have been developed for citizen scientists. These modules, created using Jupyter Notebooks, are designed for use by high school students and citizen scientists to support their engagement in asteroid characterization and planetary defense. Training on using open data for asteroid categorization has been provided to over 1,500 students during "Space Day" and "Asteroid Day" events in collaboration with observatories and community organizations such as Royal Astronomical Society of Canada. See link to Github: https://github.com/Spacegirl123/Asteroid-Characterization-By-PhAst

### Acknowledgments

The development and application of PhAst have been possible thanks to contributions from numerous observatories, citizen scientists, and research institutions. Special thanks to the teams behind GAIA, Zwicky Transient Facility (ZTF), ATLAS, and other photometric surveys for providing the data that made this research possible. I also acknowledge the support of various citizen science communities and educational organizations for their collaboration and participation.

### References

[1] V. G. Shevchenko, E. F. Tedesco, L. O. Kovalchuk, A. M. Fiacconi, and V. A. Zubarev, "Phase integral of asteroids," *Astronomy & Astrophysics*, vol. 626, p. A87, 2019, doi: 10.1051/0004-6361/201935588.

[2] ALCDEF, "Asteroid Lightcurve Photometry Database." [Online]. Available: https://alcdef.org/

[3] P. Pravec and A. W. Harris, "Fast and Slow Rotation of Asteroids," *Icarus*, vol. 148, no. 1, pp. 12–20, 2000, doi: 10.1006/icar.2000.6482.

[4] F. E. DeMeo and B. Carry, "Solar System evolution from compositional mapping of the asteroid belt," *Nature*, vol. 505, no. 7485, pp. 629–634, 2014, doi: 10.1038/nature12908.

Open Access | https://doi.org/10.25080/DHKD1726

# AI-Driven Watermarking Technique for Safeguarding Text Integrity in the Digital Age

Atharva Rasane[1]

[1]KLE Technology University

SciPy 2024

July 8 - July 14, 2024

Proceedings of the 23rd Python in Science Conference

ISSN: 2575-9752

Published Jul 10, 2024

Correspondence to
Atharva Rasane
rratharva@gmail.com

## Abstract

The internet's growth has led to a surge in text usage. Now, with public access to generative AI models like ChatGPT/Bard, identifying the source is vital. This is crucial due to concerns about copyright infringement and plagiarism. Moreover, it is essential to differentiate AI-generated text to curb misinformation from AI model hallucinations.

In this paper, we explore text watermarking as a potential solution, focusing on plain ASCII text in English. We investigate techniques including physical watermarking (e.g., UniSpaCh by Por et al.), which modifies text to hide a binary message using Unicode Spaces, and logical watermarking (e.g., word context by Jalil et al.), which generates a watermark key via a defined process. While logical watermarking is difficult to break but undetectable without prior knowledge, physical watermarks are easily detected but also easy to break.

This paper presents a unique physical watermarking technique based on word substitution to address these challenges. The core idea is that AI models consistently produce the same output for the same input. Initially, we replaced every i-th word (for example, every 5th word) with a "[MASK]," a placeholder token used in natural language processing models to indicate where a word has been removed and needs to be predicted. Then, we used a BERT model to predict the most probable token in place of "[MASK]." The resulting text constitutes the watermarked text. To verify, we reran the algorithm on the watermarked text and compared the input and output for similarity.

The Python implementation of the algorithm in this paper employs models from the HuggingFace Transformer Library, namely "bert-base-uncased" and "distilroberta-base". The "[MASK]" placeholder was generated by splitting the input string using the split() function and then replacing every 5th element in the list with "[MASK]". This modified list served as the input text for the BERT model, where the output corresponding to each "[MASK]" was replaced accordingly. Finally, applying the join() function to the list produces the watermarked text.

This technique tends to generate nearly invisible watermarked text, preserving its integrity or completely changing the meaning of the text based on how similar the text is to the training dataset of BERT. This was observed when the algorithm was run on the story of Red Riding Hood, where its meaning was altered. However, the nature of this watermark makes it extremely difficult to break due to the black-box nature of the AI model.

Keywords physical watermark, logical watermark, HuggingFace Transformer Library, BERT

## 1. Introduction

The growth of the internet is driven by the spread of web pages, which are written in HTML (Hyper Text Markup Language). These web pages contain large amounts of text. Almost every webpage, in some form or another, contains text, making it a popular mode of communication, whether it be blogs, posts, articles, comments, etc. Text can be represented as a collection of ASCII or Unicode values, where each value corresponds to a specific character. Given the text-focused nature of the internet and tools like ChatGPT or Bard, it is crucial to identify the source of text. This helps to manage copyright issues and distinguish between AI-generated and human-written text, thereby preventing the spread of misinformation. Currently, detecting AI-generated text relies on machine learning classifiers that need frequent retraining with the latest AI-generated data. However, this method has drawbacks, such as the rapid evolution of AI models producing increasingly human-like text. Therefore, a more stable approach is needed, one that does not depend on the specific AI model generating the text.

Watermarks are an identifying pattern used to trace the origin of the data. In this case, we specifically want to focus on text watermarking (watermarking of plain text). Text watermarking can broadly be classified into 2 types, Logical Embedding, and Physical Embedding, which in turn can be classified further [1]. Logical Embedding involves the user generating a watermark key by some logic from the input text. Note that this means that the input text is not altered, and the user instead keeps the generated watermark key to identify the text. Physical Embedding involves the user altering the input text itself to insert a message into it, and the user instead runs an algorithm to find this message to identify the text. In this paper, we will propose an algorithm to watermark text using BERT (Bidirectional Encoder Representations from Transformers), a model introduced by Google, whose main purpose is to replace a special symbol “[MASK]” with the most probable word given the context.

BERT (Bidirectional Encoder Representations from Transformers) is a pre-trained model introduced by Google in 2018, which has revolutionized natural language processing (NLP) [2]. “Pre-trained” means the model has already been trained on a large dataset before being fine-tuned for specific tasks. This allows the model to learn general features and patterns from a broad range of text data. For BERT, this pre-training involves vast amounts of text from books, articles, and websites, enabling it to understand the intricacies of human language. This pre-training allows BERT to be adapted quickly to various NLP tasks with relatively small amounts of task-specific data. Traditional models read text sequentially, either left-to-right or right-to-left. In contrast, BERT reads text in both directions simultaneously, providing a deeper understanding of context and meaning. This bidirectional approach allows BERT to perform exceptionally well in various NLP tasks, including question answering, text classification, and named entity recognition. By grasping the nuances of language more effectively, BERT sets a new standard for accuracy and efficiency in NLP applications [2].

At its core, BERT employs a bi-directional Transformer encoder, which helps understand the relationships between words in a sentence. This enhances its comprehension of text by understanding context from both directions simultaneously. BERT undergoes pre-training through two tasks: Masked Language Modeling (MLM), where certain words in a sentence are masked and the model predicts them based on surrounding words, and Next Sentence Prediction (NSP), which involves determining if one sentence logically follows another. This comprehensive training enables BERT to excel in numerous NLP applications like question answering, text classification, and named entity recognition. Given its deep understanding of context and semantics, BERT is highly relevant to text watermarking. Watermarking text involves embedding identifying patterns within the text to trace its origin, which can be critical for copyright protection and distinguishing between AI-generated and human-written content. BERT’s sophisticated handling of language makes it ideal for embedding watermarks in a way that is subtle yet robust, ensuring that the text remains natural while the watermark is detectable. This capability provides a more stable and reliable method for watermarking text, irrespective of the model generating the text, therefore offering a concrete solution amidst the evolving landscape of AI-generated content.

## 2. Related Work

In this section, we will review two text watermarking algorithms before introducing our proposed technique. Let's first look at the current standards for text watermarking. Text watermarking algorithms embed unique identifiers in text to protect copyright and verify authenticity. They are important because they help prevent unauthorized use, copying, and distribution of text.

The first algorithm is Word Context, developed by Z. Jalil and A. M. Mirza [3]. It is a type of logical watermarking that generates a watermark key without altering the original text [3]. Logical watermarking involves embedding a watermark key without changing the original text. Word Context generates a watermark key by analyzing the structure of the text around selected keywords and creating a pattern based on word lengths [3]. In Word Context, a keyword is selected. For example, using the keyword `'is'` in the text `'Pakistan is a developing country, with Islamabad is the capital of Pakistan. It is located in Asia.'` The lengths of the words before and after `'is'` are recorded: `'Pakistan'` (8) and `'a'` (1), `'Islamabad'` (9) and `'the'` (3), `'It'` (2) and `'located'` (7). The watermark is then 8-1-9-3-2-7 [3]. The keyword is chosen based on its significance in the text. Word lengths are used to create the watermark because they provide a unique pattern without altering the text, ensuring the watermark is imperceptible [3].

The second algorithm, UniSpaCh by Kamaruddin et al. in 2018, modifies the white spaces in text to embed a binary message directly into it [4]. Modifying white spaces changes the spacing patterns in the text, embedding binary information. A binary message is a sequence of bits (0s and 1s) that represents data. This method uses different types of spaces to encode these bits [4]. UniSpaCh uses 2-bit categorization to create a binary string (e.g., `'10'`, `'01'`, `'00'`, `'11'`). Each pair of bits is replaced with a unique type of space (like punctuation space, thin space). These spaces are then placed in areas like between words, sentences, and paragraphs. This method is highly invisible but has low capacity, making it unsuitable for embedding long messages [4]. 2-bit categorization assigns pairs of bits to specific types of spaces. This method is considered invisible because the changes are subtle and not easily noticeable by readers. It has low capacity because only a few bits can be embedded per space, limiting the amount of information that can be hidden [4].

The first approach by Z. Jalil and A. M. Mirza [3] is not suitable for today's fast-paced generation of AI text, as it is impractical to store a logical watermark for each new text [3]. It is impractical to store a logical watermark for each text because the volume of generated text is too high, making it difficult to manage and store all watermarks. AI text generation has made it easier and faster to produce large amounts of text, increasing the need for scalable watermarking solutions [3]. The second approach by Por et al. (2012) is also not suitable because the watermark can be easily removed by reformatting the text. We need a robust and imperceptible watermarking technique [4]. The watermark can be removed by reformatting because changes in text layout, such as altering spaces or reformatting paragraphs, can disrupt the embedded watermark. A robust watermarking technique can withstand such changes and remain detectable, while an imperceptible technique ensures the watermark is not noticeable to the reader [4].

Our proposed technique is based on a method by Lancaster (2023) for ChatGPT [5]. It replaces every fifth word in a sequence of five consecutive words (non-overlapping 5-gram) with a word generated using a fixed random seed. For example, in the sentence `'The friendly robot greeted the visitors with a cheerful beep and a wave of its metal arms,'` the non-overlapping 5-grams are `'The friendly robot greeted the,'` `'visitors with a cheerful beep,'` and `'and a wave of its metal.'` We replace the words `'the,'` `'visitors,'` and `'metal'` with words generated by ChatGPT using a fixed random seed [5]. A non-overlapping 5-gram is a sequence of five consecutive words without any overlap. Replacing every fifth word embeds the watermark without altering the overall meaning of the text, making it a subtle and effective method for embedding the watermark [5].

We check the watermark using overlapping 5-grams, which overlap by four words. For example, `'The friendly robot greeted the,' 'friendly robot greeted the visitors,' 'robot greeted the visitors with,'` etc. This method uses ChatGPT to watermark its own text, but it requires running two ChatGPT models to ensure consistency across different outputs from the same seed. Overlapping 5-grams are sequences of five words that overlap by four words. Two models of ChatGPT are needed to ensure consistent watermarking across different outputs because different models might produce different results with the same random seed, and consistency is crucial for verifying the watermark.

We propose using BERT, a model designed to find missing words, as a better alternative to ChatGPT. BERT is more precise and smaller. Its bidirectional nature uses more context for word prediction, potentially leading to better results. While ChatGPT-based algorithms are best for ChatGPT text, BERT can be used for any text, regardless of its origin. BERT is better than ChatGPT for this purpose because it is more precise and smaller, making it more efficient. BERT's bidirectional nature means it uses context from both the preceding and following words to predict a missing word, which can lead to more accurate results.

## 3. Proposed Model

"BERT-based watermarking is based on the 5-gram approach by Lancaster[5]. However, our focus is on watermarking any text, regardless of its origin. This paper will use **bert-base-uncased** model, which finds the most probable uncased English word to replace the [MASK] token.

Note that a different variant of BERT can be trained on different language datasets and thus will generate a different result and as such the unique identity to consider here is the BERT model i.e. if the user wants a unique watermark they need to train/develop the BERT model on their own. This paper is not concerned with the type of BERT model and is focused on its conceptual application for watermarking. Thus for us, BERT is a black box model that returns the most probable word given the context with the only condition being that it has a constant temperature i.e. it does not hallucinate (produce different results for the same input). For our purposes, you can think of the proposed algorithm as a many to one function which is responsible for converting the input text into a subset of watermarked set.

## 4. Algorithm

**Watermark Encoding**

The above is a simple implementation of the algorithm where we are assuming

1. The only white spaces in the text are `" "`.
2. BERT model has infinite context.

This simplified code allows us to grasp the core of the algorithm. First, we split the input text into a list of words using the split() function. Next, we replace every 5th word with the string "[MASK]" which represents a special token indicating where BERT should predict a word. For each [MASK] token, we pass the preceding words and the 4 following words to the BERT model, assuming BERT can handle an infinite context. In reality, BERT has a limited context, so we use up to maximum_context_size - 5 words along with the [MASK] token. The missing_word_form_BERT() function returns the most probable word, which replaces the [MASK] token in the list. We continue this process until all [MASK] tokens are replaced, then convert the list of words back into a string using `" ".join()`.

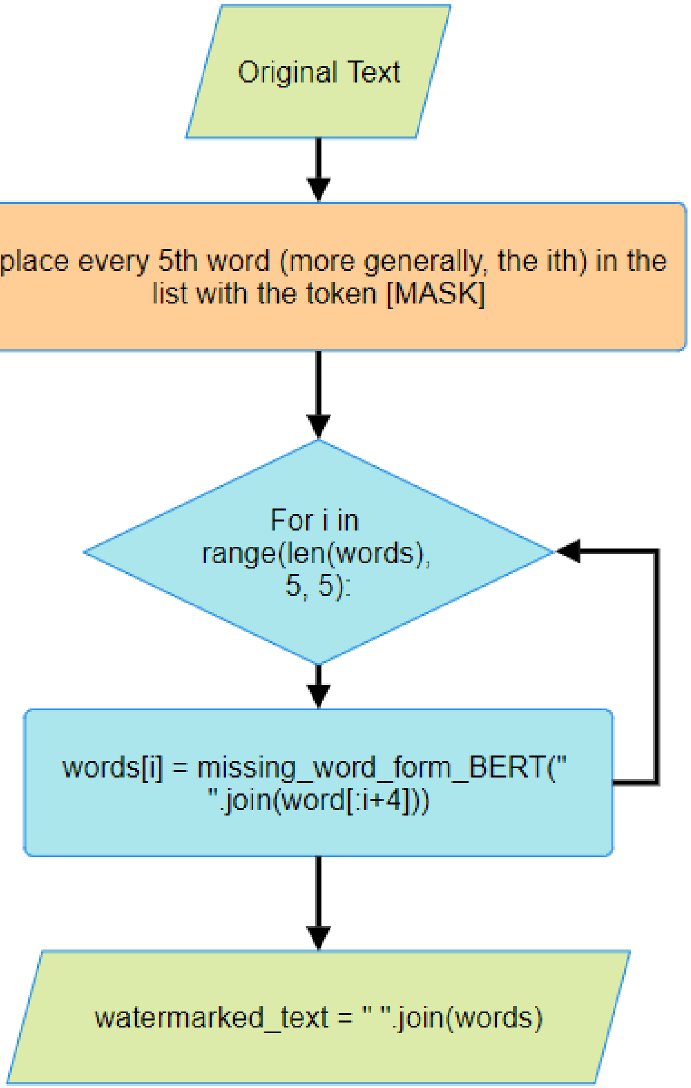

Original Text
Replace every 5th word (more generally, the ith) in the list with the token [MASK]
For i in range(len(words), 5, 5):
words[i] = missing_word_form_BERT(" ".join(word[:i+4]))
watermarked_text = " ".join(words)

**Figure 1**. *Encoding algorithm to watermark input text*

The beauty of the algorithm is that if we were to run it again on the watermarked text the output that we would get would be the same as the input thus to check if a given text is watermarked we simply need to compare the input and output to determine if a given text is watermarked we simply need to run the above algorithm again, but with a few changes we will have to take in offset as a consideration as the one plagiarizing the text might insert additional words that may lead to the text

**Watermark Detection**

The algorithm checks if a given text is watermarked by comparing the input and output texts, considering possible word insertions that may offset the watermark pattern.

1. **Input Text Preparation :** Obtain the suspected watermarked text as input.
2. **Run Watermark Detection Algorithm:** Run the watermark detection algorithm on the input text.

3. **Compare Input and Output:** If the input matches the output, the text is watermarked.If not, proceed to check with offsets.
4. **Offset Consideration:** Initialize an array to store match percentages for each offset: `offsets = [0, 1, 2, 3, 4]`.For each offset, adjust the input text by removing `n % 5` words where `n` is the number of words added.
5. **Check for Matches:** For each offset, count the matches where the watermark pattern (every 5th word replaced) aligns.
6. **Store Match Percentages:** Calculate the percentage of matches for each offset and store them.
7. **Statistical Analysis:** Compute the highest percentage of matches (`Highest Ratio`). Compute the average percentage of matches for the remaining offsets (`Average Others`). Calculate the T-Statistic and P-Value to determine the statistical difference between `Highest Ratio` and `Average Others`. The T-Statistic measures the difference between groups, and the P-Value indicates the significance of this difference.
8. **Classification:** Use a pre-trained model to classify the text based on the metrics (`Highest Ratio`, `Average Others`, T-Statistic, P-Value) as watermarked or not.

## 5. Implementation - Encoding module

Let's examine a Python implementation of the proposed watermarking model. The watermark_text module identifies every 5th word in the input string, splits them using Python's built-in split() function, and marks them for modification using BERT. These placeholders are replaced with the [MASK] token. Although we use BERT here, the module can be adapted to other AI models. We chose BERT due to its efficiency in altering individual words. The 5th word is selected to ensure a consistent and detectable pattern. **The choice of index = 5 is because? Also is this code picked from a paper?**

```python
import os
os.environ['HUGGINGFACEHUB_API_TOKEN'] = '<ENTER_HUGGING_FACE_API_KEY>'
from transformers import pipeline, AutoTokenizer, AutoModelForMaskedLM
import torch

def watermark_text(text, model_name="bert-base-uncased", offset=0):
    # Clean and split the input text
    text = " ".join(text.split())
    words = text.split()

    # Replace every fifth word with [MASK], starting from the offset
    for i in range(offset, len(words)):
        if (i + 1 - offset) % 5 == 0:
            words[i] = '[MASK]'

    # Initialize the tokenizer and model, move to GPU if available
    device = 0 if torch.cuda.is_available() else -1
    tokenizer = AutoTokenizer.from_pretrained(model_name)
    model = AutoModelForMaskedLM.from_pretrained(model_name).to(device)

    # Initialize the fill-mask pipeline
    classifier = pipeline("fill-mask", model=model, tokenizer=tokenizer, device=device)

    # Make a copy of the words list to modify it
    watermarked_words = words.copy()

    # Process the text in chunks
    for i in range(offset, len(words), 5):
        chunk = " ".join(watermarked_words[:i+9])
        if '[MASK]' in chunk:
            try:
                tempd = classifier(chunk)
            except Exception as e:
                print(f"Error processing chunk '{chunk}': {e}")
                continue

            if tempd:
                templ = tempd[0]
                temps = templ['token_str']
                watermarked_words[i+4] = temps.split()[0]

    return " ".join(watermarked_words)

# Example usage
text = "Quantum computing is a rapidly evolving field that leverages the principles of quantum mechanics
to perform computations that are infeasible for classical computers. Unlike classical computers, which
use bits as the fundamental unit of information, quantum computers use quantum bits or qubits. Qubits
can exist in multiple states simultaneously due to the principles of superposition and entanglement,
providing a significant advantage in solving complex computational problems."
watermark_text(text, offset=0)
result = "Quantum computing is a rapidly evolving field that leverages the principles of quantum
mechanics to perform computations that are impossible for classical computers. Unlike quantum computers,
which use bits as the fundamental unit of , quantum computers use quantum bits or qubits. Qubits
can exist in multiple states simultaneously according to the principles of symmetry and entanglement,
providing a significant advantage in solving complex mathematical problems."
```

In the result, the module has replaced each 5th word with the most probable replacement word selected by BERT. There will always be some words that AI would not alter. For example the 10th word “the” and the 15th word “to”. These cannot be changed by AI without altering the entire sentence. Further, to speed up the AI computing, we can employ GPUs in this module as well as the Detection module.

## 6. Implementation - Detection Module

Now that we have our watermarked text, we need to identify potential copyright infringement. We assume this text is what a plagiarizer has access to.

For this, we create a module to check the number of word matches if the AI model with the same offset parameter is run on the watermarked text again. The algorithm’s elegance lies in its consistency: if we run it again on the watermarked text, the output will match the input because the most probable words are already present at every 5th offset. Consequently, we get a 100% match rate with a match ratio of 1. If all the 5th words were altered, our match rate would be 0.

Altering written text is a posibility we cannot ignore. consider a scenario where a plagiarizer might insert extra words, causing the input not to match the output exactly. This means our model needs to check for watermarks not only at a specific index but also in the surrounding words. Therefore, our model needs to check for the watermark at different offsets (0 to 4) to account for potential word insertions.

Here is how the offset works:

- If 1 word is added at the start, the offset is 1.
- If 2 words are added, the offset is 2.
- If 3 words are added, the offset is 3.
- If 4 words are added, the offset is 4.
- If 5 words are added, the offset is 0 (since the algorithm replaces every 5th word).

In general, if ‘n’ words are added, the offset is `n % 5`. Since we do not know how many words were added, we need to check all possible offsets (0, 1, 2, 3, 4).

If words are added in the middle of the text, the majority of the watermark pattern (every 5th word replaced) will still be detectable at some offset. The idea is that one offset will show a higher number of matches compared to others, indicating a watermark.

For detection, we store the percentage of matches for each offset. There is no fixed threshold for determining a watermark, as the choice of words affects the number of matches. For non-watermarked text, the percentage of matches at each offset will be similar. For watermarked text, one offset will have a significantly higher percentage of matches.

The output of “watermark_text_and_calculate_matches” module is a match ratio for offsets 0-4 acting as a seed for the next stage of detection. Below is the python code for generating the list of match ratios.

```python
def watermark_text_and_calculate_matches(text, model_name="bert-base-uncased", max_offset=5):
    # Clean and split the input text
    text = " ".join(text.split())
    words = text.split()

    # Initialize the tokenizer and model, move to GPU if available
    device = 0 if torch.cuda.is_available() else -1
    tokenizer = AutoTokenizer.from_pretrained(model_name)
    model = AutoModelForMaskedLM.from_pretrained(model_name).to(device)

    # Initialize the fill-mask pipeline
    classifier = pipeline("fill-mask", model=model, tokenizer=tokenizer, device=device)

    # Dictionary to store match ratios for each offset
    match_ratios = {}

    # Loop over each offset
    for offset in range(max_offset):
        # Replace every fifth word with [MASK], starting from the offset
        modified_words = words.copy()
        for i in range(offset, len(modified_words)):
            if (i + 1 - offset) % 5 == 0:
                modified_words[i] = '[MASK]'

        # Make a copy of the modified words list to work on
        watermarked_words = modified_words.copy()
        total_replacements = 0
        total_matches = 0

        # Process the text in chunks
        for i in range(offset, len(modified_words), 5):
            chunk = " ".join(watermarked_words[:i+9])
            if '[MASK]' in chunk:
                try:
                    tempd = classifier(chunk)
                except Exception as e:
                    print(f"Error processing chunk '{chunk}': {e}")
                    continue

                if tempd:
                    templ = tempd[0]
                    temps = templ['token_str']
                    original_word = words[i+4]
                    replaced_word = temps.split()[0]
                    watermarked_words[i+4] = replaced_word

                    # Increment total replacements and matches
                    total_replacements += 1
                    if replaced_word == original_word:
                        total_matches += 1

        # Calculate the match ratio for the current offset
        if total_replacements > 0:
            match_ratio = total_matches / total_replacements
        else:
            match_ratio = 0

        match_ratios[offset] = match_ratio

    # Return the match ratios for each offset
    return match_ratios

# Example usage
text = "Quantum computing is a rapidly evolving field that leverages the principles of quantum mechanics
to perform computations that are infeasible for classical computers. Unlike classical computers, which
use bits as the fundamental unit of information, quantum computers use quantum bits or qubits. Qubits
can exist in multiple states simultaneously due to the principles of superposition and entanglement,
providing a significant advantage in solving complex computational problems."

# Calculate match ratios
match_ratios = watermark_text_and_calculate_matches(text, max_offset=5)
# (result rounded) match_ratio = {0: 0.54, 1: 0.62, 2: 0.58, 3: 0.67, 4: 0.58}
```

The final stage of detection involves determining if the match ratios are statistically significant. To determine whether the text is watermarked, we rely on a binary classification of whether a text is watermarked. For this, we use a pre-trained model based on metrics including Highest Ratio, Average Others, T-Statistic, and P-Value. This approach is necessary because, as illustrated in the graphs later, there is no discernible or easily observable difference between the T-statistics and P-values of watermarked and non-watermarked texts. Consequently, we resort to using a pre-trained model for classification, which has achieved the highest accuracy of 94%.

The module `check_significant_difference` generates a list of significance.

```python
from scipy.stats import ttest_1samp
import numpy as np

def check_significant_difference(match_ratios):
    # Extract ratios into a list
    ratios = list(match_ratios.values())

    # Find the highest ratio
    highest_ratio = max(ratios)

    # Find the average of the other ratios
    other_ratios = [r for r in ratios if r != highest_ratio]
    average_other_ratios = np.mean(other_ratios)

    # Perform a t-test to compare the highest ratio to the average of the others
    t_stat, p_value = ttest_1samp(other_ratios, highest_ratio)

    # Print the results
    print(f"Highest Match Ratio: {highest_ratio}")
    print(f"Average of Other Ratios: {average_other_ratios}")
    print(f"T-Statistic: {t_stat}")
    print(f"P-Value: {p_value}")

    # Determine if the difference is statistically significant (e.g., at the 0.05 significance level)
    if p_value < 0.05:
        print("The highest ratio is significantly different from the others.")
    else:
        print("The highest ratio is not significantly different from the others.")

    return [highest_ratio, average_other_ratios, t_stat, p_value]

# Example usage
text = "Quantum computing is a rapidly evolving field that leverages the principles of quantum mechanics
to perform computations that are infeasible for classical computers. Unlike classical computers, which
use bits as the fundamental unit of information, quantum computers use quantum bits or qubits. Qubits
can exist in multiple states simultaneously due to the principles of superposition and entanglement,
providing a significant advantage in solving complex computational problems."
match_ratios = watermark_text_and_calculate_matches(text, max_offset=5)
check_significant_difference(match_ratios)
```

The module `randomly_add_words` was created to simulate the scenario where additional words have been added to the watermarked test for testing purposes.

```python
import random

def randomly_add_words(text, words_to_add, num_words_to_add):
    # Clean and split the input text
    text = " ".join(text.split())
    words = text.split()

    # Insert words randomly into the text
    for _ in range(num_words_to_add):
        # Choose a random position to insert the word
        position = random.randint(0, len(words))
        # Choose a random word to insert
        word_to_insert = random.choice(words_to_add)
        # Insert the word at the random position
        words.insert(position, word_to_insert)

    # Join the list back into a string and return the modified text
    modified_text = " ".join(words)
    return modified_text

# Example usage
text = "Quantum computing is a rapidly evolving field that leverages the principles of quantum mechanics
to perform computations that are infeasible for classical computers. Unlike classical computers, which
use bits as the fundamental unit of information, quantum computers use quantum bits or qubits. Qubits
can exist in multiple states simultaneously due to the principles of superposition and entanglement,
providing a significant advantage in solving complex computational problems."
words_to_add = ["example", "test", "random", "insert"]
num_words_to_add = 5

# modified_text = randomly_add_words(text, words_to_add, num_words_to_add)
modified_text = randomly_add_words(watermark_text(text, offset=0), words_to_add, num_words_to_add)
(result) modified_text = "Quantum computing is example a rapidly evolving field that leverages the
principles of quantum mechanics to perform random computations that are impossible for classical
computers. Unlike quantum computers, which use bits as the random insert fundamental unit of , quantum
computers use quantum bits or qubits. Qubits can exist in multiple states simultaneously according
random to the principles of symmetry and entanglement, providing a significant advantage in solving
complex mathematical problems."

match_ratios = watermark_text_and_calculate_matches(modified_text, max_offset=5)
# (result rounded) match_ratios = {0: 0.57, 1: 0.57, 2: 0.54, 3: 0.38, 4: 0.77}

check_significant_difference(match_ratios)
# (result rounded)
#    Highest Match Ratio: 0.77
#    Average of Other Ratios: 0.52
#    T-Statistic: -5.66
#    P-Value: 0.01
# The highest ratio is significantly different from the others.
```

Once the list of significance is defined, to show the significance of using a pre-trained model, lets plot them to futher understand the statistical summary. Here is the python code used to generate the plots.

```python
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
from scipy.stats import ttest_ind
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import classification_report, confusion_matrix

# Assuming list_of_significance and list_of_significance_watermarked are already defined
# Create DataFrames from the lists
df_significance = pd.DataFrame(list_of_significance, columns=['Highest Ratio', 'Average Others', 'T-Statistic', 'P-Value'])
df_significance_watermarked = pd.DataFrame(list_of_significance_watermarked, columns=['Highest Ratio', 'Average Others', 'T-Statistic', 'P-Value'])

# Add a label column to distinguish between the two sets
df_significance['Label'] = 'Original'
df_significance_watermarked['Label'] = 'Watermarked'

# Combine the DataFrames
combined_df = pd.concat([df_significance, df_significance_watermarked], ignore_index=True)

# Perform EDA
def perform_eda(df):
    # Display the first few rows of the DataFrame
    print("First few rows of the DataFrame:")
    print(df.head())

    # Display statistical summary
    print("\nStatistical Summary:")
    print(df.describe())

    # Check for missing values
    print("\nMissing Values:")
    print(df.isnull().sum())

    # Visualize the distributions of the features
    plt.figure(figsize=(12, 8))
    sns.histplot(data=df, x='Highest Ratio', hue='Label', element='step', kde=True)
    plt.title('Distribution of Highest Ratio')
    plt.show()

    plt.figure(figsize=(12, 8))
    sns.histplot(data=df, x='Average Others', hue='Label', element='step', kde=True)
    plt.title('Distribution of Average Others')
    plt.show()

    plt.figure(figsize=(12, 8))
    sns.histplot(data=df, x='T-Statistic', hue='Label', element='step', kde=True)
    plt.title('Distribution of T-Statistic')
    plt.show()

    plt.figure(figsize=(12, 8))
    sns.histplot(data=df, x='P-Value', hue='Label', element='step', kde=True)
    plt.title('Distribution of P-Value')
    plt.show()

    # Pairplot to see relationships
    sns.pairplot(df, hue='Label')
    plt.show()

    # Correlation matrix
    plt.figure(figsize=(10, 8))
    sns.heatmap(df.drop(columns=['Label']).corr(), annot=True, cmap='coolwarm')
    plt.title('Correlation Matrix')
    plt.show()

    # T-test to check for significant differences
    original = df[df['Label'] == 'Original']
    watermarked = df[df['Label'] == 'Watermarked']

    for column in ['Highest Ratio', 'Average Others', 'T-Statistic', 'P-Value']:
        t_stat, p_value = ttest_ind(original[column], watermarked[column])
        print(f"T-test for {column}: T-Statistic = {t_stat}, P-Value = {p_value}")
```

```
# Perform EDA on the combined DataFrame
perform_eda(combined_df)

# Check if the data is ready for machine learning classification

# Prepare the data
X = combined_df.drop(columns=['Label'])
y = combined_df['Label']

# Convert labels to numerical values for ML model
y = y.map({'Original': 0, 'Watermarked': 1})

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Train a RandomForestClassifier
clf = RandomForestClassifier(random_state=42)
clf.fit(X_train, y_train)

# Make predictions
y_pred = clf.predict(X_test)

# Evaluate the model
print("\nClassification Report:")
print(classification_report(y_test, y_pred))

print("\nConfusion Matrix:")
print(confusion_matrix(y_test, y_pred))

# Feature importances
feature_importances = clf.feature_importances_

# Create a DataFrame for feature importances
feature_importances_df = pd.DataFrame({
    'Feature': X.columns,
    'Importance': feature_importances
}).sort_values(by='Importance', ascending=False)

# Plot feature importances
plt.figure(figsize=(12, 8))
sns.barplot(x='Importance', y='Feature', data=feature_importances_df, palette='viridis')
plt.title('Feature Importances')
plt.show()

# Heatmap for feature importances
plt.figure(figsize=(10, 8))
sns.heatmap(feature_importances_df.set_index('Feature').T, annot=True, cmap='viridis')
plt.title('Heatmap of Feature Importances')
plt.show()
```

The plots are created using the result from our previous example with check_significant_difference returned: Highest Match Ratio: 0.7692307692307693 Average of Other Ratios: 0.5164835164835164 T-Statistic: -5.66220858504931 P-Value: 0.010908789440745323

Missing Values: Highest Ratio 0 Average Others 0 T-Statistic 1 P-Value 1 Label 0 dtype: int64

From the graphs and statistical summaries, several inferences can be drawn regarding the distributions and relationships between the variables in the dataset:

**Distribution of Highest Ratio:** The distribution of the “Highest Ratio” variable shows a clear distinction between the “Original” and “Watermarked” categories. The “Original” category has a peak around 0.4, while the “Watermarked” category peaks around 0.5, indicating a shift in the distribution towards higher values for the watermarked data.

**Distribution of Average Others:** Similarly, the “Average Others” variable shows a distinction between the two categories. The “Original” category peaks around 0.3, whereas the “Watermarked” category peaks slightly higher, around 0.4. This suggests that the average

**Table 1.** *First few rows of the DataFrame*

| | Highest Ratio | Average Others | T-Statistic | P-Value | Label |
|---|---|---|---|---|---|
| 0 | 0.233333 | 0.182203 | -3.532758 | 0.038563 | Original |
| 1 | 0.203390 | 0.139195 | -3.440591 | 0.041218 | Original |
| 2 | 0.338983 | 0.270339 | -2.228608 | 0.112142 | Original |
| 3 | 0.254237 | 0.168362 | -2.451613 | 0.246559 | Original |
| 4 | 0.288136 | 0.210876 | -5.467540 | 0.012026 | Original |

**Table 2.** *Statistical Summary*

| | Highest Ratio | Average Others | T-Statistic | P-Value |
|---|---|---|---|---|
| count | 4000.000000 | 4000.000000 | 3999.000000 | 3999.000000 |
| mean | 0.490285 | 0.339968 | -6.076672 | 0.036783 |
| std | 0.128376 | 0.082900 | 5.580957 | 0.043217 |
| min | 0.101695 | 0.066667 | -111.524590 | 0.000002 |
| 25% | 0.416667 | 0.296610 | -6.938964 | 0.006418 |
| 50% | 0.491525 | 0.354732 | -4.431515 | 0.021973 |
| 75% | 0.573770 | 0.398224 | -3.176861 | 0.052069 |
| max | 0.868852 | 0.580601 | -1.166065 | 0.451288 |

values for other ratios are higher in the watermarked data compared to the original data. Distribution of T-Statistic:

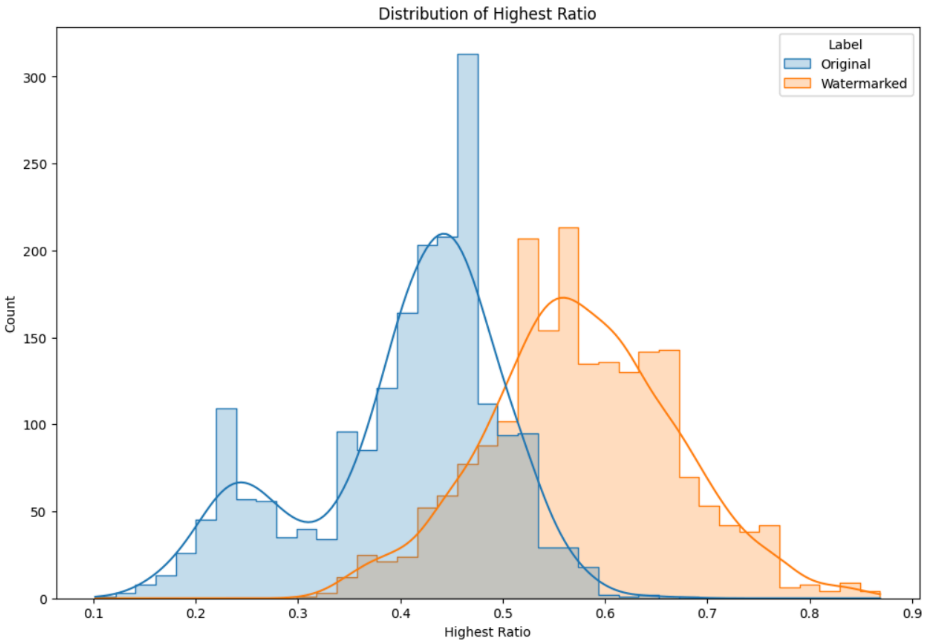


**Figure 2.** *Distribution of highest ratio*

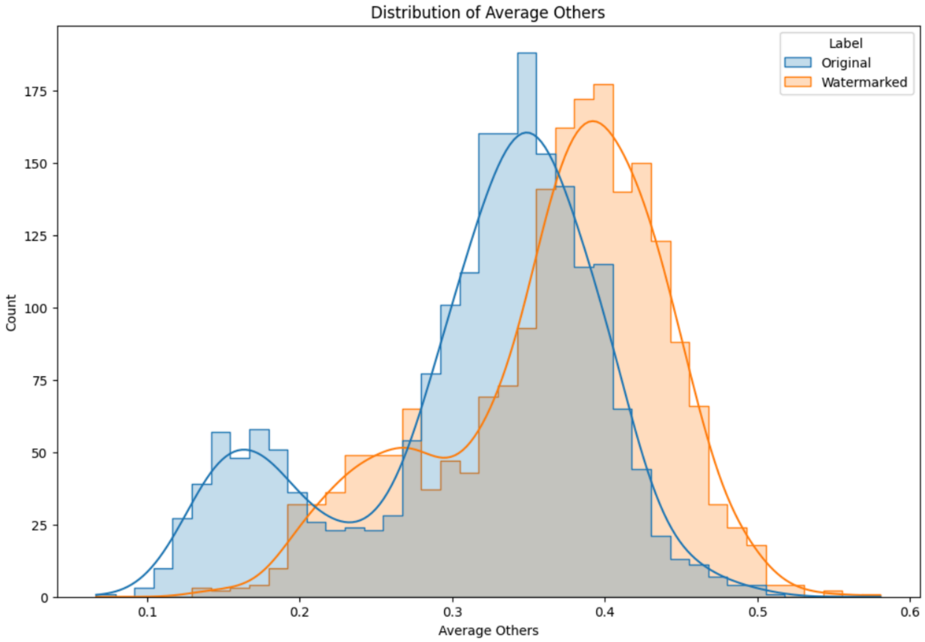


**Figure 3**. *Distribution of average others*

**Distribution of T-statistic:** The distribution of the T-statistic is highly skewed to the left for both categories, with a long tail extending to very negative values. The “Original” category appears to have a more pronounced peak near 0, while the “Watermarked” category has a lower count at the peak and a wider spread.

**Distribution of P-Value:** The P-value distribution is heavily skewed towards 0 for both categories, with the “Watermarked” category showing a sharper peak at 0. This suggests that most of the tests result in very low p-values, indicating strong statistical significance in the differences observed.

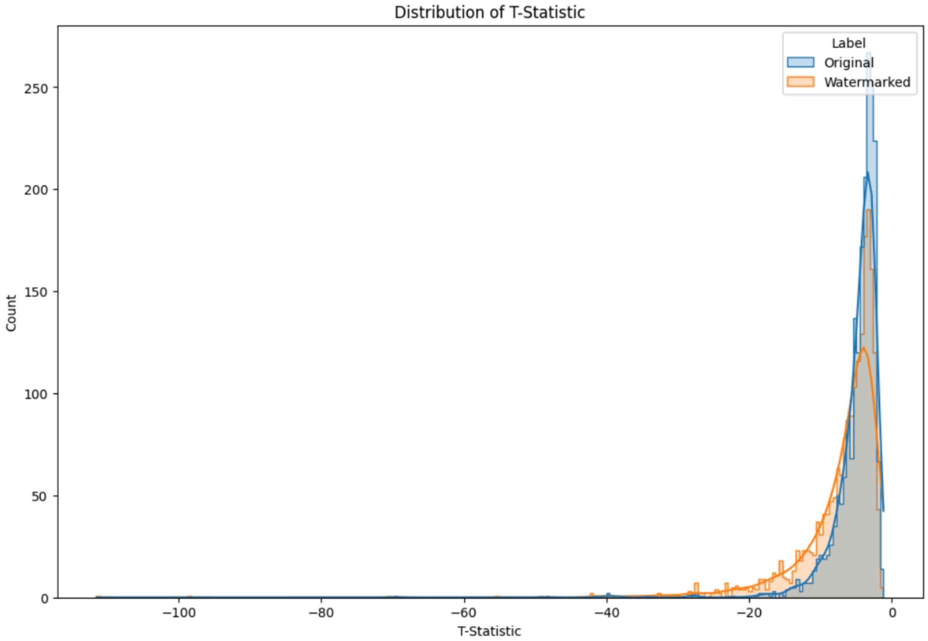


**Figure 4**. *Distribution of t-statistics*

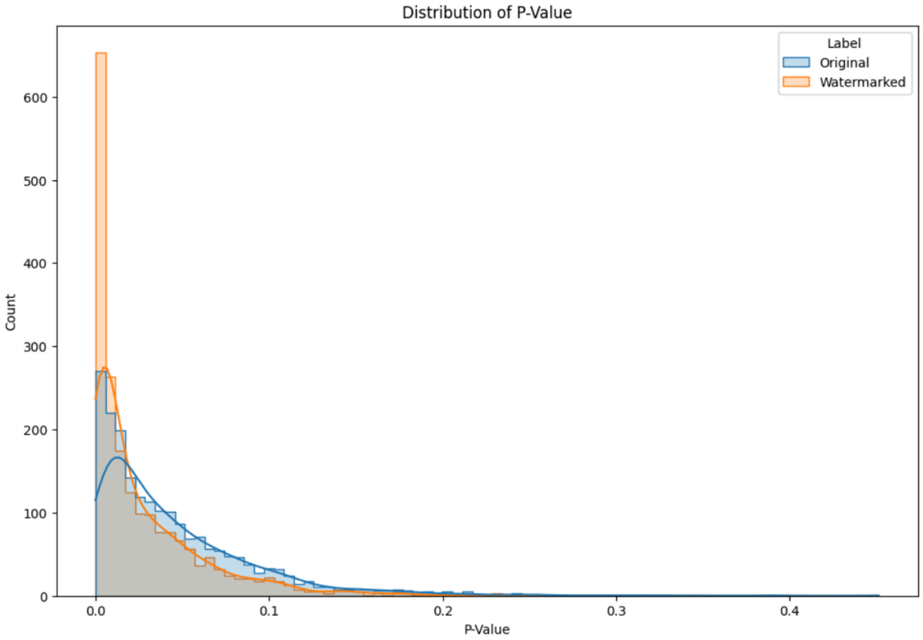


**Figure 5**. *Distribution of P-value*

**Pair Plot:** The pair plot provides a visual comparison of the relationships between the variables for the two categories. There are clear clusters and separations between the "Original" and "Watermarked" categories in the scatter plots, particularly for "Highest Ratio" vs. "Average Others" and "Highest Ratio" vs. "P-Value". This reinforces the idea that the watermarked data exhibits different characteristics compared to the original data.

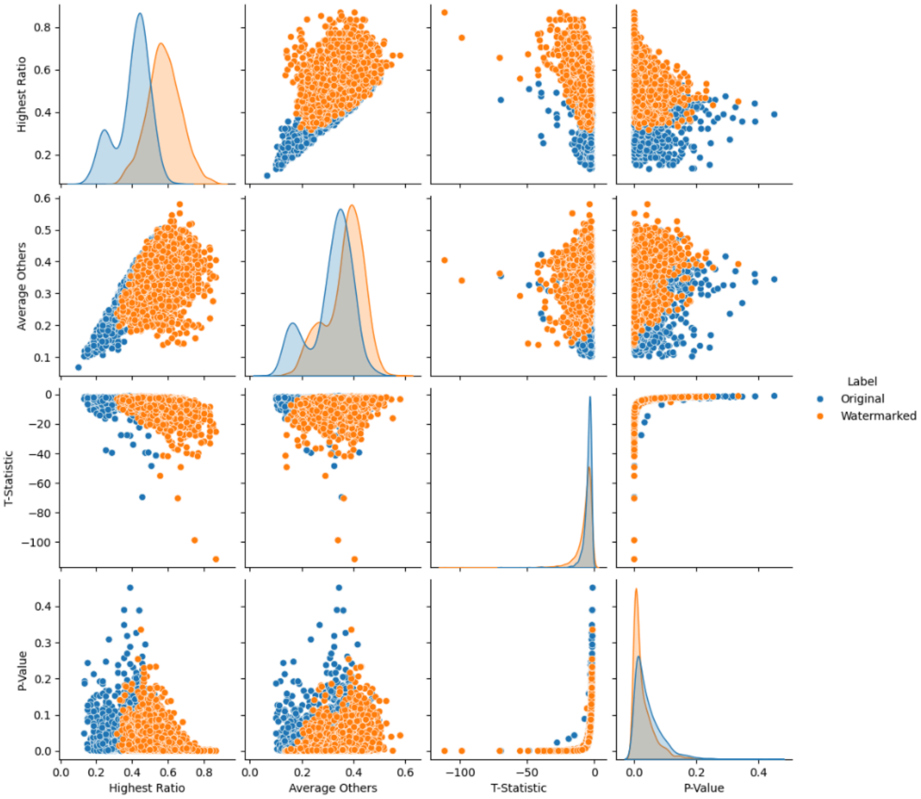


**Figure 6**. *Dataset*

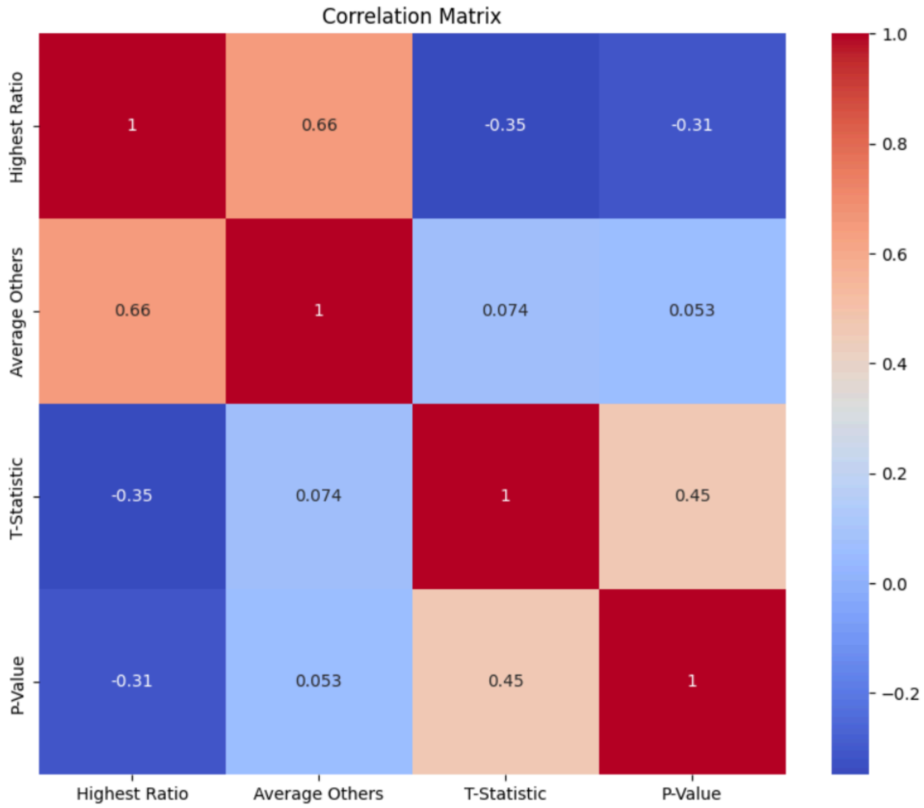


**Figure 7**. *Correlation_Matrix*

**Correlation Matrix:** The correlation matrix shows the pairwise correlation coefficients between the variables. “Highest Ratio” and “Average Others” are positively correlated (0.66), indicating that higher values of the highest ratio tend to be associated with higher average values of other ratios. “T-Statistic” has a negative correlation with “Highest Ratio” (-0.35) and “P-Value” (-0.31), suggesting that higher ratios tend to result in more negative T-statistics and lower p-values.

**Overall Observations:** The “Watermarked” data tends to have higher ratios and averages compared to the “Original” data. The T-statistics and p-values indicate strong statistical differences between the original and watermarked categories. The pair plot and correlation matrix provide further evidence of distinct patterns and relationships in the watermarked data compared to the original data.

While these plots do show a difference between the watermarked and non-watermarked text, using a pre-trained model help us achieve higher efficiency and consistency in our comparisons.

## 7. Model Training, Testing and Efficiency

The algorithm in this paper was trained using a dataset generated from Gutenberg’s top 10 books [6], [7], [8], [9], [10], [11], [12], [13], [14], [15]. Specifically, 2000 random 300-word paragraphs were taken from these books, ensuring an equal number of paragraphs from each book. Each paragraph was watermarked, and then statistical analysis was performed. The Highest Match Ratio, Average of Other Ratios, T-Statistic, and P-Value were calculated and stored in Results.csv. The models were trained using an 80/20 split of the dataset, with the following models being trained: Logistic Regression, Decision Tree, Random Forest, Support Vector Machine, Gradient Boosting, AdaBoost, Naive Bayes, and K-Nearest Neighbors. Gradient Boosting gave the highest accuracy, resulting in an overall accuracy of 94% in identifying watermarked text

**Code used for model training**

```python
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier, GradientBoostingClassifier, AdaBoostClassifier
from sklearn.svm import SVC
from sklearn.naive_bayes import GaussianNB
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import classification_report, confusion_matrix

# Assuming list_of_significance and list_of_significance_watermarked are already defined
# Create DataFrames from the lists
df_significance = pd.DataFrame(list_of_significance, columns=['Highest Ratio', 'Average Others', 'T-
Statistic', 'P-Value'])
df_significance_watermarked = pd.DataFrame(list_of_significance_watermarked, columns=['Highest Ratio',
'Average Others', 'T-Statistic', 'P-Value'])

# Add a label column to distinguish between the two sets
df_significance['Label'] = 'Original'
df_significance_watermarked['Label'] = 'Watermarked'

# Combine the DataFrames
combined_df = pd.concat([df_significance, df_significance_watermarked], ignore_index=True)
combined_df = combined_df.dropna()

# Prepare the data
X = combined_df.drop(columns=['Label'])
y = combined_df['Label']

# Convert labels to numerical values for ML model
y = y.map({'Original': 0, 'Watermarked': 1})

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Initialize models
models = {
    'Logistic Regression': LogisticRegression(random_state=42, max_iter=1000),
    'Decision Tree': DecisionTreeClassifier(random_state=42),
    'Random Forest': RandomForestClassifier(random_state=42),
    'Support Vector Machine': SVC(random_state=42),
    'Gradient Boosting': GradientBoostingClassifier(random_state=42),
    'AdaBoost': AdaBoostClassifier(random_state=42),
    'Naive Bayes': GaussianNB(),
    'K-Nearest Neighbors': KNeighborsClassifier()
}

# Train and evaluate models
for model_name, model in models.items():
    model.fit(X_train, y_train)
    y_pred = model.predict(X_test)
    print(f"\n{model_name} Classification Report:")
    print(classification_report(y_test, y_pred))
    print(f"\n{model_name} Confusion Matrix:")
    print(confusion_matrix(y_test, y_pred))

    # Feature importances (only for models that provide it)
    if hasattr(model, 'feature_importances_'):
        feature_importances = model.feature_importances_
        feature_importances_df = pd.DataFrame({
            'Feature': X.columns,
            'Importance': feature_importances
        }).sort_values(by='Importance', ascending=False)

        # Plot feature importances
        plt.figure(figsize=(12, 8))
        sns.barplot(x='Importance', y='Feature', data=feature_importances_df, palette='viridis')
        plt.title(f'{model_name} Feature Importances')
        plt.show()
```

## Code for Model testing

```python
import os
import random

def extract_test_cases(folder_path, num_cases=2000, words_per_case=300):
    test_cases = []
    book_files = [f for f in os.listdir(folder_path) if os.path.isfile(os.path.join(folder_path, f))]

    # Calculate the number of test cases to extract from each book
    cases_per_book = num_cases // len(book_files)
    extra_cases = num_cases % len(book_files)

    for book_file in book_files:
        with open(os.path.join(folder_path, book_file), 'r', encoding='utf-8') as file:
            text = file.read()
            words = text.split()
            num_words = len(words)

            # Ensure enough words are available to extract the cases
            if num_words < words_per_case:
                continue

            # Determine the number of cases to extract from this book
            num_cases_from_book = cases_per_book
            if extra_cases > 0:
                num_cases_from_book += 1
                extra_cases -= 1

            for _ in range(num_cases_from_book):
                start_index = random.randint(0, num_words - words_per_case)
                case = ' '.join(words[start_index:start_index + words_per_case])
                test_cases.append(case)

                if len(test_cases) == num_cases:
                    return test_cases

    return test_cases

# Usage example
folder_path = 'books'
test_cases = extract_test_cases(folder_path)
```

```python
list_of_significance = []
list_of_significance_watermarked = []
count_t = 0
for text in test_cases:
  count_t+=1
  print("______________________________")
  print("Doing", count_t)
  print("______________________________")

  words_to_add = ["example", "test", "random", "insert"]
  num_words_to_add = 5

  modified_text = randomly_add_words(watermark_text(text, offset=0), words_to_add, num_words_to_add)

  match_ratios = watermark_text_and_calculate_matches(modified_text, max_offset=5)
  list_of_significance_watermarked.append(check_significant_difference(match_ratios))

  match_ratios = watermark_text_and_calculate_matches(text, max_offset=5)
  list_of_significance.append(check_significant_difference(match_ratios))

  print("______________________________")
  print("Done", count_t, )
  print("______________________________")
```

## 8. Analysis of the Algorithm

**Strengths:**

1. Robustness against attacks: The BERT-based watermarking algorithm uses the sophisticated context-understanding capability of BERT to embed watermarks. This makes the watermark integration deeply intertwined with the text's semantic structure, which is difficult to detect and remove without altering the underlying meaning, thus providing robustness against simple text manipulation attacks.
2. Comparison with existing methods: Compared to traditional watermarking methods like word context and UniSpaCh, the BERT-based approach offers a more adaptable and less detectable method. It does not rely on altering visible text elements or patterns easily erased, like white spaces or specific word sequences. Instead, it uses semantic embedding, making it superior in maintaining the natural flow and readability of the text.
3. Scalability and adaptability: The method is scalable to different languages and text forms by adjusting the BERT model used. It can be adapted to work with different BERT variants trained on specific datasets, enhancing flexibility in deployment.

**Challenges:**

1. Dependency on model consistency: The watermark detection relies heavily on the consistency of the BERT model's output. Any updates or changes in the model could potentially alter the watermark, making it undetectable. If the watermark can embbed some sort of version history and control, this could be managed.
2. Data Integrity is highly dependent on the Model: the integrity of the watermarked text depends on how good the model is at replacing the given word, due to the nature of AI-generated text where all the previous tokens are used to generate new ones BERT watermarking can preserve integrity much more effectively. However if it were to watermark text which is completely different from its training dataset it might return an incoherent output, for example if the dataset of BERT consists of scientific papers it will struggle immensely when trying to watermark fairy tails.
3. Potential for false positives/negatives: Given the probabilistic nature of BERT's predictions, there is a risk of incorrect watermark detection, especially in texts with complex semantics or those that closely mimic the watermark patterns without actually being watermarked.
4. Potential loss of context: When words are replaced, the intended context of delivery could be altered. However, AI models are continually improving, and we hope that a well-trained model can significantly mitigate this risk.

**Real-world applicability:**

1. Versatility in applications: This method can be applied across various fields such as copyright protection, and content authentication, and in legal and academic settings where proof of authorship is crucial. It is particularly beneficial for managing copyrights in digital media, academic papers, and any online content where text is dynamically generated or reused.
2. Integration with existing systems: The algorithm can be seamlessly integrated with current content management systems (CMS) and digital rights management (DRM) systems, enhancing their capabilities to include advanced text watermarking features. This integration helps organizations maintain control over their content distribution and monitor usage without invasive methods.
3. Application in AI-generated text: With the proliferation of AI-generated content from models like ChatGPT, GPT-4, and other AI writing assistants, distinguishing between human-generated and AI-generated text becomes crucial. The BERT-based watermarking can be used to embed unique, non-intrusive identifiers into AI-generated texts, ensuring that each piece of content can be traced back to its source. This is particularly valuable in preventing the spread of misinformation, verifying the

authenticity of content, and in applications where copyright claims on AI-generated content might be disputed.

4. Forensic Linguistics in Cybersecurity: In cybersecurity, determining the origin of phishing emails or malicious texts can be crucial. BERT-based watermarking can assist forensic linguists and security professionals by providing a means to trace the origins of specific texts back to their creators, helping to identify patterns or sources of cyber threats.
5. Enhanced Licensing Control for Digital Text: As digital content licensing becomes more complex with different rights for different geographies and platforms, watermarking can help content owners and licensing agencies enforce these rights more effectively. The watermark makes it easier to enforce and monitor compliance automatically.

## 9. Conclusion

By leveraging the BERT model and the proposed algorithm, we have achieved a 94% accuracy rate in detecting watermarked text. With an appropriate training dataset and ongoing advancements in AI technology, this approach promises even more robust watermarking techniques. This progress will enhance our ability to identify AI-generated content and provide an effective means for detecting plagiarism.

## References

[1] N. S. Kamaruddin, A. Kamsin, L. Y. Por, and H. Rahman, “A Review of Text Watermarking: Theory, Methods, and Applications,” *IEEE Access*, vol. 6, no. 3, 2018, doi: 10.1109/ACCESS.2018.2796585.

[2] Y. Wu, Z. Jin, C. Shi, P. Liang, and T. Zhan, “Research on the Application of Deep Learning-based BERT Model in Sentiment Analysis,” *ArXiv*, 2024, [Online]. Available: https://api.semanticscholar.org/CorpusID:268379403

[3] Z. Jalil and A. M. Mirza, “A Review of Digital Watermarking Techniques for Text Documents,” in *2009 International Conference on Information and Multimedia Technology*, 2009, pp. 230–234. doi: 10.1109/ICIMT.2009.11.

[4] L. Y. Por, K. Wong, and K. O. Chee, “UniSpaCh: A text-based data hiding method using Unicode space characters,” *Journal of Systems and Software*, vol. 85, no. 5, pp. 1075–1082, 2012, doi: https://doi.org/10.1016/j.jss.2011.12.023.

[5] T. Lancaster, “Artificial intelligence, text generation tools and ChatGPT - does digital watermarking offer a solution?,” *Int J Educ Integr*, vol. 19, no. 10, pp. 8011–8028, 2023, doi: https://doi.org/10.1007/s40979-023-00131-6.

[6] W. Shakespeare, *Romeo and Juliet*. USA, 1998. [Online]. Available: https://www.gutenberg.org/ebooks/1513

[7] H. Melville, *Moby Dick; Or, The Whale*. USA, 2001. [Online]. Available: https://www.gutenberg.org/ebooks/2701

[8] J. Austen, *Pride and Prejudice*. USA, 1998. [Online]. Available: https://www.gutenberg.org/ebooks/1342

[9] M. W. Shelley, *Frankenstein; Or, The Modern Prometheus*. USA, 1993. [Online]. Available: https://www.gutenberg.org/ebooks/84

[10] G. Eliot, *Middlemarch*. USA, 1994. [Online]. Available: https://www.gutenberg.org/ebooks/145

[11] W. Shakespeare, *The Complete Works of William Shakespeare*. USA, 1994. [Online]. Available: https://www.gutenberg.org/ebooks/100

[12] E. M. Forster, *A Room with a View*. USA, 2001. [Online]. Available: https://www.gutenberg.org/ebooks/2641

[13] L. M. Alcott, *Little Women; Or, Meg, Jo, Beth, and Amy*. USA, 2011. [Online]. Available: https://www.gutenberg.org/ebooks/37106

[14] L. M. Montgomery, *The Blue Castle*. USA, 2022. [Online]. Available: https://www.gutenberg.org/ebooks/67979

[15] E. V. Arnim, *The Enchanted April*. USA, 2005. [Online]. Available: https://www.gutenberg.org/ebooks/16389

Open Access | https://doi.org/10.25080/VPNX1595

# Evaluating Probabilistic Forecasters with sktime and tsbootstrap — Easy-to-Use, Configurable Frameworks for Reproducible Science

**Benedikt Heidrich**[1], **Sankalp Gilda**[2], and **Franz Kiraly**[1]

[1]sktime, [2]DevelopYours, LLC

*SciPy 2024*

*July 8 - July 14, 2024*

Proceedings of the 23rd Python in Science Conference

ISSN: 2575-9752

**Published** Jul 10, 2024

**Correspondence to**
Benedikt Heidrich
benedikt.heidrich@sktime.net

## Abstract

Evaluating probabilistic forecasts is complex and essential across various domains, yet no comprehensive software framework exists to simplify this task. Despite extensive literature on evaluation methodologies, current practices are fragmented and often lack reproducibility. To address this gap, we introduce a reproducible experimental workflow for evaluating probabilistic forecasting algorithms using the sktime package. Our framework features a unified software API for forecasting algorithms, a simple specification language for complex algorithms, including meta-algorithms like bootstrapping, probabilistic performance metrics, and standardized evaluation workflows. We demonstrate the framework's efficacy through a study evaluating prediction intervals added to point forecasts. Our results highlight the improved prediction accuracy and reliability of combined approaches. We provide reusable code and invite contributions from the research community to extend our experiments and tackle computational challenges for broader studies.

**Keywords** time series, machine learning, benchmarking

## 1. Introduction

Making probabilistic forecasts is challenging, and evaluating probabilistic forecasts, or the algorithms that produce them, is even more difficult.

A significant body of literature focuses on developing robust meta-methodologies for evaluation. This includes evaluation metrics such as the Continuous Ranked Probability Score (CRPS) [1] and their properties, like properness, as well as benchmarking setups and competitions like the Makridakis competitions [2], [3]. This meta-field builds upon a broader primary field that develops methodologies for algorithms producing probabilistic forecasts, encompassing classical methods, uncertainty estimation techniques like bootstrap or conformal intervals, and modern deep learning and foundation models [4], [5], [6], [7].

Despite the critical importance of evaluating probabilistic forecasts in various domains, including finance, energy, healthcare, and climate science, no comprehensive software framework or interface design has emerged to cover all these needs with a simple workflow or specification language. For instance, the reproducing code for the Makridakis competitions—while extensive in scope—relies on forecasts generated from disparate software interfaces. Similar issues are found in other benchmarking studies, where code availability is often limited or nonexistent [8], [9]. This lack of unified interfaces makes it challenging for practitioners in both industry and academia to contribute to or verify the growing body of evidence.

To address these limitations, we present a simple, reproducible experimental workflow for evaluating probabilistic forecasting algorithms using `sktime` [10]. As of 2024, the `sktime` package provides the most comprehensive collection of time series-related algorithms in unified interfaces and stands out as the only major, non-commercially governed framework for time series forecasting.

The key components of this reproducible benchmarking framework are:

- A unified software API for forecasting algorithms, mirroring a unified mathematical interface.
- Composite forecasters (meta-algorithms), such as adding prediction intervals via time series bootstrapping, which themselves follow the same forecasting interface from both software and mathematical perspectives.
- A first-order language that allows for the unambiguous specification of even complex forecasting algorithms.
- A unified software API for probabilistic performance metrics, covering metrics for distribution as well as interval or quantile forecasts.
- A standardized workflow for obtaining benchmark result tables for combinations of algorithms, metrics, and experimental setups.

To demonstrate the efficacy and ease of use of `sktime` in benchmarking probabilistic forecasters, we conducted a small study exploring the performance of various meta-algorithms (wrappers) that add prediction intervals to point forecasters. We investigated a range of forecasters, including Naive Forecasting and AutoTheta models [11], along with probabilistic wrappers such as Conformal Intervals and BaggingForecaster with different bootstrapping methods. For the bootstrapping methods, we use the `tsbootstrap` library [12], [13].

The study's goal was to evaluate the effectiveness of these combined approaches in improving prediction accuracy and reliability.

We conducted experiments on several common datasets, including Australian electricity demand [14], sunspot activity [15], and US births [16]. These datasets represent different time frequencies and characteristics.

Our paper is accompanied by easily reusable code, and we invite the open research and open-source communities to contribute to extending our experiments or using our code to set up their own. As is often the case in modern data science, computational power is a limiting factor, so we hope to leverage the SciPy conference to plan a more comprehensive set of studies.

The remainder of this paper is organized as follows: In Section 3, we describe the forecasting methods and probabilistic wrappers used in our experiments. Section 4.2 provides an overview of the datasets used for evaluation. In Section 4.3, we present the experimental results and discuss the performance of the combined approaches. Finally, in Section 5, we conclude the paper and outline directions for future research.

## 2. sktime and tsbootstrap for Reproducible Experiments

In this section, we summarize the key design principles used for reproducible benchmarking in `sktime`. Thereby, from a software perspective, it is worth noting that **sktime** [10], [17] contains multiple native implementations, including naive methods, all probability wrappers, and pipeline composition, but also provides a unified interface across multiple packages in the time series ecosystem, e.g.:

- **tsbootstrap** [12], [13]: A library for time series bootstrapping methods, integrated with `sktime`.
- **statsforecast** [18]: A library for statistical and econometric forecasting methods, featuring the Auto-Theta algorithm.
- **statsmodels** [19]: A foundational library for statistical methods, used for the deseasonalizer and various statistical primitives.

This hybrid use of `sktime` as a framework covering first-party (itself), second-party (`tsbootstrap`), and third-party (`statsmodels`, `statsforecast`) packages is significant. Credit goes to the maintainers and implementers of these packages for implementing the contained algorithms that we can interface.

### *2.1. Unified Interface*

In `sktime` and `tsbootstrap`, algorithms and mathematical objects are treated as first-class citizens. All objects of the same type, such as forecasters, follow the same interface. This consistency ensures that every forecaster is exchangeable with any other. The same holds for all bootstrapping methods in `tsbootstrap`. Thus, they are easily exchangeable enabling simple and fast experimentation.

Forecasters are objects with `fit` and `predict` methods, allowing both the target time series (endogenous data) and the time series that influence the target time series (exogenous data) to be passed. For example, the following code (Program 1) specifies a forecaster, fits it, and makes predictions.

A crucial design element in the above is that line 9 can specify any forecaster - for instance, `ARIMA` or `ExponentialSmoothing`, and the rest of the code will work without modification.

Currently, `sktime` supports the construction of 82 individual forecasters. Some of these are implemented directly in sktime, but sktime also provides adapters providing a unified API to forecasting routines from other libraries, such as gluonts and prophet. Each forecaster is parametric, with configurations ranging from 1 to 47 parameters across different forecasters. A list of forecasters can be obtained or filtered via the `all_estimators` utility.

```
# importing the necessary modules
from sktime.datasets import load_airline
from sktime.forecasting.naive import NaiveForecaster

# load exemplary data
y = load_airline()

# specifying the forecasting algorithm
forecaster = NaiveForecaster(strategy="last", sp=12)

# fitting the forecaster -- forecast y into the future, 3 steps ahead
forecaster.fit(y, fh=[1, 2, 3])

# querying predictions
y_pred = forecaster.predict()
```

**Program 1**. *Exemplary code for fitting and predicting with a forecaster*

```
from sktime.registry import all_estimators

# get all forecasters
all_forecasters = all_estimators("forecaster")

# get all forecasters that support prediction intervals
all_estimators("forecaster", filter_tags={"capability:pred_int": True})
```

**Program 2**. *Code to list all forecaster and all probabilistic forecasters (tag=capability:pred_int)*

```
# importing the necessary modules
from sktime.datasets import load_airline
from sktime.forecasting.naive import NaiveForecaster

# load exemplary data
y = load_airline()

# specifying the forecasting algorithm
forecaster = NaiveForecaster(strategy="last", sp=12)

# fitting the forecaster -- forecast y into the future, 3 steps ahead
forecaster.fit(y, fh=[1, 2, 3])

# making interval predictions
y_pred_int = forecaster.predict_interval()
# making distribution predictions
y_pred_proba = forecaster.predict_proba()
```

**Program 3**. *Exemplary probabilistic forecast with NaiveForecaster*

To better filter the forecasters (as well as the other estimators such as classifiers and regressors) based on their properties and capabilities and to control their behaviour, `sktime` implements a tag system. Each estimator, such as a forecaster, has a dict `_tags`, with a string as a key describing the name and a value of an arbitrary type describing the property. This type system enables an easy identification of all probabilistic forecasters, since they are tagged with the `"capability:pred_int"` tag. Currently, 45 of the 82 forecasters support probabilistic prediction modes (Program 2), such as prediction intervals, quantile predictions, or full distributional predictions:

More details can be found in the official tutorials (an overview of notebooks and tutorials is provided on our homepage). Creating algorithms with compatible APIs is straightforward.

`sktime` and `tsbootstrap` provide fill-in extension templates for creating algorithms in a compatible interface, which can be shared in a third-party code base and indexed by `sktime`, or contributed directly to `sktime`.

## *2.2. Reproducible Specification Language*

All objects in `sktime` are uniquely specified by their construction string, which serves as a reproducible blueprint. Algorithms are intended to be stable and uniquely referenced across versions; full replicability can be achieved by freezing Python environment versions and setting random seeds.

For example, the specification string `NaiveForecaster(strategy="last", sp=12)` uniquely specifies a native implementation of the seasonal last-value-carried-forecaster, with seasonality parameter 12.

Typical `sktime` code specifies the forecaster as Python code, but it can also be used as a string to store or share specifications. The `registry.craft` utility converts a Python string into an `sktime` object, ensuring easy reproducibility:

which can be used to copy a full specification from a research paper and immediately construct the respective algorithm in a Python environment, even if full code has not been shared by the author.

```
from sktime.registry import craft
forecaster = craft('NaiveForecaster(strategy="last", sp=12)')
```

**Program 4**. *Code to craft a forecaster from a string*

```python
# importing the necessary modules
from sktime.datasets import load_airline
from sktime.forecasting.naive import NaiveForecaster
from sktime.forecasting.conformal import ConformalIntervals

# load exemplary data
y = load_airline()

# specifying the forecasting algorithm
forecaster = ConformalIntervals(NaiveForecaster(strategy="last", sp=12))

# fitting the forecaster -- forecast y into the future, 3 steps ahead
forecaster.fit(y, fh=[1, 2, 3])

# making interval predictions
y_pred_int = forecaster.predict_interval()
```

**Program 5**. *Code to add conformal prediction intervals to a forecaster*

This approach makes it extremely easy to share specifications reproducibly, simplifying the often convoluted process of describing algorithms in research papers.

## *2.3. Meta-Algorithms and Composability*

`sktime` provides not only simple algorithms as objects in unified interfaces but also meta-algorithms, such as data processing pipelines or algorithms that add interval forecast capability to existing forecasting algorithms. Importantly, these composites also follow unified interfaces.

This leads to a compositional specification language with rich combinatorial flexibility. For example, the following code adds conformal prediction interval estimates to the `NaiveForecaster` (Program 5):

The conformal prediction interval fits multiple instances of the wrapped forecaster on parts of the time series using window sliding. In particular, each forecaster's instance $i$ is fit on the time series $y_{1:t_i}$, where $t_i \neq t_j$. Afterwards, the residuals are computed on the remaining time series $y_{t_{i+1}:n}$, where $n$ is the length of the time series. Out of these residuals, the prediction intervals are computed. The resulting algorithm possesses `fit`, `predict`, and - added by the wrapper - `predict_interval`, as well as the `capability:pred_int` tag.

Data transformation pipelines can be constructed similarly, or with an operator-based specification syntax (Program 6):

This creates a forecasting algorithm that first computes differences, then remove the seasonality (deseasonalize) by assuming a periodicity of 12. Then the

```python
# importing the necessary modules
from sktime.datasets import load_airline
from sktime.forecasting.naive import NaiveForecaster
from sktime.transformations.series.detrend import Deseasonalizer
from sktime.transformations.series.difference import Differencer

# load exemplary data
y = load_airline()

pipeline = Differencer() * Deseasonalizer(sp=12) * NaiveForecaster(strategy="last")

# fitting the forecaster -- forecast y into the future, 3 steps ahead
pipeline.fit(y, fh=[1, 2, 3])

# making interval predictions
y_pred_int = pipeline.predict()
```

**Program 6**. *Code to construct a pipeline with a differencer and a seasonalizer*

```python
# importing the necessary modules
from sktime.datasets import load_airline
from sktime.forecasting.naive import NaiveForecaster
from sktime.performance_metrics.forecasting.probabilistic import CRPS
from sktime.split import temporal_train_test_split

# load exemplary data
y = load_airline()
y_train, y_test = temporal_train_test_split(y, test_size=36)

# specifying the forecasting algorithm
forecaster = NaiveForecaster(strategy="last", sp=12)

# fitting the forecaster -- forecast y into the future, 3 steps ahead
forecaster.fit(y_train, fh=y_test.index)

# making interval predictions
y_pred_int = forecaster.predict_interval()
# making distribution predictions
y_pred_proba = forecaster.predict_proba()

# Initialise the CRPS metric
crps = CRPS()

# Calculate the CRPS
crps(y_test, y_pred_proba)
```

**Program 7**. *Code to make probabilistic forecasts and calculate the CRPS*

`NaiveForecaster(strategy="last")` uses the last value as prediction for the next value (last-value-carry-forward method). Finally it adds the seasonality back and inverts the first differences. As before, the resulting forecaster provides unified interface points and is interchangeable with other forecasters in `sktime`.

### *2.4. Probabilistic Metrics*

In `sktime`, evaluation metrics are first-class citizens. Probabilistic metrics compare the ground truth time series with predictions, representing probabilistic objects such as predictive intervals or distributions. For example:

Tags control the type of probabilistic prediction expected, such as `"scitype:y_pred"` with the value `"pred_proba"` for CRPS.

### *2.5. Benchmarking and Evaluation*

Our benchmarking framework involves a standardized workflow for obtaining benchmark result tables for combinations of algorithms, metrics, and experimental setups. Here is an example of how to add forecasters and tasks:

This approach ensures that our benchmarking process is both comprehensive and reproducible.

### *2.6. Prior Work*

The design principles of `sktime` draw inspiration from several established machine learning frameworks. Notably, `scikit-learn` in Python [20], `mlr` in R [21], and Weka [22] have pioneered the use of first-order specification languages and object-oriented design patterns for machine learning algorithms.

`sktime` extends these foundational ideas by introducing specialized interfaces for time series forecasting, including both point and probabilistic forecasts. It also incorporates unique features such as:

- Probabilistic forecasters and associated metrics interfaces

```python
# Example code to add forecasters and tasks

# Import the necessary modules
from sktime.benchmarking.forecasting import ForecastingBenchmark
from sktime.datasets import load_airline
from sktime.forecasting.naive import NaiveForecaster
from sktime.performance_metrics.forecasting.probabilistic import CRPS
from sktime.split import SlidingWindowSplitter

# Initialise the benchmark object
benchmark = ForecastingBenchmark()

# Add a forecaster to the benchmark
benchmark.add_estimator(
    estimator=NaiveForecaster(strategy="last", sp=12),
    estimator_id="naive_forecaster"
)

# Initialise the Splitter
cv_splitter = SlidingWindowSplitter(
    step_length=12,
    window_length=24,
    fh=range(12)
)

# Add a task to the benchmark
# A task is a combination of a dataset loader, a splitter, and a list of metrics
benchmark.add_task(
    load_airline,
    cv_splitter,
    [CRPS()]
)

# Run the benchmark
benchmark.run("result.csv")
```

**Program 8.** *Example usage of the ForecastingBenchmark in sktime.*

- Reproducibility patterns and APIs
- Meta-algorithms for enhanced composability
- Inhomogeneous API composition for flexible algorithmic design

Additionally, the `R/fable` package [23] has contributed similar concepts specifically tailored to forecasting, which `sktime` has expanded upon or adapted to fit within its framework.

By leveraging and building upon these prior works, `sktime` offers a comprehensive and adaptable toolkit for time series forecasting, fostering reproducible research and facilitating extensive benchmarking and evaluation. For a detailed analysis of design patterns in AI framework packages and innovations in `sktime`, see [24].

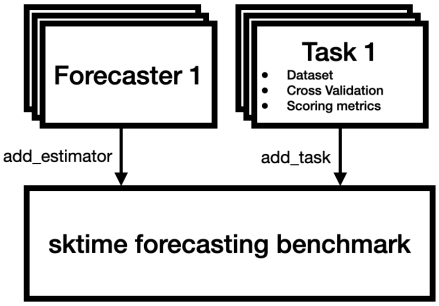


**Figure 1.** *Benchmarking and evaluation framework*

## 3. Algorithmic Setup

In this section, we describe the forecasting algorithms used in our experiments. Our methods combine traditional forecasting models with uncertainty estimation wrappers, showcasing the benchmarking and model specification capabilities of `sktime`. This study serves as an invitation to the scientific Python community to engage and contribute to a more systematic study with reproducible specifications.

### 3.1. *Forecasting Pipeline*

Each forecaster is wrapped in a `Differencer` and a `Deseasonalizer` as preprocessing steps to improve stationarity. These preprocessors are necessary because some forecasters require the time series to be stationary (i.e., the properties of the time series at time $t + 1$, $t + 2$, …, $t + n$ do not depend on the observation at time $t$ [25]) and non-seasonal.

- `Differencer`: Computes first differences, which are inverted after forecasting by cumulative summing.
- `Deseasonalizer(sp=data_sp)`: Removes the seasonal component, which is added back after forecasting. It estimates the trend by applying a convolution filter to the data, removing the trend, and then averaging the de-trended series for each period to return the seasonal component.

All used forecasters are point forecasters, i.e., for each time step they provide one value (point), and no information about the uncertainty of the forecast. Thus, they are combined with one of the probabilistic wrappers to generate prediction intervals or quantile forecasts.

The partial pipeline specification, illustrated in Figure 2, is:

```
Differencer() * Deseasonalizer(sp=data_sp) * wrapper(forecaster)
```

where variable parts are `wrapper`, `forecaster`, and `data_sp`. These components are varied as described below. Note the code for running the whole benchmark is provided in the Section 4.

### 3.2. *Component Forecasting Models*

We use several component forecasting models in this study, each with unique characteristics:

- `NaiveForecaster(strategy, sp)`: Uses simple heuristics for forecasting. The `strategy` parameter allows selecting the type of algorithm:
    - `mean`: Uses the mean of the last $N$ values, with seasonal periodicity `sp` if passed.

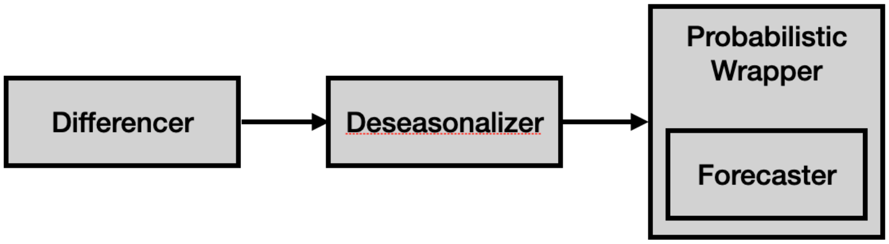


**Figure 2.** *The forecasting pipeline used for all the forecasters*

  - `last`: Uses the last observed, with seasonal periodicity `sp` if passed.
  - `drift`: Fits a line between the first and last values of the considered window and extrapolates forward.
- `StatsForecastAutoTheta(sp)`: A variant of the Theta model of V. Assimakopoulos and K. Nikolopoulos [26] with automated parameter tuning, from the `statsforecast` library.

### 3.3. *Probabilistic Wrappers*

We use the following probabilistic wrappers to enhance the forecasting models:

- `ConformalIntervals(forecaster, strategy)`: Uses conformal prediction methods G. Shafer and V. Vovk [27] to produce non-parametric prediction intervals. Variants of the method are selected by the `strategy` parameter: **Empirical** and **Empirical Residual** use training quantiles, with the latter using symmetrized residuals. **Conformal** implements the method of G. Shafer and V. Vovk [27], and **Conformal Bonferroni** applies the Bonferroni correction [28] .
- `BaggingForecaster(bootstrap_transformer, forecaster)`: Provides probabilistic forecasts by bootstrapping time series and aggregating the bootstrap forecasts [25], [29]. The `BaggingForecaster` takes a bootstrap algorithm `bootstrap_transformer`, a first-class object in `sktime`. Various bootstrap algorithms with their parameters are applied in the study.
- `NaiveVariance(forecaster)`: Uses a sliding window to compute backtesting residuals, aggregated by forecasting horizon to a variance estimate. The mean is obtained from the wrapped forecaster, and variance from the pooled backtesting estimate.
- `SquaringResiduals(forecaster, residual_forecaster)`: Uses backtesting residuals on the training set, squares them, and fits the `residual_forecaster` to the squared residuals. Forecasts of `residual_forecaster` are used as variance predictions, with mean predictions from `forecaster`, to obtain a normal distributed forecast. In this study, `residual_forecaster` is always `NaiveForecaster(strategy="last")`.

### 3.4. *Bootstrapping Techniques*

Bootstrapping methods generate multiple resampled datasets from the original time series data, which can be used as part of wrappers to estimate prediction intervals or predictive distributions. In this study, we use bootstrap algorithms from `tsbootstrap` [12], [13], `sktime` [10], and `scikit-learn` [20] compatible framework library dedicated to time series bootstrap algorithms. `sktime` adapts these algorithms via the `TSBootstrapAdapter`, used as `bootstrap_transformer` in `BaggingForecaster`.

- `MovingBlockBootstrap`: Divides the time series data into overlapping blocks of a fixed size and resamples these blocks to create new datasets. The block size is chosen to capture the dependence structure in the data.
- `BlockDistributionBootstrap`: Generates bootstrapped samples by fitting a distribution to the residuals of a model and then generating new residuals from the fitted distribution. This method assumes that the residuals follow a specific distribution, such as Gaussian or Poisson, and handles dependencies by resampling blocks of residuals. To create a new time series, the bootstrapped residuals are added to the model's fitted values.
- `BlockResidualBootstrap`: Designed for time series data where a model is fit to the data, and the residuals (the difference between the observed and predicted data) are bootstrapped. This method is particularly useful when a good model fit is available for the data. The bootstrapped samples are created by adding the bootstrapped residuals to the model's fitted values.
- `BlockStatisticPreservingBootstrap`: Generates bootstrapped time series data while preserving a specific statistic of the original data. This method handles dependencies by

resampling blocks of data while ensuring that the preserved statistic remains consistent.

In this study, these bootstrapping techniques are used to estimate the distribution of forecasts and generate robust prediction intervals and predictive distributions as part of the `BaggingForecaster`.

### *3.5. Evaluation Metrics*

We evaluate the performance of our forecasting models using the following metrics:

- `CRPS` - Continuous Ranked Probability Score [30] measures the accuracy of probabilistic forecasts by comparing the predicted distribution to the observed values. The CRPS for a real-valued forecast distribution $d$ and an observation $y$ can be defined as:

$$\mathrm{CRPS}(d, y) = \boldsymbol{E}[|\ X - y\ |] - \frac{1}{2}\boldsymbol{E}[|\ X - X'\ |], \tag{1}$$

where $X$ and $X'$ are independent random variables with distribution $d$.

- `PinballLoss` - the pinball loss, also known as quantile loss [31], evaluates the accuracy of quantile forecasts by penalizing deviations from the true values based on specified quantiles. For quantile forecasts $\hat{q}_1, ..., \hat{q}_k$ at levels $\tau_1, ..., \tau_k$ and an observation $y$, the Pinball Loss is defined as:

$$L(\hat{q}, y) = \frac{1}{k}\sum_{i=1}^{k} \max(\tau_i(y - \hat{q}_i), (1 - \tau_i)(\hat{q}_i - y)) \tag{2}$$

The experiment uses the Pinball Loss at quantiles 0.05, and 0.95.

- `AUCalibration` - The Area between the calibration curve and the diagonal assesses how well the prediction intervals capture the true values. For observations $y_1, ..., y_n$ and corresponding distributional predictions with quantile functions $Q_1, ..., Q_n$ (where $Q_i = F_i^{-1}$ for the cdf $F_i$), the AUCalibration is defined as:

$$\mathrm{AUC}(Q_:, y_:) = \frac{1}{N}\sum_{i=1}^{N} |\ c_{(i)} - \frac{i}{N}\ |, \tag{3}$$

where $c_i := Q_i(y_i)$, and $c_{(i)}$ is the $i$-th order statistic of $c_1, ..., c_N$.

- `IntervalWidth` - width of prediction intervals, or sharpness measures the concentration of the prediction intervals. More concentrated intervals indicate higher confidence in the forecasts. Sharpness is desirable because it indicates precise predictions. Sharpness is calculated as the average width of the prediction intervals.
- `EmpiricalCoverage` - Empirical coverage measures how much of the observations are within the predicted interval. It is computed as the proportion of observations that fall within the prediction, providing a direct measure of the reliability of the intervals. A prediction interval ranging from the 5th to the 95th quantile should cover 90% of the observations. I.e., the empirical coverage should be close to 0.9.
- `runtime` - Besides metrics that assess the quality of the forecast, average runtime for an individual fit/interence run is also reported. Runtime measures the computational efficiency of the forecasting methods, which is crucial for practical applications.

## 4. Experiments

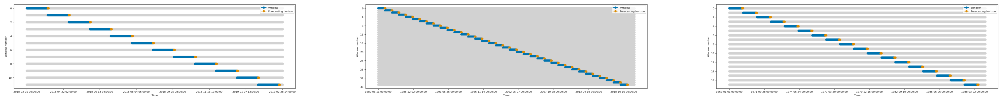
In this section, we describe the experimental setup, the datasets, the evaluation metrics, and the experimental procedures. We explain how the experiments are designed to compare the performance of different forecasting methods.

### 4.1. *Experimental Setup*

To perform the benchmarking study, we use the framework described in Section 3. The benchmarking compares different probabilistic wrappers on different datasets and with different forecasters regarding CRPS, Pinball Loss, AUCalibration, and Runtime.

To enable easy replication of the experiments, we provide for each used forecaster, and wrapper the hyperparameters by providing the used Python object instantiation in Table 1. Note, that the parameter seasonal periodicity (sp) is dataset dependent and is set to 48 for the Australian Electricity Dataset and 1 for the other datasets.

To create the cross validation folds, we use the `SlidingWindowSplitter` from `sktime`. The instantiation of the splitter for each dataset is shown in Table 3. Figure 3 is showing the resulting cross validation folds for the tree datasets. The properties of the datasets are summarized in Table 2.



(a) (b) (c)

**Figure 3**. *The splits used for the evaluation on the three datasets. Blue indicates the training data, and orange indicates the test data. The splits are created using the parameters from Table 2 and Table 3*

### 4.2. *Datasets*

We evaluate our forecasting methods and probabilistic wrappers on several diverse time series datasets, each offering unique characteristics:

- **Australian Electricity Demand [14]**: Half-hourly electricity demand data for five states of Australia: Victoria, New South Wales, Queensland, Tasmania, and South Australia, useful for high-frequency data evaluation.
- **Sunspot Activity [15]**: Weekly observations of sunspot numbers, ideal for testing forecasting robustness on long-term periodic patterns.
- **US Births [16]**: Daily birth records in the United States, with a clear seasonal pattern, suitable for daily data performance assessment.

### 4.3. *Results*

In this section, we present the results of our experiments. We evaluate the performance of the forecasting methods combined with probabilistic wrappers on the datasets described in Section 4.2.

To increase the conciseness, we calculated the rank of each probabilistic wrapper for each combination of forecaster, metric, and dataset. Afterwards, for each metric, probabilistic wrapper and dataset, we have calculated the average across all forecasters and time series. In the following, we present the results for each dataset separately, except for the runtime, which is the same for all three experiments. Thus, we describe it only for the Australian Electricity Demand dataset.

#### 4.3.1. *Performance on Australian Electricity Demand:*

The results for the Australian electricity demand dataset are summarized in Table 4. We compare the performance of different forecasting models and probabilistic wrappers using the previously described evaluation metrics.

The ranked based evaluation show that diverse results regarding the different metrics. E.g. while CI Empirical Residual performs best on CRPS, it is only mediocre regarding the

**Table 1.** *The table lists the specification strings for the estimators used in the study. Note that a full pipeline consists of pre-processing, wrapper, and base forecaster, as detailed in Section 3.*
*Some of the parameters are determined by the used dataset: sp is 48 for the Autralian Electricity Dataset and 1 for the other. The sample_freq is 0.005 for the Australian Electricity Dataset and 0.1 for the other.*

| Role | Name | Hyperparameters |
|---|---|---|
| Base Forecaster | Naive last | `NaiveForecaster(strategy="last", sp=sp)` |
| Base Forecaster | Naive mean | `NaiveForecaster(strategy="mean", sp=sp)` |
| Base Forecaster | Naive drift | `NaiveForecaster(strategy="drift", sp=sp)` |
| Base Forecaster | Theta | `StatsForecastAutoTheta(season_length=sp)` |
| Wrapper | CI Empirical | `ConformalIntervals(forecaster, sample_frac=sample_frac)` |
| Wrapper | CI Empirical residuals | `ConformalIntervals(forecaster, sample_frac=sample_frac, method="empirical_residual")` |
| Wrapper | CI Conformal | `ConformalIntervals(forecaster, sample_frac=sample_frac, method="conformal")` |
| Wrapper | CI Bonferroni | `ConformalIntervals(forecaster, sample_frac=sample_frac, method="conformal_bonferroni")` |
| Wrapper | BaggingForecaster | `BaggingForecaster(ts_bootstrap_adapter, forecaster))` |
| Wrapper | Naive Variance | `NaiveVariance(forecaster, initial_window=14*sp))` |
| Wrapper | Squaring Residuals | `SquaringResiduals(forecaster, initial_window=14*sp))` |
| Forecasting Pipeline | Pipeline | `Differencer(1) * Deaseasonalizer(sp=sp) * Wrapper` |
| ts_bootstrap_adapter | TSBootstrapAdapter | `TSBootstrapAdapter(tsbootsrap)` |
| tsbootstrap | Moving Block Bootstrap | `MovingBlockBootstrap()` |
| tsbootstrap | Block Residual Bootstrap | `BlockDistributionBootstrap()` |
| tsbootstrap | Block Statistic Preserving Bootstrap | `BlockStatisticPreservingBootstrap()` |
| tsbootstrap | Block Distribution Bootstrap | `BlockDistributionBootstrap()` |

Pinball Loss and the AUCalibration. On PinballLoss, the best method is CI Empirical and on AUCalibration, it is Moving Block Bootstrap. Regarding the runtime, the fastest method

**Table 2.** *To perform the evaluation, we used three datasets. Due to the different frequencies and lengths, we used different parameters for the Sliding Window Splitter to create the cross-validation folds. This table presents the parameters used for each dataset.*

| Dataset | Forecast Horizon | Step Width | Window Size | Cutout Period | Number of Folds | Seasonal Periodicity |
|---|---|---|---|---|---|---|
| Australian Electricity Demand | 48 | 1440 | 1440 | Last Year | 12 | 48 |
| Sunspot Activity | 28 | 395 | 365 | Last 40 Years | 12 | 1 |
| US Births | 28 | 395 | 365 | Whole Time Series | 12 | 1 |

**Table 3.** *The code instantiation of the cross-validation splits used for the evaluation on the three datasets. The parameters are taken from Table 2.*

| Dataset | CV splitter |
|---|---|
| Australian Electricity Demand | `SlidingWindowSplitter(step_length=48*30, window_length=48*30,fh=range(48))` |
| Sunspot Activity | `SlidingWindowSplitter(step_length=395, window_length=365,fh=range(28))` |
| US Births | `SlidingWindowSplitter(step_length=395, window_length=365,fh=range(28))` |

**Table 4.** *Performance of forecasting methods on the Australian electricity demand dataset*

| Wrapper | CRPS | Pinball Loss | AUCalibration | Runtime |
|---|---|---|---|---|
| Fallback | 9.45 | 9.66 | 5.61 | 1.00 |
| Naive Variance | 7.75 | 7.20 | 7.15 | 10.00 |
| Squaring Residuals | 7.45 | 6.20 | 5.05 | 11.00 |
| Block Distribution Bootstrap | 4.08 | 3.15 | 6.82 | 7.83 |
| Block Residual Bootstrap | 4.30 | 5.58 | 5.84 | 8.32 |
| Block Statistic Preserving Bootstrap | 4.25 | 5.87 | 6.15 | 7.29 |
| Moving Block Bootstrap | 6.12 | 6.51 | 4.26 | 6.00 |
| CI Conformal | 5.75 | 5.40 | 5.69 | 3.92 |
| CI Empirical | 3.79 | 2.63 | 6.97 | 3.30 |
| CI Empirical Residual | 2.37 | 5.42 | 5.35 | 3.49 |
| CI Bonferroni | 10.50 | 8.05 | 6.70 | 3.70 |

is the fallback probabilistic prediction of the base forecaster. The slowest methods are NaiveVariance and SquaringResiduals. Furthermore, it seems that the ConformalIntervals are slightly faster than the BaggingForecasters.

### 4.3.2. *Performance on Sunspot Activity:*

Table 5 shows the performance of our methods on the sunspot activity dataset. The long-term periodic patterns in this dataset provide a challenging test for our forecasting models.

The ranked based evaluation show that BaggingForecaster with the Block Distribution Bootstrap scores clearly best regarding the CRPS and Pinball Loss, and AUCalibration.

### 4.3.3. *Performance on US Births:*

The results for the US births dataset are presented in Table 6. This dataset, with its clear seasonal pattern, allows us to assess the models' ability to handle daily data. The ranked based evaluation show that BaggingForecaster with the Block Distribution Bootstrap scores best regarding the CRPS and Pinball Loss. Regarding the AUCalibration, the best score is achieved by CI Conformal.

## 5. Discussion and Conclusion

Our experiments demonstrate that the benchmarking framework in `sktime` provides an easy-to-use solution for reproducible benchmarks. We showed this by conducting simple benchmark studies of probabilistic wrappers for point forecasts on three different systems and make the corresponding code available at: https://github.com/sktime/code_for_paper_scipyconf24/tree/main.

**Table 5**. *Performance of forecasting methods on the sunspot activity dataset*

| Wrapper | CRPS | PinballLoss | AUCalibration | Runtime |
|---|---:|---:|---:|---:|
| Fallback | 9.00 | 7.25 | 9.50 | 1.00 |
| Naive Variance | 6.50 | 6.25 | 7.75 | 10.00 |
| Squaring Residuals | 8.00 | 8.00 | 4.75 | 11.00 |
| Block Distribution Bootstrap | 1.00 | 1.00 | 3.00 | 7.25 |
| Block Residual Bootstrap | 6.00 | 6.25 | 5.00 | 6.75 |
| Block Statistic Preserving Bootstrap | 5.50 | 7.00 | 2.50 | 6.25 |
| Moving Block Bootstrap | 5.75 | 5.75 | 7.50 | 5.00 |
| CI Conformal | 4.75 | 4.50 | 6.75 | 4.50 |
| CI Empirical | 5.50 | 4.00 | 7.25 | 4.50 |
| CI Empirical Residual | 3.50 | 8.00 | 6.25 | 4.25 |
| CI Bonferroni | 10.50 | 8.00 | 5.75 | 5.50 |

**Table 6**. *Performance of forecasting methods on the US births dataset*

| Wrapper | CRPS | PinballLoss | AUCalibration | Runtime |
|---|---:|---:|---:|---:|
| Fallback | 9.25 | 9.50 | 7.88 | 1.00 |
| Naive Variance | 6.25 | 5.75 | 6.25 | 10.00 |
| Squaring Residuals | 9.50 | 8.50 | 6.50 | 11.00 |
| Block Distribution Bootstrap | 1.00 | 1.00 | 8.25 | 7.25 |
| Block Residual Bootstrap | 3.75 | 5.75 | 5.50 | 7.00 |
| Block Statistic Preserving Bootstrap | 5.50 | 6.00 | 7.38 | 6.25 |
| Moving Block Bootstrap | 5.75 | 5.25 | 5.12 | 4.75 |
| CI Conformal | 6.00 | 6.00 | 4.75 | 4.25 |
| CI Empirical | 5.50 | 4.50 | 4.88 | 4.75 |
| CI Empirical Residual | 3.00 | 6.00 | 5.00 | 5.25 |
| CI Bonferroni | 10.50 | 7.75 | 4.50 | 4.50 |

Regarding our benchmark study, we note that this is a limited study primarily aimed at showcasing the capabilities of the proposed framework. Therefore, future work should include a comprehensive hyperparameter search to identify the best parameters for the probabilistic wrappers. Additionally, further bootstrapping methods need to be explored, as well as other wrappers such as generative neural networks, including Generative Adversarial Networks, Variational Autoencoders, or Invertible Neural Networks [32], [33].

Besides extending the range of wrappers, we also plan to include additional point forecasters as base models, such as AutoARIMA, in our study. Furthermore, the number of examined datasets should be expanded to provide a more comprehensive evaluation. Finally, we did not perform a grid search on the hyperparameters for the wrappers, which means that with different hyperparameters, their performance and runtime might change.

In conclusion, the `sktime` evaluation modules enable the performance of reproducible forecasting benchmarks. We demonstrated its applicability in a small benchmarking study that compares different probabilistic wrappers for point forecasters. In future work, we aim to collaborate with the scientific community to integrate more wrappers and conduct a broader benchmark study on this topic.

## References

[1] T. Gneiting and M. Katzfuss, “Probabilistic forecasting,” *Annual Review of Statistics and Its Application*, vol. 1, pp. 125–151, 2014, doi: https://doi.org/10.1146/annurev-statistics-062713-085831.

[2] S. Makridakis, E. Spiliotis, and V. Assimakopoulos, “The M4 Competition: 100,000 time series and 61 forecasting methods,” *International Journal of Forecasting*, vol. 36, no. 1, pp. 54–74, 2020, doi: https://doi.org/10.1016/j.ijforecast.2019.04.014.

[3] S. Makridakis, E. Spiliotis, and V. Assimakopoulos, “M5 accuracy competition: Results, findings, and conclusions,” *International Journal of Forecasting*, vol. 38, no. 4, pp. 1346–1364, 2022, doi: https://doi.org/10.1016/j.ijforecast.2021.11.013.

[4] Y. Chen, Y. Kang, Y. Chen, and Z. Wang, “Probabilistic forecasting with temporal convolutional neural network,” *Neurocomputing*, vol. 399, pp. 491–501, 2020, doi: https://doi.org/10.1016/j.neucom.2020.03.011.

[5] J. Nowotarski and R. Weron, “Recent advances in electricity price forecasting: A review of probabilistic forecasting,” *Renewable and Sustainable Energy Reviews*, vol. 81, pp. 1548–1568, 2018, doi: https://doi.org/10.1016/j.rser.2017.05.234.

[6] K. Rasul *et al.*, “Lag-llama: Towards foundation models for time series forecasting,” *arXiv preprint arXiv:2310.08278*, 2023, doi: https://doi.org/10.48550/arXiv.2310.08278.

[7] A. Das, W. Kong, R. Sen, and Y. Zhou, “A decoder-only foundation model for time-series forecasting,” *arXiv preprint arXiv:2310.10688*, 2023, doi: https://doi.org/10.48550/arXiv.2310.10688.

[8] H. Semmelrock, S. Kopeinik, D. Theiler, T. Ross-Hellauer, and D. Kowald, “Reproducibility in Machine Learning-Driven Research.” [Online]. Available: https://arxiv.org/abs/2307.10320

[9] F. J. Király, B. Mateen, and R. Sonabend, “NIPS - Not Even Wrong? A Systematic Review of Empirically Complete Demonstrations of Algorithmic Effectiveness in the Machine Learning and Artificial Intelligence Literature.” [Online]. Available: https://arxiv.org/abs/1812.07519

[10] F. Király *et al.*, “sktime/sktime: v0.29.0.” [Online]. Available: https://doi.org/10.5281/zenodo.11095261

[11] E. Spiliotis, V. Assimakopoulos, and S. Makridakis, “Generalizing the theta method for automatic forecasting,” *European Journal of Operational Research*, vol. 284, no. 2, pp. 550–558, 2020, doi: https://doi.org/10.1016/j.ejor.2020.01.007.

[12] S. Gilda, “tsbootstrap.” [Online]. Available: https://doi.org/10.5281/zenodo.10866090

[13] S. Gilda, B. Heidrich, and F. Kiraly, “tsbootstrap: Enhancing Time Series Analysis with Advanced Bootstrapping Techniques.” 2024. doi: https://doi.org/10.48550/arXiv.2404.15227.

[14] R. Godahewa, C. Bergmeir, G. Webb, R. Hyndman, and P. Montero-Manso, “Australian Electricity Demand Dataset.” [Online]. Available: https://doi.org/10.5281/zenodo.4659727

[15] R. Godahewa, C. Bergmeir, G. Webb, R. Hyndman, and P. Montero-Manso, “Sunspot Daily Dataset (without Missing Values).” [Online]. Available: https://doi.org/10.5281/zenodo.4654722

[16] R. Godahewa, C. Bergmeir, G. Webb, R. Hyndman, and P. Montero-Manso, “US Births Dataset.” [Online]. Available: https://doi.org/10.5281/zenodo.4656049

[17] M. Löning, A. Bagnall, S. Ganesh, V. Kazakov, J. Lines, and F. J. Király, “sktime: A unified interface for machine learning with time series,” *arXiv preprint arXiv:1909.07872*, 2019, doi: https://doi.org/10.48550/arXiv.1909.07872.

[18] Federico Garza, “StatsForecast: Lightning fast forecasting with statistical and econometric models.” [Online]. Available: https://github.com/Nixtla/statsforecast

[19] J. Perktold *et al.*, “statsmodels/statsmodels: Release 0.14.2.” [Online]. Available: https://doi.org/10.5281/zenodo.10984387

[20] F. Pedregosa *et al.*, “Scikit-learn: Machine learning in Python,” *the Journal of machine Learning research*, vol. 12, pp. 2825–2830, 2011, doi: https://doi.org/10.48550/arXiv.1201.0490.

[21] B. Bischl *et al.*, “mlr: Machine Learning in R,” *Journal of Machine Learning Research*, vol. 17, no. 170, pp. 1–5, 2016, doi: https://doi.org/10.48550/arXiv.1609.06146.

[22] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, “The WEKA data mining software: an update,” *SIGKDD Explor. Newsl.*, vol. 11, no. 1, pp. 10–18, 2009, doi: 10.1145/1656274.1656278.

[23] M. O'Hara-Wild, R. Hyndman, and E. Wang, “fable: Forecasting Models for Tidy Time Series,” 2024. [Online]. Available: https://fable.tidyverts.org/

[24] F. J. Király, M. Löning, A. Blaom, A. Guecioueur, and R. Sonabend, “Designing machine learning toolboxes: Concepts, principles and patterns,” *arXiv preprint arXiv:2101.04938*, 2021, doi: https://doi.org/10.48550/arXiv.2101.04938.

[25] R. J. Hyndman and G. Athanasopoulos, *Forecasting: principles and practice*. OTexts, 2018.

[26] V. Assimakopoulos and K. Nikolopoulos, “The theta model: a decomposition approach to forecasting,” *International Journal of Forecasting*, vol. 16, no. 4, pp. 521–530, 2000, doi: https://doi.org/10.1016/S0169-2070(00)00066-2.

[27] G. Shafer and V. Vovk, “A tutorial on conformal prediction.,” *Journal of Machine Learning Research*, vol. 9, no. 3, 2008, doi: https://doi.org/10.48550/arXiv.0706.3188.

[28] P. Sedgwick, “Multiple significance tests: the Bonferroni correction,” *BMJ*, vol. 344, 2012, doi: 10.1136/bmj.e509.

[29] C. Bergmeir, R. J. Hyndman, and J. M. Benítez, “Bagging exponential smoothing methods using STL decomposition and Box–Cox transformation,” *International Journal of Forecasting*, vol. 32, no. 2, pp. 303–312, 2016, doi: https://doi.org/10.1016/j.ijforecast.2015.07.002.

[30] J. E. Matheson and R. L. Winkler, “Scoring Rules for Continuous Probability Distributions,” *Management Science*, vol. 22, no. 10, pp. 1087–1096, 1976, doi: 10.1287/mnsc.22.10.1087.

[31] I. Steinwart and A. Christmann, “Estimating conditional quantiles with the help of the pinball loss,” 2011, doi: https://doi.org/10.48550/arXiv.1102.2101.

[32] K. Phipps, B. Heidrich, M. Turowski, M. Wittig, R. Mikut, and V. Hagenmeyer, “Generating probabilistic forecasts from arbitrary point forecasts using a conditional invertible neural network,” *Applied Intelligence*, pp. 1–29, 2024, doi: https://doi.org/10.1007/s10489-024-05346-9.

[33] Y. Wang, G. Hug, Z. Liu, and N. Zhang, “Modeling load forecast uncertainty using generative adversarial networks,” *Electric Power Systems Research*, vol. 189, p. 106732, 2020, doi: https://doi.org/10.1016/j.epsr.2020.106732.

Open Access | https://doi.org/10.25080/MCYV2126

# Voice Computing with Python in Jupyter Notebooks

**Blaine H. M. Mooers**[1,2,3]

[1]Department of Biochemistry and Physiology, College of Medicine, University of Oklahoma Health Sciences, Oklahoma City, OK 73104, USA, [2]Laboratory of Biomolecular Structure and Function, University of Oklahoma Health Sciences, Oklahoma City, OK 73104, USA, [3]Stephenson Cancer Center, University of Oklahoma Health Sciences, Oklahoma City, OK 73104, USA

SciPy 2024

July 8 - July 14, 2024

Proceedings of the 23nd Python in Science Conference
ISSN: 2575-9752

## Abstract

Jupyter is a popular platform for writing interactive computational narratives that contain computer code and its output interleaved with prose that describes the code and the output. It is possible to use one's voice to interact with Jupyter notebooks. This capability improves access to those with impaired use of their hands. Voice computing also increases the productivity of workers who are tired of typing and increases the productivity of those workers who speak faster than they can type. Voice computing can be divided into three activities: speech-to-text, speech-to-command, and speech-to-code. We will provide examples of the first two activities with the Voice-In Plus plugin for Google Chrome and Microsoft Edge. To support the editing of Markdown and code cells in Jupyter notebooks, we provide several libraries of voice commands at MooersLab on GitHub.

**Keywords** accessibility, productivity, human-computer interface

## 1. Introduction

Voice computing includes speech-to-text, speech-to-commands, and speech-to-code. These activities enable you to use your voice to generate prose, operate your computer, and write code. Using your voice can partially replace use of the keyboard when tired of typing, suffering from repetitive stress injuries, or both. With the Voice In Plus plugin for Google Chrome and Microsoft Edge, we could be productive within an hour. This plugin is easy to install, provides accurate dictation, and is easy to modify to correct wrong word insertions with text replacements.

We mapped spoken words to be replaced, called *voice triggers*, to equations set in LaTeX and to code snippets that span one to many lines. These *voice-triggered snippets* are analogous to traditional tab-triggered snippets supported by most text editors. (A tab trigger is a placeholder word replaced with the corresponding code when the tab key is pressed after entering the tab trigger. The existing extensions for code snippets in Jupyter do not support tab triggers.) We could use Voice In Plus to insert voice-triggered snippets into code and Markdown cells in Jupyter notebooks. Our voice-triggered snippets still require customizing to the problem at hand via some use of the keyboard, but their insertion by voice command saves time.

To facilitate voice commands in Jupyter notebooks, we have developed libraries of voice-triggered snippets for use in Markdown or code cells with the *Voice-In Plus* plugin. We are building on our experience with tab-triggered code snippets in text editors [1] and domain-specific code snippet libraries for Jupyter [2]. We have made libraries of these voice-triggered snippets for several of the popular modules of the scientific computing stack

**Published** Jul 10, 2024

**Correspondence to**
Blaine H. M. Mooers
blaine-mooers@ouhsc.edu

for Python. These voice-triggered snippets are another tool for software engineering that complements existing tools for enhancing productivity.

## 2. Methods and Materials

### *2.1. Hardware*

We used a 2018 15-inch MacBook Pro laptop computer. It had 32 gigabytes of RAM and one Radeon Pro 560X 4 GB GPU. We used the laptop's built-in microphone to record dictation while sitting or standing up to 20 feet (ca. 6 m) away from the computer.

### *2.2. Installation of Voice In Plus*

We used the *Voice In* plugin provided by Dictanote Inc. First, we installed the *Voice In* plugin by navigating to the Plugin In page in the Google Chrome Web Store on the World Wide Web. Second, the Microsoft Edge Addons web site was accessed to install the plugin in Microsoft Edge.

We needed an internet connection to use *Voice In* because Dictanote tracks the websites visited and whether the plugin worked on those websites. *Voice In* uses the browser's built-in Speech-to-Text software to transcribe speech into text, so no remote servers are used for the transcription,/ so the transcription process was nearly instant and kept up with the dictation of multiple paragraphs. Dictanote does not store the audio or the transcripts.

After activating the plugin, we customized it by selecting a dictation language from a pull-down menu. We selected **English (United States)** from the 15 dialects of English. The English variants include dialects from Western Europe, Africa, South Asia, and Southeast Asia; many languages other than English are also supported.

Next, we set a keyboard shortcut to activate the plugin. We selected command-L on our Mac because this shortcut was not already in use. A list of Mac shortcuts can be found here. This second customization was the last one that we could make for the free version of *Voice In*.

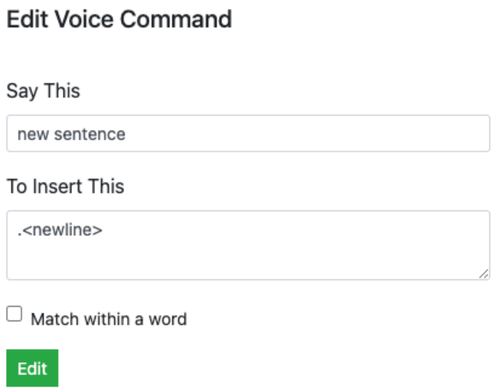
Customized text replacements are available in *Voice In Plus*. *Voice In Plus* is activated by purchasing a $39 annual subscription through the **Subscription** sub-menu in the *Voice In Settings* sidebar of the *Voice In Options* web page. Monthly and lifetime subscription options are available. Only one subscription was required to use *Voice In Plus* in both web browsers. The library is synched between the browsers.

After purchasing a subscription, we accessed the **Custom Commands** in the *Voice In Settings* sidebar. We used the **Add Command** button to enter a voice trigger to start a new sentence when writing one sentence per line Figure 1. The custom command for `new paragraph` can include a `period` followed by two `new line` commands, which works well when writing with blank lines between paragraphs and without indentation.

The phrases making up these voice triggers will no longer be available during dictation because they will be used to trigger text replacements. It is best to pick as voice triggers phrases that are unlikely to be spoken during dictation.

We kept the voice triggers as short as possible to ease accurate recall of the voice triggers. We had trouble correctly recalling voice triggers longer than four or five words. Three-word voice triggers are a good compromise between specificity and success at recall.

An exception to this guideline for shorter voice triggers was using two to three words at the beginning of a set of voice triggers to group them in the online *Voice In* library. Grouping related voice commands made finding them in the online library easier. For example, all Jupyter-related line magic voice triggers start with the phrase *line magic*. The prefix *line*



Edit Voice Command

Say This

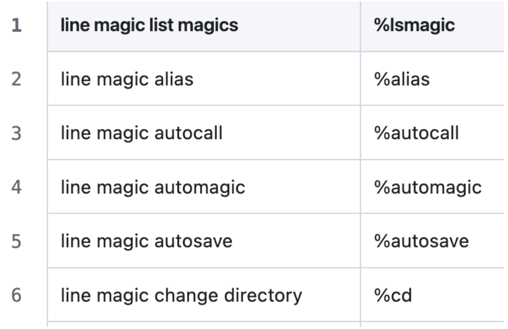
new sentence

To Insert This

.<newline>

☐ Match within a word

**Figure 1.** *Entering a single voice trigger and the corresponding command in Voice In Plus.*



| line magic list magics | %lsmagic |
|---|---|
| line magic alias | %alias |
| line magic autocall | %autocall |
| line magic automagic | %automagic |
| line magic autosave | %autosave |
| line magic change directory | %cd |

**Figure 2.** *Snapshot of CSV file on Github for the* jupyter-voice-in *library.*

*magic* is easy to remember, so it only adds a little to the recall problem. We show below a snapshot of the CSV file displayed in a pleasant format on GitHub for the Jupyter line magics Figure 2. Note that these CSV files are atypical because they do not contain a line of column headers; *Voice In* does not recognize column headings.

The accuracy of the software's interpretation of the phrase was another limitation. We would replace frequently misinterpreted words with other words that were more often correctly interpreted.

We had to avoid making voice triggers that overlapped longer voice triggers to avoid unintended text replacements. For example, when we found a two-word phrase starting a four-word phrase, we would extend the two-word phrase to three words by selecting a third word to break the overlap between the two voice triggers. Otherwise, the two-word phrase would trigger a text replacement different from the text replacement expected from the four-word phrase.

We used the **Bulk Add** button to upload multiple commands from a two-column CSV file with commas as the field separator. We selected the file contents and pasted them in the open text box after clicking the **Bulk Add** button. The voice triggers reside in the left column, and the text replacements reside in the right column. The software ignored any capitalization in the voice trigger.

We handled multiline text replacements in one of two ways. First, we placed all the text on a single line and inserted the built-in command `<newline>` where linebreaks were needed. Second, we enclosed the multiline replacement text with one set of double quotes. Double quotes inside these text blocks had to be replaced with single quotes. We could use a backslash to escape internal pre-existing double quotation marks. Text replacements consisting of commas also had to be enclosed with double else the commas would be misinterpreted as field separators. This also applied on Python code that contained commas.

We used the **Bulk Add** button to upload multiple commands from a two-column CSV file with commas as the field separator. We selected the file contents and pasted them in the open text box after clicking the **Bulk Add** button. The voice triggers reside in the left column, and the text replacements reside in the right column. The software ignored any capitalization in the voice trigger.

We handled multiline text replacements in one of two ways. First, we placed all the text on a single line and inserted the built-in command `<newline>` where linebreaks were needed. Second, we enclosed the multiline replacement text with one set of double quotes. Double quotes inside these text blocks had to be replaced with single quotes. We could not use a backslash to escape internal pre-existing double quotation marks.

The **Export** button opened a text box with the custom commands in CSV file format. We selected all the text box contents, copied them, and pasted them into a local CSV file using either the text editor TextMate or Emacs version 29.3. We used the **Synch** button to synchronize devices.

A GUI shows all the voice triggers and their text replacements immediately below the row of buttons mentioned above. Each row in the GUI has edit and delete icons. The edit icon opens a pop-up menu similar to the pop-up menu invoked by the **Add Command** button.

### *2.3. Construction of the snippet libraries*

Some of our voice snippets had already been used for a year to compose prose using dictation. These snippets are in modular CSV files to ease their selective use. The contents of these files can be copied and pasted into the `bulk add` text area of the Voice In Plus configuration GUI.

### *2.4. Construction of interactive quizzes*

We developed an interactive quiz to aid the mastery of the *Voice In Plus* syntax. We wrote the quiz as a Python script that can run interactively in the terminal or Jupyter notebooks. The quiz randomizes the order of the questions upon restart, ensuring a fresh experience every time. When you encounter a question you cannot answer, the quiz steps in to provide feedback, empowering you to learn from your mistakes and improve. Your wrongly answered questions are recycled during the current quiz session to allow you to answer the question correctly. This repetition helps build recall of the correct answers. We set a limit of 40 questions per quiz, allowing you to pace your learning and avoid exhaustion.

### *2.5. Availability of the libraries and quizzes*

We tested the libraries using Jupyter Lab version 4.2 and Python 3.12 installed from MacPorts. All libraries are available at MooersLab on GitHub for download.

## 3. Results

First, we describe the contents of the snippet libraries and the kinds of problems they solve. We group the libraries into several categories to simplify their explanation. Second, we describe the deployment of the snippet libraries for Voice In Plus (VIP).

### *3.1. Composition of the libraries*

Our descriptions of these libraries include examples to illustrate how voice-triggered snippets work with automated speech recognition software. Developers can leverage our libraries in two ways: by enhancing them with their unique commands or by using them as blueprints to create libraries from scratch.

The libraries are made available in a modular format, so the user can select the most valuable commands for their workflow. In general, our libraries are broad in scope, so they meet the needs of most users. Several libraries are domain-specific. These domain-specific libraries catalyze the creation of libraries tailored to other fields, sparking innovation and expanding the reach of voice-triggered snippets.

We divided the contents of the libraries into two categories. One subset of libraries supports dictating about science in the Markdown cells of Jupyter notebooks, while the other subsets support writing scientific Python in code cells. You can also apply these voice-triggered snippets to Markdown and code cells in Colab Notebooks. While some code, such as IPython line and cell magics, is specific to Jupyter, you can use most voice-triggered snippets to edit Markdown and Python files in Jupyter Lab.

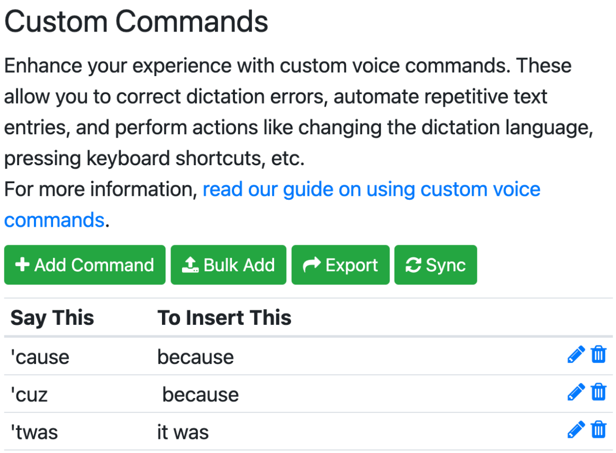
Likewise, you can use these snippets in other browser-hosted text editors, such as the web version of Visual Studio Code because Voice In Plus works in most text areas of web browsers. You can use web services with ample text areas to draft documents and Python scripts with the help of voice-triggered snippets. You can also use Voice In Plus inside text boxes in local HTML files; however, Voice In Plus still requires an internet connection. You can edit the code or text with an advanced text editor using the GhostText plugin to connect a web-based text area to a text editor. Alternatively, you can save a draft document or script to a file and import it into Jupyter Lab for further editing off line.

### *3.2. Libraries for Markdown cells*

These libraries contain a short phrase paired with its replacement: another phrase or a chunk of computer code. In analogy to a tab trigger in text editors, we call the first short phrase a voice trigger. Tab triggers are initiated by typing the tab trigger, followed by the tab key, which inserts the corresponding code snippet. Some text editors can autocomplete the tab-trigger name, so these text editors require two tab key entries. The first tab auto-completes the tab-trigger name. Then, the second tab leads to the insertion of the corresponding code. This two-step process of tab triggers empowers users with the flexibility to select a different tab trigger before inserting the code, enhancing the customization potential of text editors. It is important to note that voice triggers, while efficient, do not allow for the revision of the voice trigger. In the event of an incorrect code insertion, the undo command must be used, underscoring the need for caution when selecting voice triggers.

The most straightforward text replacements involved the replacement of English contractions with their expansions Figure 3. Science writers do not use English contractions in formal writing for many reasons. Many automatic speech recognition software packages will default to using contractions because the software's audience is people who write informally for social media, where English contractions are acceptable. Adding the library that maps contractions to their expansions will insert the expansion whenever the contraction is spoken. This automated replacement of English contractions saves time during the editing process.

We grouped the voice triggers by the first word of the voice trigger Table 1. This first word is often a verb. For instance, the word `expand' is used before an acronym you want to replace with its expansion. Your memory of the words represented by the letters in the acronym often must be accurate. The library ensures that you use the correct expansion of the acronym. By providing instant access to the correct acronym expansions, the library significantly reduces the time that would otherwise be spent on manual lookups, allowing you to focus on your work. We have included acronyms widely used in Python programming,



## Custom Commands

Enhance your experience with custom voice commands. These allow you to correct dictation errors, automate repetitive text entries, and perform actions like changing the dictation language, pressing keyboard shortcuts, etc.
For more information, read our guide on using custom voice commands.

| Say This | To Insert This |
|---|---|
| 'cause | because |
| 'cuz | because |
| 'twas | it was |

**Figure 3**. *Webpage of custom commands. The buttons are used to edit existing commands and add new commands. Three English contractions and their expansions are shown.*

scientific computing, data science, statistics, machine learning, and Bayesian data analysis. These commands are found in domain-specific libraries, so users can select which voice triggers to add to their private collection of voice commands.

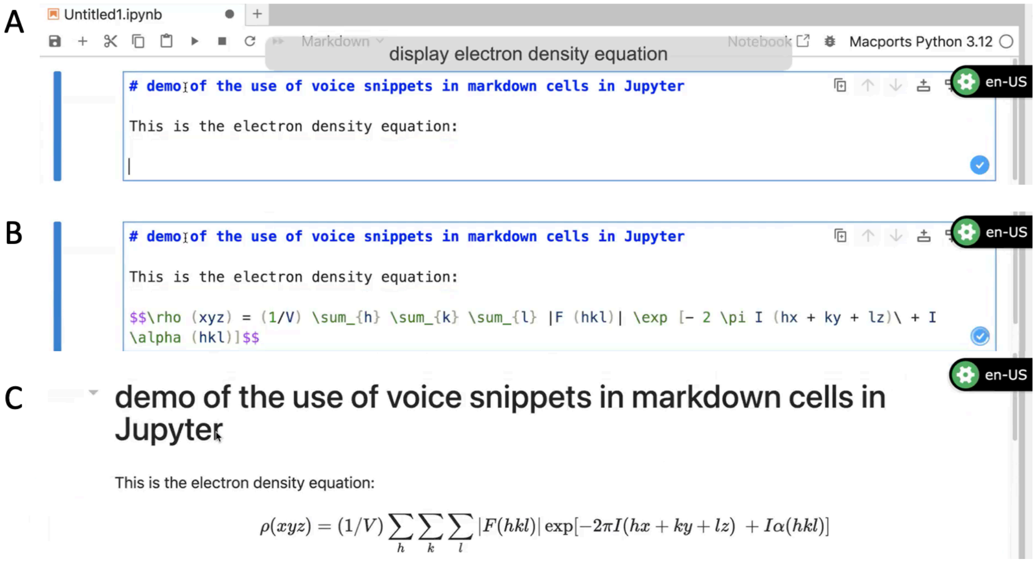
The customized commands are listed alphabetically in the Voice-In Plus GUI, with one command and its corresponding text replacement per row. The prefixes group like commands and, thereby, ease the manual lookup of the commands.

Some prefixes are two or more words long. For example, the compound prefix *insert Python* aids the grouping of voice triggers by programming language.

Another example of a verb starting a voice trigger is the command `display <equation name>`. This command is used in Markdown cells to insert equations in the display mode of LaTeX in Markdown cells. For instance, the voice trigger `display the electron density equation` is a testament to the convenience of our system, as shown in the transcript of a Zoom video Figure 4. The image in the middle shows the text replacement as a LaTeX equation in the display mode. This image is followed by the resulting Markdown cell after rendering by running the cell.

**Table 1**. *Examples of voice commands with the prefix in bold that is used to group commands.*

| Voice commands |
|---|
| **expand** acronyms |
| **the book** title |
| **email** inserts list of e-mail addresses (e.g., email bayesian study group) |
| **insert Python** (e.g., insert Python for horizontal bar plot) |
| **insert Markdown** (e.g., insert Markdown header 3) |
| **list** (e.g., list font sizes in beamer slides.) |
| **open** webpage (e.g., open google scholar) |
| **display** insert equation in display mode (e.g., display electron density equation) |
| **display with terms** above plus list of terms and their definitions (e.g., display electron density equation) |
| **inline** equation in-line (e.g., inline information entropy) |
| **site** insert corresponding citekey (e.g., site Jaynes 1957) |



display electron density equation

```
# demo of the use of voice snippets in markdown cells in Jupyter

This is the electron density equation:

```

```
# demo of the use of voice snippets in markdown cells in Jupyter

This is the electron density equation:

$$\rho (xyz) = (1/V) \sum_{h} \sum_{k} \sum_{l} |F (hkl)| \exp [- 2 \pi I (hx + ky + lz)\ + I
\alpha (hkl)]$$
```

demo of the use of voice snippets in markdown cells in Jupyter

This is the electron density equation:

$$\rho(xyz) = (1/V) \sum_{h} \sum_{k} \sum_{l} |F(hkl)| \exp[-2\pi I(hx + ky + lz)\ + I\alpha(hkl)]$$

**Figure 4**. *Three snapshots from a Zoom video of using the voice-trigger* display electron density equation *in a Markdown cell in a Jupyter notebook. A. The Zoom transcript showing the spoken voice trigger. B. The text replacement in the form of a math equation written in LaTeX in display mode in the Markdown cell. C. The rendered Markdown cell. The green and black tab on the right of each panel indicates that the Voice In plugin is active and listening for speech.*

Likewise, the command `inline <equation name>` is used to insert equations in prose sections in Markdown cells. We have introduced voice-triggered snippet libraries for equations commonly used in machine learning and Bayesian data analysis https://github.com/MooersLab/. You can use these equations as templates to generate new equations. While the development of libraries is still in progress, they serve as flexible templates that you can adapt to your domain.

Some voice triggers start with a noun. For example, the voice trigger `URL` is used to insert URLs for essential websites.

Another example involves using the verb `list` in the voice trigger, as in `list matplotlib color codes`, to generate a list of the color codes used in Matplotlib plots. These voice triggers provide instant access to essential information, saving you the time and effort of manual searches.

The markup language code is inserted using the verb *insert*, followed by the markup language name and the name of the code. For example, the command `insert markdown itemized list` will insert five vertically aligned dashes to start an itemized list. The command `insert latex itemized list` will insert the corresponding code for an itemized list in LaTeX.

We have developed a library specifically for the flavor of Markdown utilized in Jupyter notebooks. This library is used to insert the appropriate Markdown code in Markdown cells. We have included a library for LaTeX because Jupyter Lab can edit tex files.

We have yet to figure out how to use voice commands to advance the cursor in a single step to sites where edits should be made, analogous to tab stops in conventional snippets. Instead, the built-in voice commands can move the cursor forward or backward and select replacement words. We included the markup associated with the yasnippet snippet libraries to serve as a benchmark for users to recognize the sites that should be considered for modification to customize the snippet for their purpose.

### 3.3. *Libraries for code cells*

The `insert` command in code-cell libraries allows the insertion of chunks of Python code, enhancing your data science workflow. We avoid making voice commands for small code fragments that might fit on a single line. An exception to this was the inclusion of a collection of one-liners in the form of several kinds of comprehensions. As mentioned, we have developed practical chunks of code designed to perform specific functions, making your data analysis tasks more efficient. These chunks of code are functions that produce an output item, such as a table or plot. The idea was to fill a code cell with the code required to produce the desired output. Unfortunately, the user will still have to use the computer mouse to move the cursor back over the code chunk and customize portions of the code chunk as required.

While self-contained examples that utilize generated data can illustrate concepts, these examples are frustrating for beginners who need to read actual data and would like to apply the code example to their problem. Reading appropriately cleaned data is a common task in data science and a common barrier to applying Jupyter notebooks to scientific problems. Our data wrangling library provides code fragments that directly import various file types, easing the task of data import and allowing focus on downstream utilization and analysis.

After the data are verified as correctly imported, exploring them by plotting them to detect relationships between a model's parameters and the output is often necessary. Our focus on the versatile *matplotlib* library, which generates various plots, is designed to inspire creativity in data visualization and analysis [*matplotlib]. Our code fragments cover the most commonly used plots, such as scatter plots, bar graphs (including horizontal bar graphs), kernel density fitted distributions, heat Maps, pie charts, and contour plots. We include a variety of examples for formatting tick marks and axis labels as well as the keys and the form of the lines so users can use this information as templates to generate plots for their purposes. Generating plots with lines of different shapes, whether solid, dashed, dotted, or combinations thereof, is essential because plots generated with just color are vulnerable to having their information compromised when printed in grayscale. Although we provide some examples from higher-order plotting programs like *seaborn* [3], we focused on matplotlib because most other plotting programs, except the interactive plotting programs, are built on top of it.

We also support the import of external images. Images often play essential roles in the stories told with Jupyter notebooks.

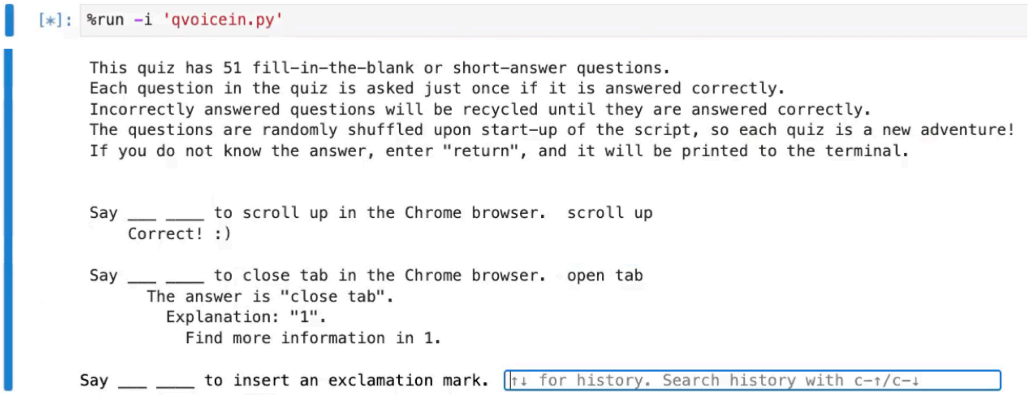
### 3.4. *Jupyter specific library*

We provide a library of 85 cell and line magics that facilitate the Jupyter notebook's interaction with the rest of the operating system. Our cell magics, easily identifiable by their cell magic prefix, and line magics, with the straightforward line magic prefix, are designed to make the Jupyter notebook experience more intuitive. For example, the voice command *line majic run* inserts `%run`. You use this command to run a script file Figure 5.

### 3.5. *Interactive quiz*

We developed a quiz to improve recall of the voice commands. These quizzes are interactive and can be run in the terminal or in Jupyter notebooks Figure 5. The latter can store a record of your performance on a quiz.

To build long-term recall of the commands, you must take the quiz five or more times on alternate days, according to the principles of spaced repetition learning. These principles were developed by the German psychologist Hermann Ebbinghaus near the end of the 19th Centur and popularized in English his 1913 book *Memory: A Contribution to Experimental*



```
[*]: %run -i 'qvoicein.py'

       This quiz has 51 fill-in-the-blank or short-answer questions.
       Each question in the quiz is asked just once if it is answered correctly.
       Incorrectly answered questions will be recycled until they are answered correctly.
       The questions are randomly shuffled upon start-up of the script, so each quiz is a new adventure!
       If you do not know the answer, enter "return", and it will be printed to the terminal.


       Say ___ ____ to scroll up in the Chrome browser.  scroll up
           Correct! :)

       Say ___ ____ to close tab in the Chrome browser.  open tab
             The answer is "close tab".
               Explanation: "1".
                 Find more information in 1.

      Say ___ ____ to insert an exclamation mark.  ↑↓ for history. Search history with c-↑/c-↓
```

**Figure 5**. *An example of an interactive session with a quiz in a Jupyter notebook. The code for running the quiz was inserted into the code cell with the voice command* `run voice in quiz`*. The quiz covers a range of voice commands, including [specific voice commands covered in the quiz].*

*Psychology*. They have been validated several times by other researchers, including those developed an algorithm for optimizing the spacing of the repetitions as function of the probability of recall [4]. Spaced repetition learning is one of the most firmly established results of research into human psychology.

Most people need discipline to carry out this kind of learning because they have to schedule the time to do the follow-up sessions. Instead, people will find it more convenient to take these quizzes several times in 20 minutes before they spend many hours utilizing the commands. If that use occurs on subsequent days, then the recall will be reinforced and retaking the quiz may not be necessary.

### 3.6. Limitations on using Voice In Plus

The plugin operates in text areas on thousands of web pages. These text areas include those of web-based email software and online sites that support distraction-free writing (e.g., Write Honey). These text areas also include the Markdown and code cells of Jupyter notebooks and other web-based computational notebooks such as Colab Notebooks. Voice In also works in plain text documents opened in Jupyter Lab for online writing.

Voice In Plus works in Jupyter Lite. It also works in streamlet-quill, which uses a Python script to generate a text box in the default web browser. It also works in the web-based version of VS Code.

Voice In will not work in desktop applications that support the editing of Jupyter notebooks, such as the *JupyterLab Desktop* application, the *nteract* application, and external text editors, such as *VS Code*, that support the editing of Jupyter notebooks. Likewise, *Voice In Plus* will not work in Python Widgets. *Voice In Plus* is limited to web browsers, whereas other automated speech recognition software can also operate in the terminal and at the command prompt in GUI-driven applications.

*Voice In Plus* is very accurate, with a word error rate that is well below 10%. Like all other dictation software, the word error rate depends on the quality of the microphone used. *Voice-In Plus* can pick out words from among background ambient noise such as load ventilation systems, traffic, and outdoor bird songs.

The language model used by *Voice-In Plus* is quite robust in that dictation can be performed without an external microphone. We found no reduction in word error rate when using a high-quality Yeti external microphone. Our experience might be a reflection of our high-end hardware and may not transfer to low-end computers.

Because of the way *Voice-In Plus* is set up to utilize the Speech-to-Text feature of the Google API, there is not much of a latency issue. The spoken words' transcriptions occur nearly in real-time; there is only a minor lag. *Voice In Plus* will listen for words for 3 minutes before automatically shutting down. *Voice In Plus* can generally keep up with dictation occurring at a moderate pace for at least several paragraphs, whereas competing dictation software packages tend to quit after one paragraph. The program hallucinates when the dictation has occurred at high speed because the transcription has fallen behind. You have to pay attention to the progress of the transcription if you want all of your spoken words captured. If the transcription halts, it is best to deactivate the plugin, activate it, and resume the dictation.

Pronounce the first word of each sentence loudly so that they are recorded. Otherwise, the first words of your sentences will be skipped. This problem of omitted words is most acute when there has been a pause in the dictation.

The software does not automatically insert punctuation marks. You have to vocalize the name of the punctuation mark where it is required. You also have to utilize the built-in new-line command to start new lines. We have combined the `period` command with the `new line` command to create a new command with the voice trigger of `new sentence`.

You have to develop the habit of using this command if you like to write one sentence per line. This latter form of writing is helpful for first drafts because it eases the shuffling of sentences in a text editor during rewriting. This form of writing is also compatible with version control systems like git, which track changes by line number.

The practical limit on the number of commands is set by the trouble you will tolerate when scrolling up and down the list of commands. We had an easy time scrolling through a library of about 8,000 commands and a hard time with a library of about 19,000 commands. Bulk deletion of selected commands required assistance from the user support at Dictanote Inc. They removed our bloated library, and we used the **bulk add** button in the GUI to upload a smaller version of our library.

## 4. Discussion

The following four discussion points can enhance understanding the use of *Voice-In Plus* with Jupyter.

### *4.1. Independence from breaking changes in Jupyter*

The Jupyter project lacks built-in support for libraries of code snippets. Developing third-party extensions is not just a choice but a crucial necessity to support using code snippets in the Jupyter environment. Unfortunately, changes in the core of Jupyter occasionally break these third-party extensions. Users are burdened with the task of creating Python environments for older versions of Jupyter to work with their favorite outdated snippet extension, all the while missing out on the new features of Jupyter. An obvious solution to this problem would be for the Jupyter developers to incorporate one of the snippet extensions into the base distribution of Jupyter to ensure that at least one form of support for snippets is always available. Using voice-triggered snippets external to Jupyter side steps the disruption of snippet extensions by breaking changes in new versions of Jupyter.

### *4.2. Voice-triggered snippets can complement AI-assisted voice computing*

The use of voice-triggered snippets requires knowledge of the code that you want to insert. The act of acquiring this knowledge is the up-front cost that the user pays to gain access to quickly inserted code snippets that work. In contrast, AI coding assistants can find code that you do not know about to solve the problem described in your prompt. From personal experience, the retrieval of the correct code can take multiple iterations of refining the prompt. Expert prompt engineers will find the correct code in several minutes while beginners may take much longer. An area of future research is to use AI assistants that have large language models indexed on snippet libraries to retrieve the correct voice-triggered snippet.

### 4.3. *Comparision with automated speech recogination extensions for Jupyter lab*

We found three extensions developed for Jupyter Lab that enable speech recognition in Jupyter notebooks. The first, jupyterlab-voice-control, supports custom commands and relies on the browser's language model. However, it is important to note that this extension is experimental and does not currently work with Jupyter 4.2. The second extension, jupyter-voice-comments, relies on the DaVinci large language model to make comments in Markdown cells and request code fragments. This program requires clicking on a microphone icon frequently, which makes the user vulnerable to repetitive stress injuries. The third extension, jupyter-voicepilot, is designed to provide a unique voice control experience. Although the extension's name suggests it uses GitHub's Copilot, it uses whisper-1 and ChatGPT3. This extension requires an API key for ChatGPT3. The robustness of our approach is that the *Voice-In Plus* should work in all browser-based versions of Jupyter Lab and Jupyter Notebook.

### 4.4. *Coping with the imperfections of the language model*

One of the most significant challenges in speech-to-text technology is the occurrence of persistent errors in transcription. These persistent errors may be due to the language model having difficulties interpreting your speech. For example, the language model often misinterprets the word *write* as *right*'. Likewise, the letter *R* is frequently returned as *are* or *our*'. It's crucial to have a remedy for these situations, which involves mapping the misinterpreted phrase to the intended phrase.

This remedy might be the best that can be done for those users who are from a country that is not represented by the selection of English dialects available in Voice In Plus. People from Eastern Europe, the Middle East, and Northeast Asia fall into this category. Users in this situation may have to add several hundred text replacements. As the customized library of text replacements grows, the frequency of wrong word insertions should decrease significantly, offering hope for improved accuracy in your speech-to-text transcriptions.

## 5. Future directions

Our future directions include building out the libraries of voice-triggered snippets. Another direction includes the development of voice stops analogous to tab stops in code snippets for advanced text editors. The cursor would advance to the site of the next voice stop that you should consider editing to customize the code snippet for the project at hand. Another related advancement would be the concept of mirroring the parameter values at identical voice stops, similar to the functionality of mirrored tab stops in conventional snippets.

## References

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

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

[3] M. L. Waskom, "seaborn: statistical data visualization," *Journal of Open Source Software*, vol. 6, no. 60, p. 3021, 2021, doi: 10.21105/joss.03021.

[4] B. Tabibian, U. Upadhyay, A. De, A. Zarezade, B. Schölkopf, and M. Gomez-Rodriguez, "Enhancing human learning via spaced repetition optimization," *Proceedings of the National Academy of Sciences*, vol. 116, no. 10, pp. 3988–3993, 2019, doi: 10.1073/pnas.1815156116.

Open Access | https://doi.org/10.25080/YCFW5807

SciPy 2024

July 8 - July 14, 2024

Proceedings of the 23nd
Python in Science Conference
ISSN: 2575-9752

# Predx-Tools: Dispelling the Mystery in Histopathological Image Processing

**Brian Falkenstein**[1], **Shannon Quinn**[1,2], **Chakra Chennubhotla**[1,3], **Filippo Pullara**[1], and **Raymond Yan**[1]

[1]PredxBio, Inc., [2]School of Computer Science, Department of Cellular Biology, University of Georgia, [3]Dept of Computational and Systems Biology, University of Pittsburgh School of Medicine

## Abstract

Histopathological images, which are digitized images of human or animal tissue, contain insights into disease state. Typically, a pathologist will look at a slide under a microscope to make decisions about prognosis and treatment. Due to the high complexity of the data, applying automatic image analysis is challenging. Often, human intervention in the form of manual annotation or quality control (QC) is required. Additionally, the data itself varies considerably in available features, size, and shape. Thus, a streamlined and interactive approach is a necessary part of any digital pathology pipeline. We present PredX-Tools, a suite of simple and easy to use python GUI applications which facilitate analysis of histopathological images and provide a no-code platform for data scientists and researchers to perform analysis on raw and transformed data.

**Keywords** digital pathology, brightfield, multiplex immunofluorescence, point process

## 1. Introduction

Histopathological image analysis reveals insights into disease state and has potential to harness the immensely complex and heterogeneous tumor microenvironment to guide understanding of disease progression, response to treatment, and aid in patient stratification for clinical trials [1], [2], [3], [4], [5]. However, applying automatic image analysis routines continues to be challenging [6], [7]. Often, human intervention in the form of manual annotation or quality control (QC) is required. Additionally, the data itself varies considerably in available features, size, and shape. For example, the available biomarkers vary in quantity between datasets, with some having only a few channels and others having hundreds or more. Images vary from small core samples (∼1 MB) to massive whole slides images (∼ 100 GB). Some datasets may contain data of multiple types with additional considerations for aligning the data to a common reference frame. Thus, a streamlined and interactive approach is a necessary part of any digital pathology pipeline [8].

We present PredX-Tools, a suite of simple and easy to use python GUI applications which facilitate all parts of a digital pathology pipeline, from image QC and labeling to visualization and analysis of results to allow for a deeper understanding of the data. By utilizing open-source python libraries and utilities, we have created a suite of tools that fit seamlessly into our analysis pipeline, require minimal expertise to run, and are easy to develop new features for, to account for the ever-changing landscape of histopathological image analysis.

## 2. Background

Raw data takes the form of multi-channel images, ranging in depth from 3 channels (RGB) up to several hundreds or even thousands. Although specifics of the routine vary depending

**Published** Jul 10, 2024

**Correspondence to**
Brian Falkenstein
brian.falkenstein@predxbio.com

on the available data and the underlying biological questions, every experiment follows a similar procedure. Images must be registered, normalized, and denoised by a data scientist. Registration is the process of aligning subsequent sections of tissue in space via rotation and translation or affine transformation to match keypoints between images. Then, a researcher performs nuclei and or cell segmentation, where each nuclear and cellular object has a boundary automatically drawn around it. Additionally, other objects in the images may be segmented, such as tumor beds or other biological structures present in the image. Next, features are extracted for the segmented objects and stored in data tables. Common cell level features include a measurement of biomarker strength within the cell boundaries, or morphological measurements about the cell object like its area or ellipticity. Finally, these tables of features, which include the spatial location of the objects, may be analyzed by machine learning scientists to derive answers to mechanistic questions or to find differences between populations.

While there have been attempts at fully automating this procedure, it is nearly impossible to design routines that can account for all forms of variation across datasets. As an example, expected size of a cell may vary wildly between different types of cancer, and thus having one threshold for calling a cell ‘large’ or ‘small’ is challenging. Thus, for optimal performance on new data, it is ideal to have a human-in-the-loop to perform manual QC, parameter tuning, and probing of the data to mitigate the error that arises when performing analysis on new datasets. This leads to a higher confidence in any results that may be found. To facilitate this, simple and easy to use tools must be developed to allow field experts to interact with the data with minimal effort. Oftentimes users may not be comfortable or familiar with programming, and thus it is far more efficient to design no-code interactive environments to facilitate interaction with the data for non-coding domain experts.

Other tools have been developed to address these needs (see Table 1), however none cover all our needs and many are rigid and allow custom integration only through script writing. There is a need for a stripped down suite of modular solutions that both fits specifically our needs for analysis, while also being modular and easy to develop to fit arising needs and challenges that come with new data. Hence, we have developed our own in-house suite of solutions that allows an expert user to plug directly into our highly complex analysis pipelines, both to check quality of generated results and allow deeper exploration of the data. More specifically, we are addressing 4 separate needs required in our analysis routine. First, we provide an environment for gathering ground truth labels from an expert with `CellLabeler`. Second, we provide a simple hub for analyzing the quality of raw and preprocessed image data with `PixelExplorer`. Third, we created `CellExplorer` for visualizing data tables of cellular object features and phenotypes, as well as the ability to fit models and export modified versions of the data tables. Last, a simple and easy to use app for performing complex spatial analysis of the cellular or nuclear objects is provided in `SpaceExplorer`. All apps were created with PySimpleGUI [9], alongside other open source python libraries Matplotlib [10], NumPy [11], pandas [12], [13], scikit-learn [14], [15], SciPy [16], Tifffile [17], and zarr [18].

## 3. CellLabeler

We introduce CellLabeler, a flexible and easy to configure GUI application for the purpose of gathering expert annotations on histopathological image data. One common issue in the space of histopathological image analysis is the lack of labeled datasets. Due to the wide heterogeneity of the data, it is often the case that the existing labeled datasets are useless for the dataset in question. For example, current analysis may be focused on a panel of biomarkers that are unique from any labeled dataset in the public domain. Alternatively, one may be working with a tissue or disease type that has little or no representation in public domain datasets. Thus, it proves useful to have a tool to quickly and efficiently provide

| Software | View Raw | Label | Pixel analysis | Mask generation | Phenotyping | Spatial analysis | Easy development |
|---|---|---|---|---|---|---|---|
| `Fiji` [19] | Yes* | Yes | Yes | Yes | No | No | Yes |
| `QuPath` [20] | Yes | Yes | Yes | Yes | Yes | No | Yes |
| `CellProfiler` [21] | Yes* | No | Yes | Yes | Yes | Yes | Yes |
| `Ilastik` [22] | Yes* | Yes | Yes | Yes | Yes | No | No |
| **Predx-Tools** | Yes | Yes | Yes | Yes | Yes | Yes | Yes |

**Table 1**. *Comparison of available softwares.*

*"*" = Limitation on image size*

labels to new data to be used for either quality control purposes (labeling a segmentation as good or bad) or for training a machine learning algorithm (labeling a cell by its type).

A configuration json file must be created for each project. Within the configuration file, the user specifies the possible labels that can be assigned to each object, the type of image data (RGB or N-stack) and which channels to show by default (N-stack only). The input data consists of cropped images of each object to be labeled, as well as an optional list of marker names in the image (N-stack only).

Once the experiment has begun and the data is loaded, the user will be shown an image and is asked to provide a label. The potential labels are listed in a table, and can be clicked to assign that label to the current image. To enable faster assignment of labels, the number pad may also be used to assign labels. The user can also take time to modify which biomarkers are being displayed, as well as which color each biomarker is assigned to (N-stack only). By default, the object to be labeled has its boundary drawn on the image, however this can be toggled on and off. Once a label has been assigned to the current image, the user can either click next or use the right arrow to navigate to the next image (similarly, left arrow or back to navigate to the previous labeled image). After each step, a simple save file is generated which tracks the labeled objects, and thus progress is never lost. It is simple to load a previously saved project to pick up where you left off. This provides a simple and easy to use app for gathering labels for objects for the purposes of training an algorithm or performing QC.

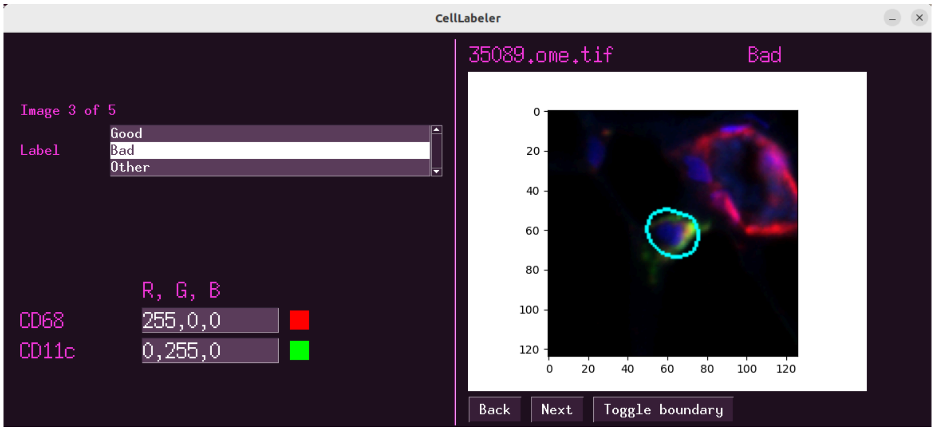


Image 3 of 5

Label: Bad

35089.ome.tif Bad

| | R, G, B | |
|---|---|---|
| CD68 | 255,0,0 | 🟥 |
| CD11c | 0,255,0 | 🟩 |

**Figure 1**. *The interface for CellLabeler.*

## 4. PixelExplorer

We have also created PixelExplorer as a way to quickly perform basic analysis on pixel level data in raw or pre-processed histopathological datasets. Analyzing biomarker distributions and fitting models is a crucial step of preprocessing and quality control of histopathological imaging data. In brightfield imaging, one may want to analyze the distribution of raw RGB signals before and after normalization or stain deconvolution, where the individual stains are separated out from the RGB into separate channels. In mIF or IMC data, it is crucial to understand the behavior and dynamic range of the various biomarkers and how they are affected by normalization and noise reduction methods.

The input data for PixelExplorer is simply a folder containing the .tif (or any other open source microscopy image format, such as .ome.tif, .svs, etc.) files to be analyzed, as well as a CSV file containing the names of the channels in the image (if not specified in the image metadata). With this, the user can quickly iterate through a set of tasks crucial to the pre-processing and QC pipeline.

Once the experiment begins, the user will first select one or more of the available images provided in the data folder. Next, selecting from the list of available biomarkers will populate the histogram in the center of the screen. The histogram can have its parameters easily tweaked, such as setting ranges of the x axis, number of bins, log of the y axis, or density plot. Additionally, the user can choose to hide pixel values above or below a threshold. The subsample rate of pixels can be adjusted to balance the tradeoff between runtime and accuracy (for larger images, it may take several seconds to sample the pixels and plot them).

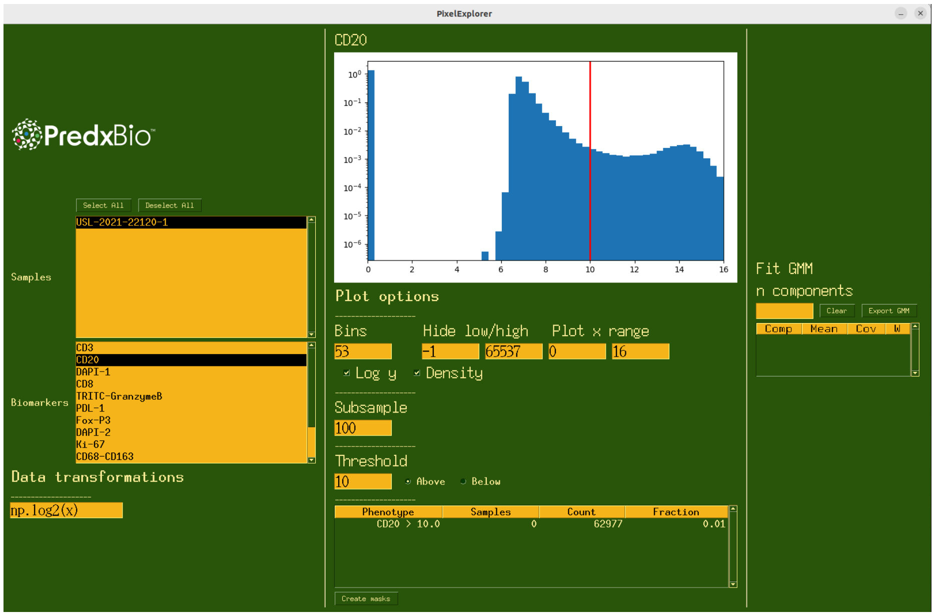
Further, any arbitrary transformation can be performed on the raw pixel values before plotting them. By default, a log2 transform is applied to transform the data from range [1, 2^16] to [0, 16]. However, any valid line of python code can be added to the 'Data transformations' tab to be applied to the data prior to plotting. This can be helpful to visualize the effect different transformations have on the underlying distribution.

Additionally, one can fit a Gaussian Mixture Model to the current distribution using the right panel by simply specifying the number of mixture components and pressing enter. This will fit a model to the data being plotted, meaning the model will be fit to the data after the transformation is applied and after any pixels above or below the desired thresholds are removed. This tool is very useful due to the fact that biomarker distributions in the log space can be effectively modeled by a mixture of gaussians [23], [24]. With this, one may derive a model for 'foreground' and 'background' signals of the images. With this knowledge, the user may want to generate a 'foreground' or 'background' mask using the Thresholding function available in the app. The user can type in a value, select whether to threshold above or below the value, and then export a binary mask which is 1 for pixels that pass the threshold and 0 elsewhere. This mask may then be used in downstream analysis, where they may be used to focus on parts of the image which may exhibit certain behavior (e.g. thresholding Ki67, a biomarker which measure cellular proliferation or reproduction, which may be associated with tumoral regions). While these masks may be derived automatically, it may be desirable to define one by thresholding manually.

With PixelExplorer, the user obtains a better understanding of the biomarker signal behavior, how it has responded to the various preprocessing steps in the backend pipeline, and can fit models and derive thresholds for further downstream analysis.

## 5. CellExplorer

To address the need of quickly analyzing tabular data of cell features, we developed CellExplorer, another GUI which allows for fast and flexible basic analysis of nuclear or cellular



Samples: USL-2021-22120-1

Biomarkers: CD3, CD20, DAPI-1, CD8, TRITC-GranzymeB, PDL-1, Fox-P3, DAPI-2, Ki-67, CD68-CD163

Data transformations: np.log2(x)

Plot options

Bins: 53 | Hide low/high: -1, 65537 | Plot x range: 0, 16

☑ Log y ☑ Density

Subsample: 100

Threshold: 10 ● Above ○ Below

| Phenotype | Samples | Count | Fraction |
|---|---|---|---|
| CD20 > 10.0 | 0 | 62977 | 0.01 |

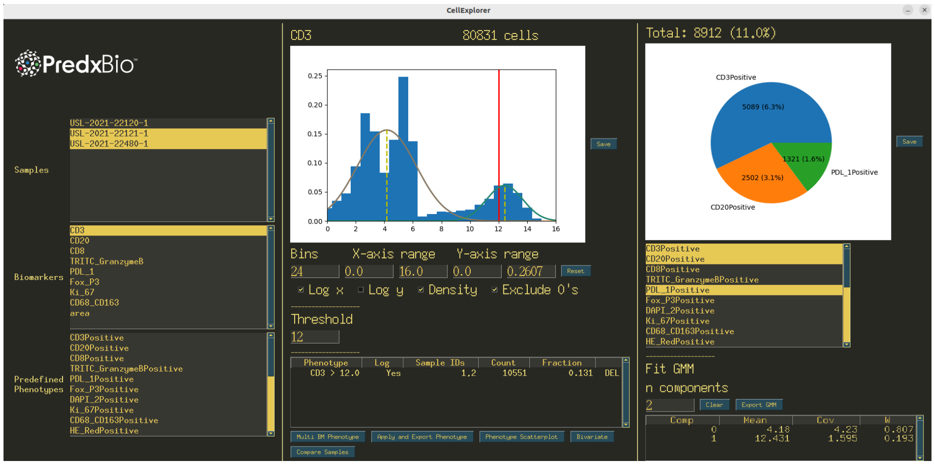
**Figure 2**. *The interface for PixelExplorer.*

feature tables. After the preprocessing and feature extraction pipelines are complete, what remains is one CSV file per image, each containing cell level features. These features may contain information about the object shape and morphology (H&E, IHC) or a measure of the biomarker signal (mIF, IMC). With these feature tables as input, several types of analysis are available using the CellExplorer GUI. CellExplorer requires as input a folder containing all of the CSV files to be analyzed. The CSV files contain rows of cellular features. Similar to PixelExplorer, when the app launches, the user will select one or more patient samples and a feature column to begin visualizing the distributions. The histogram visualization has the same controls as those found in PixelExplorer. Additionally, the user can define thresholds on the features and view counts or ratios of positive cells or nuclei. Thresholded phenotypes can then be exported as new columns in the CSV input files. Again, a Gaussian Mixture Model can be fit to the current distribution.

CellExplorer also contains some additional features more useful in the context of analyzing cellular data, as opposed to pixel data. These features are gated behind popup windows, accessed via buttons at the bottom of the screen (Figure 4). This is to allow the main panel of the GUI to be more streamlined and simple, and the more advanced analysis methods hidden in popups. This also allows for modular programming design, allowing different modules to be developed and deployed as new popup windows. Phenotypes may be a part of the input data (columns in the CSV with ‘Positive’ in the name), or may be generated as previously mentioned. These phenotypes can be selected and shown as a pie chart to view relative frequencies of the phenotypes. Phenotypes may be generated using multiple features.

## 6. SpaceExplorer

After cells have been assigned a phenotype label, it is advantageous to analyze the spatial composition of the cell types. This involves quantifying the degree with which the cells cluster together or repel each other relative to some background distribution. That is, we may discover that tumor cells are clustered tightly together with other tumor cells, or that immune cells may be surrounded by many tumor cells, but few other immune cells. These spatial relationships reveal the underlying mechanisms driving the tumor microenvironment [2]. However, biomarker panels may include a large number of biomarkers, and



Bins: 24 | X-axis range: 0.0 16.0 | Y-axis range: 0.0 0.2607

☑ Log x ☐ Log y ☑ Density ☑ Exclude 0's

Threshold: 12

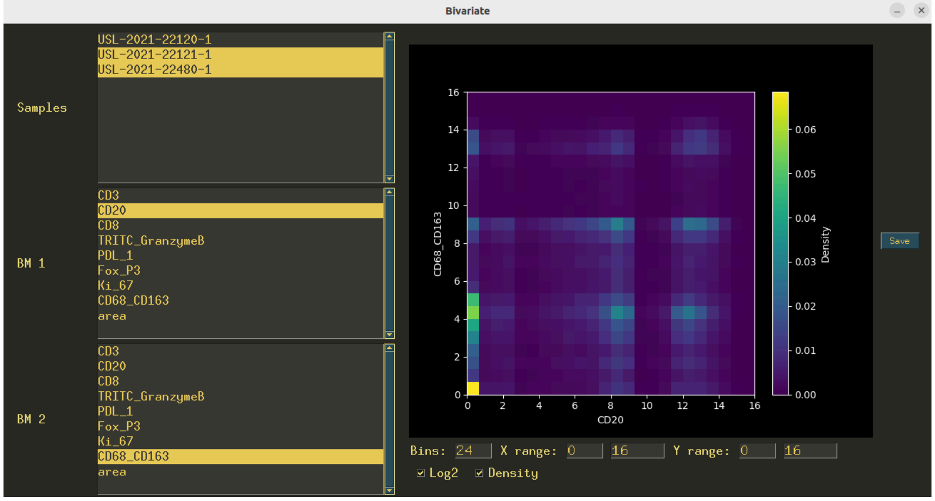
| Phenotype | Log | Sample IDs | Count | Fraction | |
|---|---|---|---|---|---|
| CD3 > 12.0 | Yes | 1,2 | 10551 | 0.131 | DEL |

Fit GMM

n components: 2

| Comp | Mean | Cov | W |
|---|---|---|---|
| 0 | 4.18 | 4.23 | 0.807 |
| 1 | 12.431 | 1.595 | 0.193 |

**Figure 3**. *The interface for CellExplorer.*



Bins: 24 | X range: 0 16 | Y range: 0 16

☑ Log2 ☑ Density

**Figure 4**. *An example of a popup window in CellExplorer. This interface allows for bivariate plots of the biomarker values.*

parsing the magnitude of statistics generated by pairwise analysis can be cumbersome. Thus, there is a need for a fast and interactive way of viewing and analyzing the various spatial statistics that can reveal insights into the tumor microenvironment.

To this end, we have created SpaceExplorer to enable a user to quickly generate a variety of spatial statistics on singular phenotypes or pairs of phenotypes, and to generate high quality figures. Specifically, SpaceExplorer functions similarly to CellExplorer, in that it reads as input a CSV file, this time containing binary assignments of phenotypes for each cell. That is, each row corresponds to a cell, and each column says whether that cell is positive or negative for that phenotype (IE tumor cell, T cell, etc.).

Once the data is read in, the user has a few options to explore. The user may select from the samples list one or more samples to compute scores for. Then, the user may select 1 or 2 cell types to perform spatial point-pattern analysis on. This analysis attempts to describe the clustering or dispersing behavior of events relative to a random point pattern, and has shown promise in other fields for describing spatial distributions of events [25], [26], [27].

Additional analysis methods include the pointwise mutual information [28], which analyzes the probability that two events co-occur (in this case, 2 cell types being spatially close together) relative to a background distribution, as well as the spatial neighbors enrichment analysis, which performs permutation tests and computes z-scores to assign a score for how often 2 events are co-occurring [29].

## 7. Conclusion

Here we have described Predx-Tools, our suite of easy to use and flexible GUI applications for interfacing with histopathological image data at various stages of processing. We developed these tools to fit the needs that arose from working with a large variety of histopathological datasets and understanding the variability that must be accounted for. These tools improve workflows and allow for a deeper understanding of the complex data. All tools were built with python and open-source python libraries.

### References

[1] S. Nam *et al.*, “Introduction to digital pathology and computer-aided pathology,” *Journal of Pathology and Translational Medicine*, vol. 54, no. 2, pp. 125–134, 2020, doi: 10.4132/jptm.2019.12.31.

[2] F. Rojas, S. Hernandez, R. Lazcano, C. Laberiano-Fernandez, and E. R. Parra, “Multiplex Immunofluorescence and the Digital Image Analysis Workflow for Evaluation of the Tumor Immune Environment in Translational Research,” *Frontiers in Oncology*, vol. 12, 2022, doi: 10.3389/fonc.2022.889886.

[3] C. Kaushal, S. Bhat, D. Koundal, and A. Singla, “Recent Trends in Computer Assisted Diagnosis (CAD) System for Breast Cancer Diagnosis Using Histopathological Images,” *IRBM*, vol. 40, no. 4, pp. 211–227, 2019, doi: 10.1016/j.irbm.2019.06.001.

[4] S. Gray and C. H. Ottensmeier, “Advancing Understanding of Non-Small Cell Lung Cancer with Multiplexed Antibody-Based Spatial Imaging Technologies,” *Cancers*, vol. 15, no. 19, p. 4797, 2023, doi: 10.3390/cancers15194797.

[5] V. Baxi, R. Edwards, M. Montalto, and S. Saha, “Digital pathology and artificial intelligence in translational medicine and clinical practice,” *Modern Pathology*, vol. 35, no. 1, pp. 23–32, 2022, doi: 10.1038/s41379-021-00919-2.

[6] C. M. Wilson *et al.*, “Challenges and Opportunities in the Statistical Analysis of Multiplex Immunofluorescence Data,” *Cancers*, vol. 13, no. 12, p. 3031, 2021, doi: 10.3390/cancers13123031.

[7] E. Mulholland and S. Leedham, “Redefining clinical practice through spatial profiling: a revolution in tissue analysis,” *The Annals of The Royal College of Surgeons of England*, vol. 106, no. 4, pp. 305–312, 2024, doi: 10.1308/rcsann.2023.0091.

[8] A. C. S. Bodén, J. Molin, S. Garvin, R. A. West, C. Lundström, and D. Treanor, “The humanintheloop: an evaluation of pathologists’ interaction with artificial intelligence in clinical practice,” *Histopathology*, vol. 79, no. 2, pp. 210–218, 2021, doi: 10.1111/his.14356.

[9] PySimpleGUI Team, “PySimpleGUI.” [Online]. Available: https://www.pysimplegui.com/

[10] J. D. Hunter, “Matplotlib: A 2D graphics environment,” *Computing in Science & Engineering*, vol. 9, no. 3, pp. 90–95, 2007, doi: https://doi.org/10.1109/MCSE.2007.55.

[11] C. R. Harris *et al.*, “Array programming with NumPy,” *Nature*, vol. 585, no. 7825, pp. 357–362, 2020, doi: https://doi.org/10.1038/s41586-020-2649-2.

[12] The Pandas Development Team, “pandas-dev/pandas: Pandas.” [Online]. Available: https://doi.org/10.5281/zenodo.3509134

[13] W. 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. doi: https://doi.org/10.25080/Majora-92bf1922-00a.

[14] F. Pedregosa *et al.*, “Scikit-learn: Machine Learning in Python,” *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[15] L. Buitinck *et al.*, “API design for machine learning software: experiences from the scikit-learn project,” in *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*, 2013, pp. 108–122.

[16] P. Virtanen *et al.*, “SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python,” *Nature Methods*, vol. 17, pp. 261–272, 2020, doi: https://doi.org/10.1038/s41592-019-0686-2.

[17] C. Gohlke, “tifffile.” [Online]. Available: https://pypi.org/project/tifffile/

[18] Zarr-Python Team, “Zarr-Python.” [Online]. Available: https://zarr.readthedocs.io/en/stable/

[19] W. S. R. &. K. W. E. Caroline A Schneider, 2012. doi: doi:10.1038/nmeth.2089.

[20] Peter Bankhead, 2017. doi: https://doi.org/10.1038/s41598-017-17204-5.

[21] Stirling, 2021. doi: https://doi.org/10.1186/s12859-021-04344-9.

[22] Berg, 2019. doi: https://doi.org/10.1038/s41592-019-0582-9.

[23] J.-R. Lin *et al.*, "High-plex immunofluorescence imaging and traditional histology of the same tissue section for discovering image-based biomarkers," *Nature Cancer*, vol. 4, no. 7, pp. 1036–1052, 2023, doi: 10.1038/s43018-023-00576-1.

[24] E. T. McKinley *et al.*, "Optimized multiplex immunofluorescence single-cell analysis reveals tuft cell heterogeneity," *JCI Insight*, vol. 2, no. 11, 2017, doi: 10.1172/jci.insight.93487.

[25] M. Ben-Said, "Spatial point-pattern analysis as a powerful tool in identifying pattern-process relationships in plant ecology: an updated review," *Ecological Processes*, vol. 10, no. 1, 2021, doi: 10.1186/s13717-021-00314-4.

[26] J. Franklin, "Spatial Point Pattern Analysis of Plants," in *Perspectives on Spatial Data Analysis*, Springer Berlin Heidelberg, 2008, pp. 113–123. doi: 10.1007/978-3-642-01976-0_9.

[27] B. D. Ripley, "Modelling Spatial Patterns," *Journal of the Royal Statistical Society Series B: Statistical Methodology*, vol. 39, no. 2, pp. 172–192, 1977, doi: 10.1111/j.2517-6161.1977.tb01615.x.

[28] K. W. Church and P. Hanks, "Word association norms, mutual information, and lexicography," in *Proceedings of the 27th annual meeting on Association for Computational Linguistics -*, 1989, pp. 76–83. doi: 10.3115/981623.981633.

[29] C. M. Langseth *et al.*, "Comprehensive in situ mapping of human cortical transcriptomic cell types," *Communications Biology*, vol. 4, no. 1, 2021, doi: 10.1038/s42003-021-02517-z.

OPEN ACCESS | https://doi.org/10.25080/WPXM6451

# Improving Code Quality with Array and DataFrame Type Hints

Christopher Ariza[1]

[1] Research Affiliates

SciPy 2024

July 8 - July 14, 2024

Proceedings of the 23rd
Python in Science Conference
ISSN: 2575-9752

Published Jul 10, 2024

Correspondence to
Christopher Ariza
ariza@flexatone.com

## Abstract

This article demonstrates practical approaches to fully type-hinting generic NumPy arrays and StaticFrame DataFrames, and shows how the same annotations can improve code quality with both static analysis and runtime validation.

**Keywords** typing, static analysis, runtime validation, variadic generics

As tools for Python type annotations (or hints) have evolved, more complex data structures can be typed, improving maintainability and static analysis. Arrays and DataFrames, as complex containers, have only recently supported complete type annotations in Python. NumPy [1] 1.22 introduced generic specification of arrays and dtypes. Building on NumPy's foundation, StaticFrame [2] 2.0 introduced complete type specification of DataFrames, employing NumPy primitives and variadic generics. This article demonstrates practical approaches to fully type-hinting arrays and DataFrames, and shows how the same annotations can improve code quality with both static analysis and runtime validation.

## 1. Type Hints Improve Code Quality

Type hints [3] improve code quality in a number of ways. Instead of using variable names or comments to communicate types, Python-object-based type annotations provide maintainable and expressive tools for type specification. These type annotations can be tested with type checkers such as `mypy` [4] or `pyright` [5], quickly discovering potential bugs without executing code.

The same annotations can be used for runtime validation. While reliance on duck-typing over runtime validation is common in Python, runtime validation is more often needed with complex data structures such as arrays and DataFrames. For example, an interface expecting a DataFrame argument, if given a Series, might not need explicit validation as usage of the wrong type will likely raise. However, an interface expecting a 2D array of floats, if given an array of Booleans, might benefit from validation as usage of the wrong type may not raise.

Many important typing utilities are only available with the most-recent versions of Python. Fortunately, the `typing-extensions` [6] package back-ports standard library utilities for older versions of Python. A related challenge is that type checkers can take time to implement full support for new features: many of the examples shown here require `mypy` 1.9.0, released just a few months ago.

## 2. Elemental Type Annotations

Without type annotations, a Python function signature gives no indication of the expected types. For example, the function below might take and return any types:

```
def process0(v, q): ... # no type information
```

By adding type annotations, the signature informs readers of the expected types. With modern Python, user-defined and built-in classes can be used to specify types, with additional resources (such as `Any`, `Iterator`, `cast()`, and `Annotated`) found in the standard library `typing` module. For example, the interface below improves the one above by making expected types explicit:

```
def process0(v: int, q: bool) -> list[float]: ...
```

When used with a type checker like `mypy`, code that violates the specifications of the type annotations will raise an error during static analysis (shown as comments, below). For example, providing an integer when a Boolean is required is an error:

```
x = process0(v=5, q=20)
# tp.py: error: Argument "q" to "process0"
# has incompatible type "int"; expected "bool"  [arg-type]
```

Static analysis can only validate statically defined types. The full range of runtime inputs and outputs is often more diverse, suggesting some form of runtime validation. The best of both worlds is possible by reusing type annotations for runtime validation. While there are libraries that do this (e.g., `typeguard` and `beartype`), StaticFrame offers `CallGuard`, a tool specialized for comprehensive array and DataFrame type-annotation validation.

A Python decorator is ideal for leveraging annotations for runtime validation. `CallGuard` offers two decorators: `@CallGuard.check`, which raises an informative `Exception` on error, or `@CallGuard.warn`, which issues a warning.

Further extending the `process0` function above with `@CallGuard.check`, the same type annotations can be used to raise an `Exception` (shown again as comments) when runtime objects violate the requirements of the type annotations:

```
import static_frame as sf

@sf.CallGuard.check
def process0(v: int, q: bool) -> list[float]:
    return [x * (0.5 if q else 0.25) for x in range(v)]

z = process0(v=5, q=20)
# static_frame.core.type_clinic.ClinicError:
# In args of (v: int, q: bool) -> list[float]
# └── Expected bool, provided int invalid
```

While type annotations must be valid Python, they are irrelevant at runtime and can be wrong: it is possible to have correctly verified types that do not reflect runtime reality. As shown above, reusing type annotations for runtime checks ensures annotations are valid.

## 3. Array Type Annotations

Python classes that permit component type specification are “generic”. Component types are specified with positional “type variables”. A list of integers, for example, is annotated with `list[int]`; a dictionary of floats keyed by tuples of integers and strings is annotated `dict[tuple[int, str], float]`.

With NumPy 1.20, `ndarray` and `dtype` become generic. The generic `ndarray` requires two arguments, a shape and a `dtype`. As the usage of the first argument is still under development, `Any` is commonly used. The second argument, `dtype`, is itself a generic that requires a type variable for a NumPy type such as `np.int64`. NumPy also offers more general generic types such as `np.integer[Any]`.

For example, an array of Booleans is annotated `np.ndarray[Any, np.dtype[np.bool_]]`; an array of any type of integer is annotated `np.ndarray[Any, np.dtype[np.integer[Any]]]`.

As generic annotations with component type specifications can become verbose, it is practical to store them as type aliases (here prefixed with "T"). The following function specifies such aliases and then uses them in a function.

```
from typing import Any
import numpy as np

TNDArrayInt8 = np.ndarray[Any, np.dtype[np.int8]]
TNDArrayBool = np.ndarray[Any, np.dtype[np.bool_]]
TNDArrayFloat64 = np.ndarray[Any, np.dtype[np.float64]]

def process1(
        v: TNDArrayInt8,
        q: TNDArrayBool,
        ) -> TNDArrayFloat64:
    s: TNDArrayFloat64 = np.where(q, 0.5, 0.25)
    return v * s
```

As before, when used with `mypy`, code that violates the type annotations will raise an error during static analysis. For example, providing an integer when a Boolean is required is an error:

```
v1: TNDArrayInt8 = np.arange(20, dtype=np.int8)
x = process1(v1, v1)
# tp.py: error: Argument 2 to "process1" has incompatible type
# "ndarray[Any, dtype[floating[_64Bit]]]"; expected "ndarray[Any, dtype[bool_]]"  [arg-type]
```

The interface requires 8-bit signed integers (`np.int8`); attempting to use a different sized integer is also an error:

```
TNDArrayInt64 = np.ndarray[Any, np.dtype[np.int64]]
v2: TNDArrayInt64 = np.arange(20, dtype=np.int64)
q: TNDArrayBool = np.arange(20) % 3 == 0
x = process1(v2, q)
# tp.py: error: Argument 1 to "process1" has incompatible type
# "ndarray[Any, dtype[signedinteger[_64Bit]]]"; expected "ndarray[Any,
dtype[signedinteger[_8Bit]]]"  [arg-type]
```

While some interfaces might benefit from such narrow numeric type specifications, broader specification is possible with NumPy's generic types such as `np.integer[Any]`, `np.signedinteger[Any]`, `np.float[Any]`, etc. For example, we can define a new function that accepts any size signed integer. Static analysis now passes with both `TNDArrayInt8` and `TNDArrayInt64` arrays.

```
TNDArrayIntAny = np.ndarray[Any, np.dtype[np.signedinteger[Any]]]
def process2(
        v: TNDArrayIntAny, # a more flexible interface
        q: TNDArrayBool,
        ) -> TNDArrayFloat64:
    s: TNDArrayFloat64 = np.where(q, 0.5, 0.25)
    return v * s

x = process2(v1, q) # no mypy error
x = process2(v2, q) # no mypy error
```

Just as shown above with elements, generically specified NumPy arrays can be validated at runtime if decorated with `CallGuard.check`:

```
@sf.CallGuard.check
def process3(v: TNDArrayIntAny, q: TNDArrayBool) -> TNDArrayFloat64:
    s: TNDArrayFloat64 = np.where(q, 0.5, 0.25)
    return v * s

x = process3(v1, q) # no error, same as mypy
x = process3(v2, q) # no error, same as mypy
v3: TNDArrayFloat64 = np.arange(20, dtype=np.float64) * 0.5
x = process3(v3, q) # error
# static_frame.core.type_clinic.ClinicError:
# In args of (v: ndarray[Any, dtype[signedinteger[Any]]],
# q: ndarray[Any, dtype[bool_]]) -> ndarray[Any, dtype[float64]]
# └── ndarray[Any, dtype[signedinteger[Any]]]
#     └── dtype[signedinteger[Any]]
#         └── Expected signedinteger, provided float64 invalid
```

StaticFrame provides utilities to extend runtime validation beyond type checking. Using the `typing` module's `Annotated` class [7], we can extend the type specification with one or more StaticFrame `Require` objects. For example, to validate that an array has a 1D shape of `(24,)`, we can replace `TNDArrayIntAny` with `Annotated[TNDArrayIntAny, sf.Require.Shape(24)]`. To validate that a float array has no NaNs, we can replace `TNDArrayFloat64` with `Annotated[TNDArrayFloat64, sf.Require.Apply(lambda a: ~a.insna().any())]`.

Implementing a new function, we can require that all input and output arrays have the shape `(24,)`. Calling this function with the previously created arrays raises an error:

```
from typing import Annotated

@sf.CallGuard.check
def process4(
        v: Annotated[TNDArrayIntAny, sf.Require.Shape(24)],
        q: Annotated[TNDArrayBool, sf.Require.Shape(24)],
        ) -> Annotated[TNDArrayFloat64, sf.Require.Shape(24)]:
    s: TNDArrayFloat64 = np.where(q, 0.5, 0.25)
    return v * s

x = process4(v1, q) # types pass, but Require.Shape fails
# static_frame.core.type_clinic.ClinicError:
# In args of (v: Annotated[ndarray[Any, dtype[int8]], Shape((24,))], q: Annotated[ndarray[Any,
dtype[bool_]], Shape((24,))]) -> Annotated[ndarray[Any, dtype[float64]], Shape((24,))]
# └── Annotated[ndarray[Any, dtype[int8]], Shape((24,))]
#     └── Shape((24,))
#         └── Expected shape ((24,)), provided shape (20,)
```

## 4. DataFrame Type Annotations

Just like a dictionary, a DataFrame is a complex data structure composed of many component types: the index labels, column labels, and the column values are all distinct types.

A challenge of generically specifying a DataFrame is that a DataFrame has a variable number of columns, where each column might be a different type. The Python `TypeVarTuple` variadic generic specifier [8], first released in Python 3.11, permits defining a variable number of column type variables.

With StaticFrame 2.0, `Frame`, `Series`, `Index` and related containers become generic. Support for variable column type definitions is provided by `TypeVarTuple`, back-ported with the implementation in `typing-extensions` for compatibility down to Python 3.9.

A generic `Frame` requires two or more type variables: the type of the index, the type of the columns, and zero or more specifications of columnar value types specified with NumPy types. A generic `Series` requires two type variables: the type of the index and a NumPy type for the values. The `Index` is itself generic, also requiring a NumPy type as a type variable.

With generic specification, a `Series` of floats, indexed by dates, can be annotated with `sf.Series[sf.IndexDate, np.float64]`. A `Frame` with dates as index labels, strings as column labels, and column values of integers and floats can be annotated with `sf.Frame[sf.IndexDate, sf.Index[np.str_], np.int64, np.float64]`.

Given a complex `Frame`, deriving the annotation might be difficult. StaticFrame offers the `via_type_clinic` interface to provide a complete generic specification for any component at runtime:

```
>>> v4 = sf.Frame.from_fields([range(5), np.arange(3, 8) * 0.5],
columns=('a', 'b'), index=sf.IndexDate.from_date_range('2021-12-30', '2022-01-03'))
>>> v4
<Frame>
<Index>         a       b         <<U1>
<IndexDate>
2021-12-30      0       1.5
2021-12-31      1       2.0
2022-01-01      2       2.5
2022-01-02      3       3.0
2022-01-03      4       3.5
<datetime64[D]> <int64> <float64>

# get a string representation of the annotation
>>> v4.via_type_clinic
Frame[IndexDate, Index[str_], int64, float64]
```

As shown with arrays, storing annotations as type aliases permits reuse and more concise function signatures. Below, a new function is defined with generic `Frame` and `Series` arguments fully annotated. A `cast` is required as not all operations can statically resolve their return type.

```
TFrameDateInts = sf.Frame[sf.IndexDate, sf.Index[np.str_], np.int64, np.int64]
TSeriesYMBool = sf.Series[sf.IndexYearMonth, np.bool_]
TSeriesDFloat = sf.Series[sf.IndexDate, np.float64]

def process5(v: TFrameDateInts, q: TSeriesYMBool) -> TSeriesDFloat:
    t = v.index.iter_label().apply(lambda l: q[l.astype('datetime64[M]')]) # type: ignore
    s = np.where(t, 0.5, 0.25)
    return cast(TSeriesDFloat, (v.via_T * s).mean(axis=1))
```

These more complex annotated interfaces can also be validated with `mypy`. Below, a `Frame` without the expected column value types is passed, causing `mypy` to error (shown as comments, below).

```
TFrameDateIntFloat = sf.Frame[sf.IndexDate, sf.Index[np.str_], np.int64, np.float64]
v5: TFrameDateIntFloat = sf.Frame.from_fields([range(5), np.arange(3, 8) * 0.5],
columns=('a', 'b'), index=sf.IndexDate.from_date_range('2021-12-30', '2022-01-03'))

q: TSeriesYMBool = sf.Series([True, False],
index=sf.IndexYearMonth.from_date_range('2021-12', '2022-01'))

x = process5(v5, q)
# tp.py: error: Argument 1 to "process5" has incompatible type
# "Frame[IndexDate, Index[str_], signedinteger[_64Bit], floating[_64Bit]]"; expected
# "Frame[IndexDate, Index[str_], signedinteger[_64Bit], signedinteger[_64Bit]]"  [arg-type]
```

To use the same type hints for runtime validation, the `sf.CallGuard.check` decorator can be applied. Below, a `Frame` of three integer columns is provided where a `Frame` of two columns is expected.

```
# a Frame of three columns of integers
TFrameDateIntIntInt = sf.Frame[sf.IndexDate, sf.Index[np.str_], np.int64, np.int64, np.int64]
v6: TFrameDateIntIntInt = sf.Frame.from_fields([range(5), range(3, 8), range(1, 6)],
columns=('a', 'b', 'c'), index=sf.IndexDate.from_date_range('2021-12-30', '2022-01-03'))

x = process5(v6, q)
# static_frame.core.type_clinic.ClinicError:
# In args of (v: Frame[IndexDate, Index[str_], signedinteger[_64Bit], signedinteger[_64Bit]],
# q: Series[IndexYearMonth, bool_]) -> Series[IndexDate, float64]
# └── Frame[IndexDate, Index[str_], signedinteger[_64Bit], signedinteger[_64Bit]]
#     └── Expected Frame has 2 dtype, provided Frame has 3 dtype
```

It might not be practical to annotate every column of every `Frame`: it is common for interfaces to work with `Frame` of variable column sizes. `TypeVarTuple` supports this through the usage of unpack operator `*tuple[]` expressions (introduced in Python 3.11, back-ported with the `Unpack` annotation). For example, the function above could be defined to take any number of integer columns with that annotation `Frame[IndexDate, Index[np.str_], *tuple[np.int64, ...]]`, where `*tuple[np.int64, ...]]` means zero or more integer columns.

The same implementation can be annotated with a far more general specification of columnar types. Below, the column values are annotated with `np.number[Any]` (permitting any type of numeric NumPy type) and a `*tuple[]` expression (permitting any number of columns): `*tuple[np.number[Any], ...]`. Now neither `mypy` nor `CallGuard` errors with either previously created `Frame`.

```
TFrameDateNums = sf.Frame[sf.IndexDate, sf.Index[np.str_], *tuple[np.number[Any], ...]]

@sf.CallGuard.check
def process6(v: TFrameDateNums, q: TSeriesYMBool) -> TSeriesDFloat:
    t = v.index.iter_label().apply(lambda l: q[l.astype('datetime64[M]')]) # type: ignore
    s = np.where(t, 0.5, 0.25)
    return tp.cast(TSeriesDFloat, (v.via_T * s).mean(axis=1))

x = process6(v5, q) # a Frame with integer, float columns passes
x = process6(v6, q) # a Frame with three integer columns passes
```

As with NumPy arrays, `Frame` annotations can wrap `Require` specifications in `Annotated` generics, permitting the definition of additional run-time validations.

## 5. Type Annotations with Other Libraries

While StaticFrame might be the first DataFrame library to offer complete generic specification and a unified solution for both static type analysis and run-time type validation, other array and DataFrame libraries offer related utilities.

Neither the Tensor class in PyTorch (2.4.0), nor the Tensor class in TensorFlow (2.17.0) support generic type or shape specification. While both libraries offer a TensorSpec object that can be used to perform run-time type and shape validation, static type checking with tools like mypy is not supported.

As of Pandas 2.2.2, neither the Pandas Series nor DataFrame support generic type specifications. A number of third-party packages have offered partial solutions. The pandas-stubs library, for example, provides type annotations for the Pandas API, but does not make the Series or DataFrame classes generic. The pandera library [9] permits defining DataFrameSchema classes that can be used for run-time validation of Pandas DataFrames. For static-analysis with mypy, pandera offers alternative DataFrame and Series subclasses that permit generic specification with the same DataFrameSchema classes. This approach does not permit the expressive opportunities of using generic NumPy types or the unpack operator for supplying variadic generic expressions.

## 6. Conclusion

Python type annotations can make static analysis of types a valuable check of code quality, discovering errors before code is even executed. Up until recently, an interface might take an array or a DataFrame, but no specification of the types contained in those containers was possible. Now, complete specification of component types is possible in NumPy and StaticFrame, permitting more powerful static analysis of types.

Providing correct type annotations is an investment. Reusing those annotations for runtime checks provides the best of both worlds. StaticFrame's CallGuard runtime type checker is specialized to correctly evaluate fully specified generic NumPy types, as well as all generic StaticFrame containers.

## References

[1] C. R. Harris *et al.*, “Array programming with NumPy,” *Nature*, vol. 585, no. 7825, pp. 357–362, 2020, doi: https://doi.org/10.1038/s41586-020-2649-2.

[2] StaticFrame Development Team, “StaticFrame.” [Online]. Available: https://github.com/static-frame/static-frame

[3] G. van Rossum, J. Lehtosalo, and Ł. Langa, “PEP 484 - Type Hints,” 2014. [Online]. Available: https://www.python.org/dev/peps/pep-0484/

[4] mypy Development Team, “mypy.” [Online]. Available: https://mypy.readthedocs.io/en/stable/

[5] pyright Development Team, “pyright.” [Online]. Available: https://github.com/microsoft/pyright/

[6] typing_extensions Development Team, “typing-extensions.” [Online]. Available: https://typing-extensions.readthedocs.io/

[7] R. Gonzalez, P. House, I. Levkivskyi, L. Roach, and G. van Rossum, “PEP 593 - Flexible Function and Variable Annotations,” 2019. [Online]. Available: https://www.python.org/dev/peps/pep-0593/

[8] M. Mendoza, M. Rahtz, P. K. Srinivasan, and V. Siles, “PEP 646 - Variadic Generics,” 2020. [Online]. Available: https://www.python.org/dev/peps/pep-0646/

[9] N. Bantilan, “pandera: Statistical Data Validation of Pandas Dataframes,” in *Proceedings of the 19th Python in Science Conference*, M. Agarwal, C. Calloway, D. Niederhut, and D. Shupe, Eds., 2020, pp. 116–124. doi: https://doi.org/10.25080/Majora-342d178e-010.

Open Access | https://doi.org/10.25080/NKVC9349

# Continuous Tools for Scientific Publishing

## Using MyST Markdown and Curvenote to encourage continuous science practices

**Rowan Cockett**[1,2], **Steve Purves**[1,2], **Franklin Koch**[1,2], and **Mike Morrison**[1]

[1]Curvenote Inc., [2]Project Jupyter

*SciPy 2024*

*July 8 - July 14, 2024*

Proceedings of the 23rd
Python in Science Conference
ISSN: 2575-9752

**Published** Jul 10, 2024

**Correspondence to**
Rowan Cockett
rowan@curvenote.com

### Abstract

Advances in technology for data workflows have increased the speed and scope of scientific discovery, however, scientific dialogue still uses outdated technology for communicating and sharing knowledge. The widespread reliance on static PDF formats for research papers starkly contrasts with the complex, data-driven and increasingly computational nature of modern science. This gap, which is especially evident in the computational sciences, impedes the speed of research dissemination, reuse, and uptake. We require new mediums to compose ideas and ways to share research findings iteratively, as early as possible and connected *directly* to software and data. In this paper we discuss two tools for scientific authoring and publishing, MyST Markdown and Curvenote, and illustrate examples of improving metadata, reimagining the reading experience, including computational content, and transforming publishing practices for individuals and societies through automation and continuous practices. We focus on the unique aspects of the tools, which enable computational and interactive content, publishing and sharing continuously through automated checking and typesetting, and provide case studies from individuals to societies who have adopted these tools.

**Keywords** scientific communication, publishing, open science

## 1. Introduction and Motivation

In the face of mounting global challenges such as climate change, pandemics, and water security, the imperative for rapid, effective scientific discovery and dissemination has never been more acute. The pace at which these problems evolve and impact societies worldwide demands an equally dynamic and innovative approach to how scientific research is published. Despite significant advancements in technologies that enhance data collection, analysis, and workflow efficiency, the mechanisms through which scientific knowledge is shared have remained largely unchanged for decades [1][1]. The widespread reliance on static PDF formats for research papers starkly contrasts with the complex, data-driven and increasingly computational nature of modern science, creating bottlenecks in knowledge dissemination and uptake.

[1] The Future of Research Communication and eScholarship (https://force11.org) conference, released their manifesto in 2012 P. E. Bourne *et al.* [1] and much of that original writing still pertains to today, with the PDF being the main format of science communication.

> A dispassionate observer, perhaps visiting from another planet, would surely be dumbfounded by how, in an age of multimedia, smartphones, 3D television and 24/7 social network connectivity, scholars and researchers continue to communicate their thoughts and research results primarily by means of the selective distribution of ink on paper, or at best via electronic facsimiles of the same.
>
> — P. E. Bourne *et al.* [1]

This paper documents some of the design decisions made to address challenges in science communication and publishing in two tools: (1) MyST Markdown (Markedly Structured Text, https://mystmd.org), a community-run open-source Jupyter sub-project[2], which is a text-based authoring framework that integrates computational content (e.g. Jupyter Notebooks); and (2) Curvenote (https://curvenote.com), which is a set of open-source utilities, command-line tools, actions and services[3] aimed to improve scientific publishing by journals, societies, lab-groups, and individuals. In this article we provide background, motivation and perspective for our efforts in developing new open-source tools for science communication, with examples ranging from individual authors to journal administrators. Though we present an overview of MyST Markdown, it should be emphasized that MyST Markdown is a community-run project and the authors of this article do not speak for all project participants; the community has varied goals for the project (including API documentation, community guidelines, educational tutorials). Our focus in this article is to give our perspectives on scientific writing and publishing and how it intersects with these open-community projects in addition to the open-source efforts that Curvenote is undertaking around scientific publishing.

In developing these integrated tools and workflows, our goal is to lower the barriers to continuously releasing and iterating on scientific ideas in the open and address the related challenges of *authoring* and *publishing* in the context of computational, open-science documents. Introducing authoring tools that can understand and express structured, interactive, and computational content has the potential to fundamentally change the way scientific writing is checked, shared, and published — enabling faster iterations and direct ties to reproducible, interactive content.

### *1.1. Authoring Structured Content*

There are currently many challenges for individuals or groups to author research information that can be shared in a structured and rigorous way. By this we mean the things that *structurally* set a scientific article apart from, for example, a blog post: structured content, cross-references, valid citations with persistent identifiers (PIDs), and standardized metadata for licensing, funding information, authors, and affiliations. These structured content and metadata, as well as the standards behind them, are what define the “scientific record” and enable archiving, discoverability, accessibility, interoperability and the ability to reuse or cite content [2]. One metric for measuring the difficulty of satisfying these scientific standards is to look at the direct costs that are spent on transforming author submissions (e.g. a PDF or a Word Document) into something that conforms to these standards and is ultimately archived. In scientific publishing, about 15% of Article Processing Charges (APCs) go to direct publication costs[4] [3]. When applied to the global publishing industry[5], this suggests that approximately USD$2 billion dollars is spent on transforming author submissions (e.g. a word-document, LaTeX, or a PDF) into a copyedited, well-formatted, typeset document that can be archived with appropriate metadata [4]. This estimate does not include the approximately USD$230 million spent on reformatting articles by scientists *before* publica-

[2] MyST Markdown became a Jupyter Project on June 28, 2024 jupyter/enhancement-proposals#123, and was previously hosted by Executable Books (https://executablebooks.org).

[3] Curvenote is a company that provides many different tools for authoring and publishing content, including a collaborative WYSIWYG online editor that can export to MyST Markdown. In this article we discuss Curvenote’s open-source tools, specifically (a) a command-line interface (https://github.com/curvenote/curvenote); and (b) GitHub actions for building and checking content (https://github.com/curvenote/actions). We also highlight ideas from working with Curvenote’s partners when they pertain to improving scientific publishing.

[4] Direct publication costs include: checking of manuscript, copyediting, typesetting, formatting figures/graphs/tables, XML and metadata preparation, and handling corrections [3].

[5] Global revenue in scientific publishing is around USD$19 billion, with over 50% of the market controlled by Elsevier, Wiley, Taylor & Francis, Springer Nature and SAGE [4].

tion [5]. Many of these processes are hidden from authors[6] as well as actionable access to many of the benefits of structured data beyond citation graphs.

One goal of the MyST Markdown project is to *dramatically* reduce these direct-publication costs[7] and provide unfettered access to structured data as an output of authoring. The availability of this structured data directly enables exported content in a variety of formats including HTML, PDF and JATS-XML (a NISO standard for archiving scientific articles). In this article, we will demonstrate that having structured data throughout authoring can lead to a number of novel reading and authoring experiences Example 1, connect to interactivity and computation Example 2, and can provide new opportunities for reuse and quality checks when publishing Example 3. Furthermore, these transformation processes can be run *continuously*, opening the possibilities for faster feedback Section 1.3, iterative drafts, small tweaks and versioned improvements that otherwise would not be worth the time and cost.

### *1.2. Computational Articles*

A compounding challenge to scientific publishing that we are exploring through MyST Markdown and Curvenote is how to deeply integrate computational workflows and content into science communication to promote interactive explorations and reproducible practices. There are a host of challenges from user-interface design, to maintenance, to archiving computational content. Many other tools have worked on aspects of integrating computation into scientific articles, notably R-Markdown [7] and it's successor Quarto (https://quarto.org); both of these projects have similar aims to MyST Markdown. From a user-experience goals perspective, we are interested in questions such as:

- how to make a change in a notebook figure and have that immediately show up in a document;
- how to ensure computed values are inserted directly from source, rather than through copy-and-paste;
- how to expose interactivity and exploration that a researcher often has when analyzing a data-set;
- how to provide and launch archived interactive computing environments.

These questions require authoring tools to be able to execute content (e.g. using MyBinder; [8]), to integrate and display computational/interactive outputs directly in reading experiences, as well as scientific publishing systems that can understand and archive computational content (e.g. Docker containers). This deep integration can open up possibilities of embedding interactive visualizations and computational notebooks directly into scientific documents Example 2, transforming articles from static texts into rich, interactive, reproducible narratives. In 2023, the authors helped to lead several working groups related to these challenges as part of *Notebooks Now!*, a Sloan Foundation funded project led by the American Geophysical Union. Those working groups found that integrating computational documents, via Jupyter Notebooks, into scholarly publishing system requires a re-imagination of the publishing processes (from submission to peer-review to production to reading)

[6] Much of the production publication processes are hidden from scientific authors, with typesetting focused on cross-references, linking citations, ensuring citations have appropriate IDs (e.g. DOIs) as well as conversion to JATS XML (a NISO standard for archiving scientific articles), metadata preparation to CrossRef, and archiving services like LOCKSS (https://lockss.org) and CLOCKSS (https://clockss.org). Additionally, the many proprietary services and tools to create both online and PDF outputs of the authors work that are nicely typeset for reading on the web or online.

[7] The cost of transforming author submissions to produce structured content and metadata should approach zero, at least for a subset of users. For example, *technical* users who can use open-source command-line tools like MyST Markdown and GitHub. This has been shown to be the case for the Journal of Open Source Software (JOSS), for example, which advertizes a very low direct-publication cost of $2.71 per article [6].

and that many existing processes and platforms are ill-equipped to handle computational articles [9]. The "executable research articles" project out of eLife [10] has similar aims to *Notebooks Now!*, with some differences in how notebooks and articles are separated which we will discuss in Section 2.3.

The ability to deeply link computational content into how we communicate science can improve reproducible practices, and surface more interlinked content about methods and algorithms in use. If used to their full extent, these can also fully integrate live computational environments into scientific articles, which provides many exciting possibilities for interrogating and extending scientific data and methods [11].

### *1.3. Continuous Science Practices*

The manual effort involved in article production Section 1.1 coupled with the inability to integrate computational work Section 1.2 negatively impacts the number of iterations/versions and the immediacy of feedback to authors[8]. In other disciplines, such as software development, these metrics of iteration and rapid feedback are often highly encouraged, measured and constantly improved [12], [13], [14]. For example, software organizations often measure and improve: the release cadence of a software product (e.g. continuous delivery); how confident you are in that release (e.g. based on continuous integration tests); how you get early feedback and confidence from linters and tests (e.g. the speed of your unit tests and integrated linters into development environments); and how fast you can obtain feedback from real usage and users on in-progress work (e.g. observability, analytics, customer interviews, design prototypes). Continuous delivery practices of software development are also extremely well studied, with large-scale surveys of organizational performance, design, robustness, and speed (see L. Leite, C. Rocha, F. Kon, D. Milojicic, and P. Meirelles [13] and references within). One industry survey based on 36,000 professionals worldwide grouped and compared respondents based on software delivery performance [15]. The highest performing teams were **46x faster** to release to production than the lowest performing teams (i.e. on-demand and multiple times per day vs monthly or bi-annually) and had **7x fewer errors** (due in part to better continuous deployment and testing infrastructure as well as smaller changesets)[9] [15]. Similarly, R. Blinde [18] found in a survey of 123 professionals that "deployment frequency" correlates the strongest with all other organizational performance metrics. The study concluded that "practices that improve lead time" (automated deployments, continuous testing and version control) have a positive impact on "both software delivery performance and organizational performance" [18]. In a continuous deployment transformation over a year, M. Callanan and A. Spillane [19] saw a 20x reduction in manual effort releasing software and a 7x speed up in releases to production.

The analogy between scientific publishing and software releases is imperfect and non-prescriptive (i.e. scientific research is very different than developing a product). However, the analogy is illustrative in areas where there is a focus on iterations, smaller changesets and

[8] There are two types of feedback that we mean: (1) technical feedback as you are authoring, for example, "is this formatted correctly?" or "is this DOI correct?"; and (2) more substantial feedback from reviewers and readers who can only give you feedback when you have published. In the current system, technical feedback of an article-proof can take weeks and should be measured in milliseconds. Improving the immediacy of feedback from readers and peer-reviewers is a harder problem that involves how our existing sociotechnical system incentivizes article publishing rather than research communication and sharing findings as early and as often as possible.

[9] In addition to increasing speed and robustness, continuous delivery practices also demonstrated that the high-performing teams spent 20% more time on new work, had 5-20% less manual work, and were 1.8x more likely to recommend their team as a great place to work [15]. These numbers are intriguing when contrasted to researchers, where (a) scientists already work 53.96 hours a week on average, and only about 36% of their time is actually spent on research (8% on grants, 32% on teaching, and 24% on service) [16] and (b) graduate students are six times more likely to experience depression than the general population [17].

releasing in-progress work[10] as soon as possible to get feedback from peers (i.e. scientific peer-review) or users (in the case of software products). The speed of scientific progress depends *in part* on the speed of iteration and feedback. The time it takes for the peer-review process is over three months, in high profile journals like Nature that time has almost doubled over the past decade [21]. Rejections are anywhere from 50-90% [3] with valuable reviews and expertise coming months or even years after the work is completed[11]. There are wide spread efforts in scientific publishing that focus on sharing smaller components of research (e.g. FigShare [23], Octopus [24], MicroPublications [25], NanoPublications [26], Protocols [27], PreRegistrations [28]), automated tools in the publication process [6], as well as sharing research sooner in the life cycle especially through preprints [29], "Preprints in Progress" [22] and getting feedback sooner from a more diverse community [30].

We refer to these related concepts as "*continuous science*", adopting language and concepts from "continuous integration and deployment". The mechanisms to support continuous processes are through automation, rapid feedback on errors, and focusing on small, rapid changesets to accelerate feedback from peers. This gives us a technical lens to assess, for example:

- How long does it take to get feedback if your metadata or DOI is incorrect?
- When a computational figure or data output changes, how long does it take to integrate that into your document?
- How can you assess and test that the structure of a document applies to editorial rules?

In science there is a highly manual, absurdly expensive and disconnected process between authoring and publishing. By moving to continuous practices and investing in the appropriate infrastructure to support *continuous science* we believe there is an opportunity to accelerate scientific discovery by multiple orders of magnitude while simultaneously increasing reproducibility, robustness and transparency of the underlying science.

### *1.4. Article Outline*

For research-communication to be transformative on a similar scale as continuous practices for software delivery, researchers require modern tools for authoring and publishing. There are two inter-related capabilities that are necessary for this transition:

1. authoring mediums that support data, computation and structured content without the need for expensive typesetting; and
2. publishing that is open and accessible to researchers at a variety of scales – individual publishing, lab-groups, societies and institutions.

[10] As an example of the the rapid and timely sharing of in-progress results and data, it is worth reflecting on the COVID-19 pandemic. Improved and more rapid sharing practices through data and preprints helped to develop a vaccine in record time [20]. Researchers published and shared data on the DNA sequence which enabled the design of a vaccine candidate in **two days** and the first manufacturing within **two weeks**.

> By the second week of January 2020, researchers in China published the DNA sequence of SARS-CoV-2, the coronavirus that causes COVID-19. By early February, a COVID-19 vaccine candidate had been designed and manufactured.
>
> — https://covid19.nih.gov

> In Massachusetts, the Moderna vaccine design took all of one weekend, and had been designed by January 13, two days after the genetic sequence had been made public.
>
> — https://nymag.com

[11] In N. C. Penfold and J. K. Polka [22], they describe the importance of adopting preprints, as the "overall peer review process can take years". For the programmers reading this, it is worth a mental comparison to the pain of a pull-request being open for years. Having relevant feedback that is close to the time of implementation or writing is invaluable.

Through the lens of MyST Markdown and Curvenote, this paper will explore how these tools aim to address critical gaps in current scientific publishing practices. Our motivation is to enhance the *speed* and *impact* of research dissemination, fostering a scientific ecosystem that is more collaborative, reproducible, and equipped to tackle the urgent global challenges of our time.

## 2. Authoring Tools

MyST Markdown (Markedly Structured Text, https://mystmd.org) is a community-driven markup language that is a superset of CommonMark (a standard form of Markdown) with special syntax for citations, cross-references, and block and inline extension points called "directives" and "roles". The block-level content provides multi-line containers surrounded by either backticks or colons; examples include callout panels, figures, equations and tables (see documentation). There is also specialized support for the types of metadata that are important to collect for scientific articles (funding, ORCIDs, CRediT Roles, etc.). In 2022, the Executable Books project (https://executablebooks.org, which hosts Jupyter Book and MyST) started work on the `mystmd` command line interface (CLI), which was initially developed as the Curvenote CLI, and later transferred to the ExecutableBooks project. In June 2024, MyST Markdown officially became part of Project Jupyter (see enhancement proposal). This tool allows authors writing in MyST Markdown to easily build websites and documents and supports the JupyterLab MyST plugin. MyST is influenced by reStructuredText (RST) and Sphinx – pulling on the nomenclature and introducing additional standards where appropriate. There are also many intentional syntax similarities of MyST Markdown to R-Markdown [7], Pandoc, and Quarto (https://quarto.org), especially in citation syntax and frontmatter structure. MyST Markdown is written in Javascript and builds upon the `unified` community of tools, output formats and transforms, whereas Quarto builds upon Pandoc written in Haskell + Lua; there are also differences in approach in the legal and community foundations (Pandoc and Quarto are GPL vs. MyST which is a more permissive MIT license; MyST is a community-driven project by Project Jupyter whereas Quarto is driven by a single corporate entity). The initial use case driving the development and design of MyST Markdown has been JupyterBook, which can create educational online textbooks and tutorials with Jupyter Notebooks and narrative content written in MyST.

This article will not attempt to describe the markup syntax directly, for that we suggest browsing the documentation at https://mystmd.org, instead we will focus our attention on the use cases for scientific publishing that we are trying to make as easy as possible. Specifically, the ability to add persistent identifiers (PIDs) and links to other structured content; hover previews to show details on demand; integrating live computational content and interactive figures; and exporting to many different formats including those used by scholarly publishing.

### *2.1. Utility of Links and Identifiers*

In MyST Markdown, citations can be added inline using `@cite-key`, following Pandoc-style syntax and referencing a BibTeX file in a project. It is also possible to directly link to DOIs using `@10.5281/zenodo.6476040`, which will create a hover reference [31] as well as a references section at the bottom of the page.

This enhanced-links concept can be extended to Wikipedia, RRIDs, RORs, GitHub issues or code, and other scientific databases to augment writing. For example, the link `<rrid:SCR_008394>` becomes SCR_008394, with rich-metadata and citations created. Wikipedia links come with previews, for example, `<wiki:gravitational_waves>` becomes gravitational waves. GitHub links to pull-requests also give hover information, for example, the following link #87 shows a hover preview in the online-version.

The use of DOIs and other structured scientific metadata can be reused in multiple different formats such as JATS XML and CrossRef deposits to create a DOI. Our goal with these integrations is to make the use of persistent identifiers (PIDs) both easy and rewarding to the author as they are writing. In traditional typesetting, this metadata is only added at publication time requiring specialized vendors and/or proprietary technology.

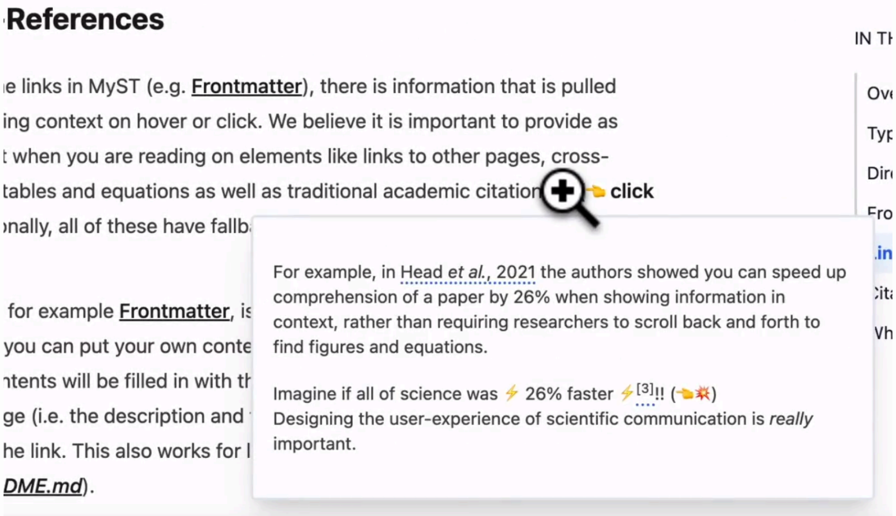
### *2.2. Hover Previews for Content and Metadata*

In A. Head *et al.* [32] the authors show a speed up in comprehension of an article by 26% when showing information in context (i.e. "details on demand" [33]), rather than requiring researchers to scroll back and forth to find figures and equations. MyST supports these concepts natively for cross-referencing equations, figures, and tables using hover-previews Example 1. This enhances the reading experience of scientific documents and retrieval of information.

In MyST Markdown we have also extended this "details on demand" concept to abbreviations to make it trivial to disambiguate the meaning of acronyms. In an analysis of over 18 million articles in A. Barnett and Z. Doubleday [34], the authors found that the vast majority of abbreviations (79%) appeared fewer than 10 times and many abbreviations had conflicting meanings even in the same discipline. In MyST Markdown, there is a trivial way to document abbreviations in YAML frontmatter or the project configuration Program 1, these are then applied to all instances of that abbreviation in the article or notebooks giving a hover-preview and accessible HTML (e.g. try hovering over these in the online version: JATS, XML, VoR).

**Example 1**. *Hover and Dive Deeper*

Any figure, table, or equation can be referenced in MyST and in addition to automated numbering the cross-references have hover-references. This design feature is important for two reasons: (1) it improves reading comprehension; and (2) it focuses on structured data which can be accessible between papers, creating an open-ecosystem of machine-actionable, reusable content. The referenced content can also be interactive or computational.



References

ie links in MyST (e.g. Frontmatter), there is information that is pulled
ing context on hover or click. We believe it is important to provide as
t when you are reading on elements like links to other pages, cross-
tables and equations as well as traditional academic citation
ınally, all of these have fallb

for example Frontmatter, is
you can put your own conte
ıtents will be filled in with th
ge (i.e. the description and
he link. This also works for l
DME.md).

For example, in Head *et al.*, 2021 the authors showed you can speed up comprehension of a paper by 26% when showing information in context, rather than requiring researchers to scroll back and forth to find figures and equations.

Imagine if all of science was ⚡ 26% faster ⚡[3]!! (👈💥)
Designing the user-experience of scientific communication is *really* important.

**Figure 1**. *Instantly accessible information can deep-dive link all the way to interactive figures. These practices help with reading comprehension by around 26% by providing information when the reader needs it [32].*

```yaml
abbreviations:
  UA: ulnar artery
```

**Program 1**. *YAML metadata used in MyST frontmatter to give accessible hovers to all abbreviations with minimal effort for the author. In this case, the acronym* UA *has over 18 distinct meanings in medicine [35] not to mention other disciplines.*

### 2.3. *Integrating Computational Content*

Beyond structured typography, integrated metadata and hover-previews, MyST Markdown understands computational content and has been integrated with Jupyter [36]. The goal of this is two fold: (1) allowing for updates to computational content, figures, tables, and calculations to directly update documents; and (2) to bring interactive figures and integrated computation directly into articles. In the composition of a scientific narrative, scientists often use individual notebooks to create various components of their research (e.g. the preparation of a single figure, table or calculation). The outputs of these notebooks (a figure, table, or calculation) can then be used in a main, narrative-driven document — a scientific article or presentation.

In some cases, it is possible to collapse all computational information into a single article, and visually hide the code to focus on the narrative or presentational flow; an approach experimented with by eLife [10]. This approach is appropriate for tutorials or the reproduction of visualizations rather than reproduction of a distinct detailed methodology that requires its own explanation and/or lengthy computation. Another approach is to include supplemental notebooks that can capture those individual steps [9], and transclude content Figure 2. Both approaches are appropriate in different circumstances, and depends on the goal of the communication, nature of the research, speed of execution, and if individual steps require dedicated narrative explanation. Additionally, the possibility of publishing a computational article, allows authors to rethink how to communicate their work and prepare specific visualizations and compact results datasets to take advantage of the format.

In Figure 2, we show an example of reusing computational outputs, such as figures or tables directly in a single computational research article. By *embedding* these rather than using a screenshot or copy-paste, any changes to the computational content can be immediately applied on a re-render. The embedding is completed through a simple MyST Markdown syntax that references a labeled cell in a Jupyter Notebook in, for example, a figure or table Program 2.

Similar to the hover-references in Example 1, this approach improves the metadata around the notebooks and exposes individual outputs or code-snippets to be imported into other

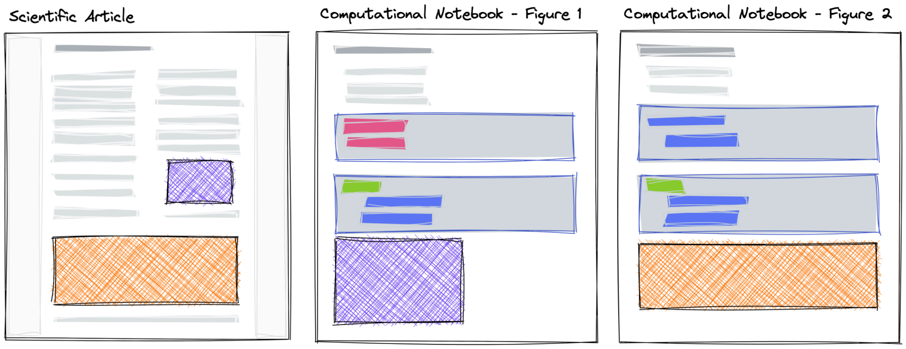

Scientific Article
Computational Notebook - Figure 1
Computational Notebook - Figure 2

**Figure 2**. *A schematic of embedding content from Jupyter Notebooks into an article. The purple and orange components, interactive figure or other computational outputs, are created in a computational notebook and subsequently used in other narrative or presentation-focused scientific article.*

```
:::{figure} #embedded-cell
Additional caption.
:::
```

**Program 2.** *Embedding a cell from a supplementary notebook directly into a computational document by referencing the label/ID of the cell and adding a caption. The cell in the notebook must be labeled with a* `#| label: embedded-cell` *as the first line of the content.*

documents or projects. Additionally, we can attach either static-interactivity (e.g. Plotly, Altair) or dynamic computation (e.g. BinderHub, JupyterHub or JupyterLite) to these figures to run live computations directly in the article Example 2. Here we are aiming at a much richer, structured information commons that moves beyond just tracking scientific metadata towards easy-to-use tools that reuse scientific content.

### *2.4. Single Source to Many Outputs*

These capabilities of cross-references, typography and embedding visualizations and dataframes are complemented by a single-source export system that supports professional article templates and JATS XML Figure 4. This is referred to as single-source publishing [37], however, many implementations in scientific publishing focus first on manual translation to XML (e.g. from Word or LaTeX), rather than on an author-facing implementation.

**Example 2.** *Computational Reproducibility and Interactivity*

MyST allows for the full reproducible environment to be specified (via REES) and reproduced through tools like MyBinder. Figures can be integrated directly into articles, pressing a button to launch live and interactive figures that build upon the Jupyter ecosystem. These tools build on the Jupyter protocols and reuse components from the JupyterLab ecosystem, extending that into various pages using a package called `thebe`.

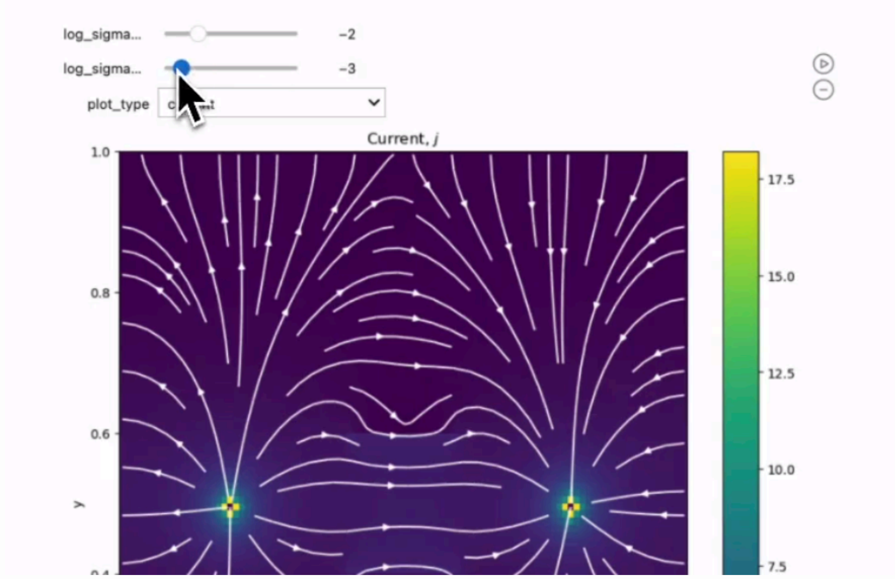


**Figure 3.** *Embedded notebook cells with live computation directly in an articles with computation backed by Jupyter. These can be running on BinderHub or directly in your browser through JupyterLite.*

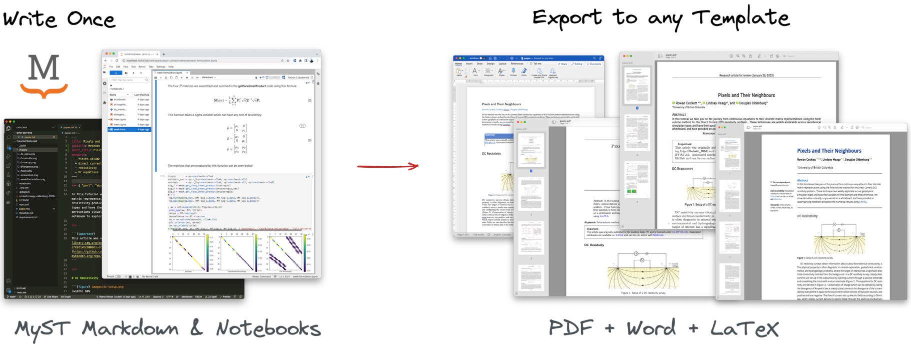


**Figure 4**. *Export to PDF using LaTeX or Typst is supported for hundreds of different journal templates in addition to Microsoft Word or JATS XML, which is used throughout scientific publishing (showing content from R. Cockett, L. J. Heagy, and D. W. Oldenburg [38] CC-BY-SA-4.0).*

With single-source publishing, we can rely on rich transformations of the source content that can create professional PDFs, interactive HTML, and structured metadata such as JATS XML, which is the current standard for scientific archiving and text-mining.

## 3. Publishing

The native way to publish or share a MyST article is through content-as-data, a collection of JSON files that represent the full documents and all associated metadata (if you are reading this online, you can add a `.json` to the URL to access the content). This MyST content may be served alongside an independent website theme, which dynamically creates HTML based on these JSON files (very similar to XML-based single source publishing [37], but using modern web-tooling). This server-side approach has a number of advantages, in that it allows you to maintain a journal/site theme, without having to upgrade or rebuild all content as would be required by static tooling. It also puts content addressability as a first class concern (via the `.json`), enabling global cross referencing and opening up opportunities for varied publishing models in future. This is the approach that we take with Curvenote journals, and provide managed services to maintain journal sites, manage and curate content, as well as provide the editorial management tools and workflows.

It is also possible to share MyST Markdown as a static HTML site using GitHub Pages. To create a static site on GitHub pages you can run `myst init --gh-pages`, which will walk you through the steps of creating a GitHub action to publish your content statically. In this scenario, a static HTML site is built from your content which can be hosted as any other static website, while some of the advantages of dynamic hosting are lost, it is an easy and accessible way for individuals to self-publish.

In 2024, Curvenote was asked to support the SciPy Proceedings and re-imagine a MyST based publishing approach that uses GitHub for open-peer-review, implementing a submission, editorial and peer review process with GitHub pull-requests. The original SciPy proceedings infrastructure was developed around 2010 and has influenced the design of the Journal of Open Source Software (JOSS), which has popularized GitHub-based peer-review [6]. In 2024, the workflow for SciPy Proceedings was updated to use MyST Markdown for the authoring process[12] (previously RST and LaTeX were supported, and the build process was in Sphinx), and the submission process now uses the open-source Curvenote CLI in combination with dedicated open-source Curvenote GitHub Actions to build, check, and preview the content of each commit, using GitHub workflows to automate the process, providing immediate feedback for authors and the conference editorial team.

[12] LaTeX is also supported by parsing and rendering directly with the `mystmd` CLI, this is completed through `@unified-latex`.

### 3.1. *Structural Checks*

The open-source Curvenote CLI (https://github.com/curvenote/curvenote) provides checks for the structure of a document to ensure it meets automated quality controls for things like references, valid links, author identifiers (e.g. ORCID), or funding information. Executing `curvenote check` in a MyST project will build the project and use the structured data to assess metadata (e.g. do authors provide ORCIDs), structural checks (e.g. does the article have an abstract?; is the article below a word count?), and check that references have valid DOIs.

We have designed these checks in a similar pattern to linting and/or unit tests in scientific software, which continually give feedback on the structural health of a codebase with near immediate feedback to authors Example 3. Authors can take action to improve their metadata directly, by including DOIs, CRediT Roles, ORCIDs, and structural checks such as word count or missing sections. For example, in the SciPy Proceedings in 2024, which used Curvenote checks for their submission system, required and optional metadata were improved by authors reacting to these automated checks without any intervention from the proceedings editorial team (e.g. scipy-conference/scipy_proceedings#915 added CRediT roles, ORCIDs abbreviations, and DOIs to get a passing check Figure 6). This is a low-friction way of improving metadata at the time of authoring or submission to elevate content to the standards that are required for a certain type of publication (e.g. proceedings vs. blog post vs. peer-reviewed journal).

### 3.2. *Automated Actions*

Upon submission of there are a number of checks that are run and a submission or draft is deployed to Curvenote's platform, which stores the structured MyST project. We utilized GitHub Actions to automate initial checks and generate previews of submissions (via https://github.com/curvenote/actions). The actions automatically generate a preview of the manuscript exactly as it would appear in publication using MyST Markdown, and these are linked within the pull request comments for easy access by reviewers Figure 6.

### 3.3. *Continuous Practices*

MyST Markdown is a simple text-based format that can integrate directly with computational analysis and results in notebooks or scripts. This simplicity means a researcher's manuscript can be easily created and maintained as an integral part of their research code base as they are working. This enables, for example, am article or draft that can be automatically rebuilt with the latest figures and data tables on every change, and any issues that you would hit on submission or publication are flagged as they are created. This is one of the ways that continuous science practices Section 1.3 can lead to radically improved efficiencies at scale.

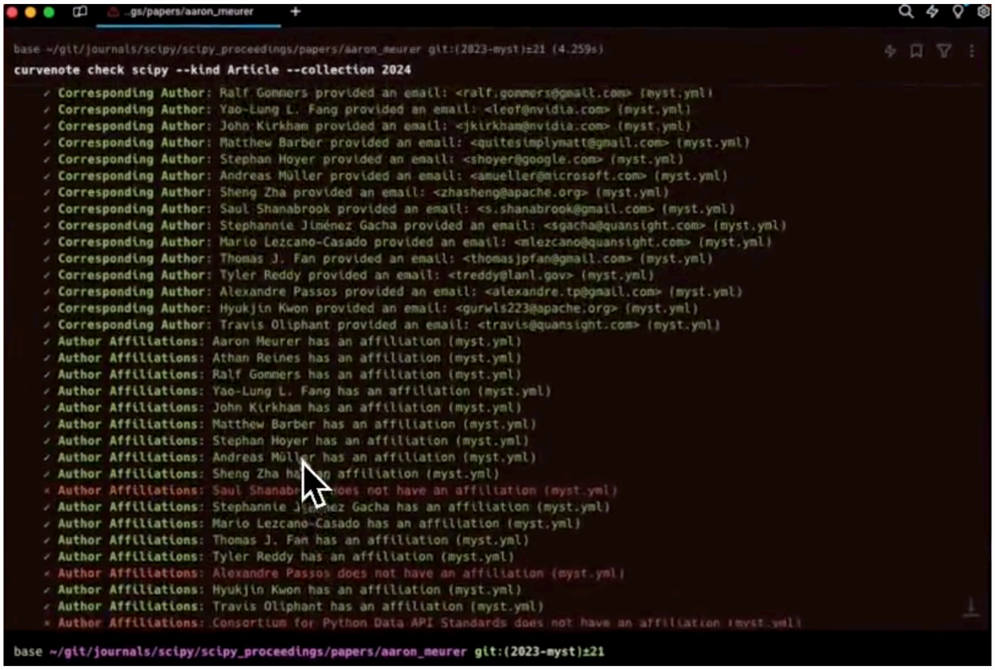
Curvenote's tools build on this theme of enabling researchers to have more visibility into how their work will appear when submitted and published. The previews generated during the journal submission process, or by the actions on a researchers own repo can also be generated by an author at any time. By running `curvenote submit <journal> --draft` researchers can get a set of check results and a preview in the style of that venue, and where submissions of `Computational Articles` are permitted, authors and reviewers will even get feedback on reproducible environment check that any interactive figures and notebooks execute as expected.

### 3.4. *Use Cases*

The rapid feedback in authoring Section 2 coupled with structural checks Section 3.1, automation Section 3.2, and archiving of the content opens workflows for varied sizes

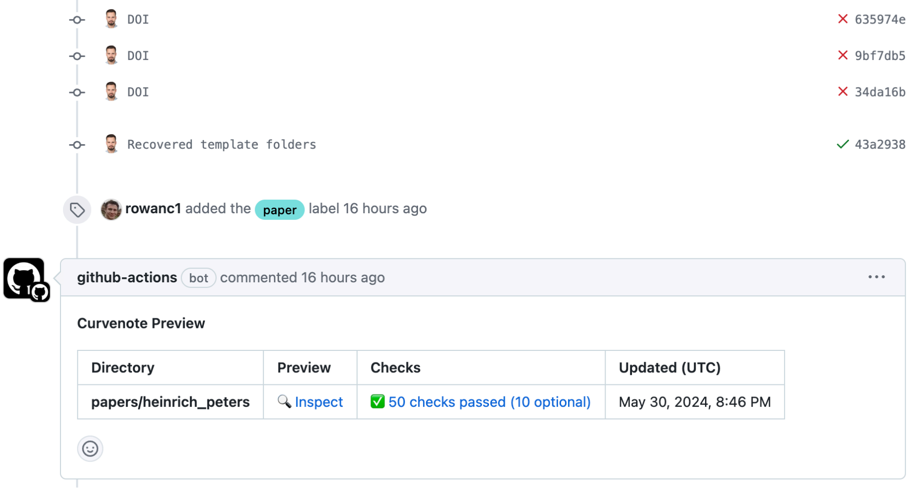
**Example 3**. *Structural Checks and Metadata Checks*

Specific checks can be setup for any kind of content being submitted with MyST, for example, a "Short Report" might have different length requirements or metadata standards than a "Research Article" or "Editorial". Using Curvenote, these checks can be configured for a specific venue or kind of article, for example, to check specifically for SciPy Proceedings articles, `curvenote check scipy --kind Article --collection 2024` can be run, where the list of checks is configured remotely by journal administrators.



```
base ~/git/journals/scipy/scipy_proceedings/papers/aaron_meurer git:(2023-myst)±21 (4.259s)
curvenote check scipy --kind Article --collection 2024
    ✓ Corresponding Author: Ralf Gommers provided an email: <ralf.gommers@gmail.com> (myst.yml)
    ✓ Corresponding Author: Yao-Lung L. Fang provided an email: <leof@nvidia.com> (myst.yml)
    ✓ Corresponding Author: John Kirkham provided an email: <jkirkham@nvidia.com> (myst.yml)
    ✓ Corresponding Author: Matthew Barber provided an email: <quitesimplymatt@gmail.com> (myst.yml)
    ✓ Corresponding Author: Stephan Hoyer provided an email: <shoyer@google.com> (myst.yml)
    ✓ Corresponding Author: Andreas Müller provided an email: <amueller@microsoft.com> (myst.yml)
    ✓ Corresponding Author: Sheng Zha provided an email: <zhasheng@apache.org> (myst.yml)
    ✓ Corresponding Author: Saul Shanabrook provided an email: <s.shanabrook@gmail.com> (myst.yml)
    ✓ Corresponding Author: Stephannie Jiménez Gacha provided an email: <sgacha@quansight.com> (myst.yml)
    ✓ Corresponding Author: Mario Lezcano-Casado provided an email: <mlezcano@quansight.com> (myst.yml)
    ✓ Corresponding Author: Thomas J. Fan provided an email: <thomasjpfan@gmail.com> (myst.yml)
    ✓ Corresponding Author: Tyler Reddy provided an email: <treddy@lanl.gov> (myst.yml)
    ✓ Corresponding Author: Alexandre Passos provided an email: <alexandre.tp@gmail.com> (myst.yml)
    ✓ Corresponding Author: Hyukjin Kwon provided an email: <gurwls223@apache.org> (myst.yml)
    ✓ Corresponding Author: Travis Oliphant provided an email: <travis@quansight.com> (myst.yml)
    ✓ Author Affiliations: Aaron Meurer has an affiliation (myst.yml)
    ✓ Author Affiliations: Athan Reines has an affiliation (myst.yml)
    ✓ Author Affiliations: Ralf Gommers has an affiliation (myst.yml)
    ✓ Author Affiliations: Yao-Lung L. Fang has an affiliation (myst.yml)
    ✓ Author Affiliations: John Kirkham has an affiliation (myst.yml)
    ✓ Author Affiliations: Matthew Barber has an affiliation (myst.yml)
    ✓ Author Affiliations: Stephan Hoyer has an affiliation (myst.yml)
    ✓ Author Affiliations: Andreas Müller has an affiliation (myst.yml)
    ✓ Author Affiliations: Sheng Zha has an affiliation (myst.yml)
    × Author Affiliations: Saul Shanabrook does not have an affiliation (myst.yml)
    ✓ Author Affiliations: Stephannie Jiménez Gacha has an affiliation (myst.yml)
    ✓ Author Affiliations: Mario Lezcano-Casado has an affiliation (myst.yml)
    ✓ Author Affiliations: Thomas J. Fan has an affiliation (myst.yml)
    ✓ Author Affiliations: Tyler Reddy has an affiliation (myst.yml)
    × Author Affiliations: Alexandre Passos does not have an affiliation (myst.yml)
    ✓ Author Affiliations: Hyukjin Kwon has an affiliation (myst.yml)
    ✓ Author Affiliations: Travis Oliphant has an affiliation (myst.yml)
    × Author Affiliations: Consortium for Python Data API Standards does not have an affiliation (myst.yml)
base ~/git/journals/scipy/scipy_proceedings/papers/aaron_meurer git:(2023-myst)±21
```

**Figure 5**. *Running* `curvenote check` *on a SciPy Proceedings article to check for missing DOIs, author ORCIDs, word-count and specific sections (e.g. abstract). These can be run in less than a few seconds both locally and through GitHub actions Figure 6, which provides a user interface on the checks.*



```
DOI                                   × 635974e
DOI                                   × 9bf7db5
DOI                                   × 34da16b
Recovered template folders            ✓ 43a2938
```

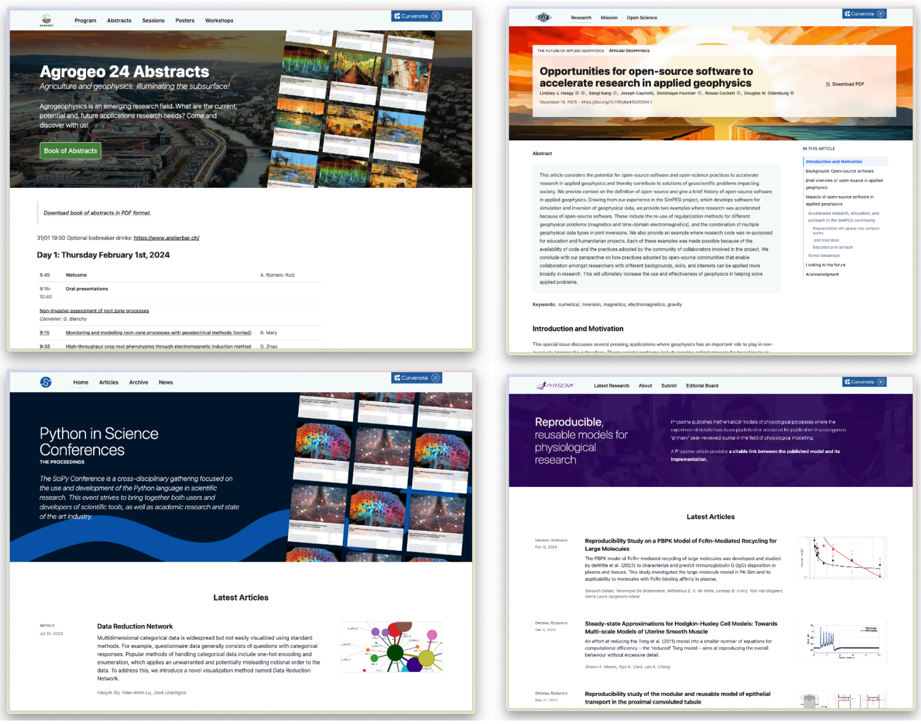
**rowanc1** added the paper label 16 hours ago

**github-actions** bot commented 16 hours ago

**Curvenote Preview**

| Directory | Preview | Checks | Updated (UTC) |
|---|---|---|---|
| papers/heinrich_peters | 🔍 Inspect | ✅ 50 checks passed (10 optional) | May 30, 2024, 8:46 PM |

**Figure 6**. *An example of a comment by a GitHub action, which shows the checks and preview of the article directly. The checks in this example have promoted the author to improve metadata, see scipy-conference/scipy_proceedings#911.*



Agrogeo 24 Abstracts

*Agriculture and geophysics: Illuminating the subsurface!*

Agrogeophysics is an emerging research field. What are the current, potential and, future applications research needs? Come and discover with us!

*Download book of abstracts in PDF format.*

31/01 19:00 Optional Icebreaker drinks: https://www.atelierbar.ch/

Day 1: Thursday February 1st, 2024

| | | |
|---|---|---|
| 8:45 | **Welcome** | A. Romero-Ruiz |
| 9:15-<br>12:45 | **Oral presentations** | |
| Non-invasive assessment of root zone processes<br>*Convener: G. Blanchy* | | |
| 9:15 | Monitoring and modelling root-zone processes with geoelectrical methods (invited) | B. Mary |
| 9:35 | High-throughput crop root phenotyping through electromagnetic induction method | D. Zhao |

THE FUTURE OF APPLIED GEOPHYSICS | APPLIED GEOPHYSICS

**Opportunities for open-source software to accelerate research in applied geophysics**

Lindsey J. Heagy, Seogi Kang, Joseph Capriotti, Dominique Fournier, Rowan Cockett, Douglas W. Oldenburg

December 18, 2023 · https://doi.org/10.1190/tle43020084.1

Abstract

This article considers the potential for open-source software and open-science practices to accelerate research in applied geophysics and thereby contribute to solutions of geoscientific problems impacting society. We provide context on the definition of open-source and give a brief history of open-source software in applied geophysics. Drawing from our experience in the SimPEG project, which develops software for simulation and inversion of geophysical data, we provide two examples where research was accelerated because of open-source software. These include the re-use of regularization methods for different geophysical problems (magnetics and time-domain electromagnetics), and the combination of multiple geophysical data types in joint inversions. We also provide an example where research code was re-purposed for education and humanitarian projects. Each of these examples was made possible because of the availability of code and the practices adopted by the community of collaborators involved in the project. We conclude with our perspective on how practices adopted by open-source communities that enable collaboration amongst researchers with different backgrounds, skills, and interests can be applied more broadly in research. This will ultimately increase the use and effectiveness of geophysics in helping solve applied problems.

Keywords: numerical, inversion, magnetics, electromagnetics, gravity

Introduction and Motivation

This special issue discusses several pressing applications where geophysics has an important role to play in non-

IN THIS ARTICLE

- Introduction and Motivation
- Background: Open-source software
- Brief overview of open-source in applied geophysics
- Impacts of open-source software in applied geophysics
  - Accelerated research, education, and outreach in the SimPEG community
    - Regularization with sparse and compact norms
    - Joint inversions
    - Education and outreach
  - Some takeaways
- Looking to the future
- Acknowledgment

Python in Science Conferences

THE PROCEEDINGS

*The SciPy Conference is a cross-disciplinary gathering focused on the use and development of the Python language in scientific research. This event strives to bring together both users and developers of scientific tools, as well as academic research and state of the art industry.*

Latest Articles

ARTICLE
Jul 10, 2023

**Data Reduction Network**

Multidimensional categorical data is widespread but not easily visualized using standard methods. For example, questionnaire data generally consists of questions with categorical responses. Popular methods of handling categorical data include one-hot encoding 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.

*Haoyin Xu, Haw-minn Lu, José Unpingco*

Reproducible, reusable models for physiological research

Physiome publishes mathematical models of physiological processes where the experimental details have been published or accepted for publication in a recognised 'primary' peer-reviewed journal in the field of physiological modeling.

A Physiome article provides **a citable link between the published model and its implementation.**

Latest Articles

ORIGINAL RESEARCH
Feb 12, 2024

**Reproducibility Study on a PBPK Model of FcRn-Mediated Recycling for Large Molecules**

The PBPK model of FcRn-mediated recycling of large molecules was developed and studied by deWitte et al. (2023) to characterize and predict Immunoglobulin G (IgG) disposition in plasma and tissues. This study investigated the large-molecule model in PK-Sim and its applicability to molecules with FcRn binding affinity in plasma.

*Soroush Safaei, Veronique De Brabandere, Wilhelmus E. A. de Witte, Lindsay B. Avery, Tom Van Bogaert, Maria Laura Sargentini-Maier*

ORIGINAL RESEARCH
Dec 4, 2023

**Steady-state Approximations for Hodgkin-Huxley Cell Models: Towards Multi-scale Models of Uterine Smooth Muscle**

An effort at reducing the Tong et al. (2011) model into a smaller number of equations for computational efficiency – the 'reduced' Tong model – aims at reproducing the overall behaviour without excessive detail.

*Shawn A. Means, Alys R. Clark, Leo K. Cheng*

ORIGINAL RESEARCH
May 27, 2023

**Reproducibility study of the modular and reusable model of epithelial transport in the proximal convoluted tubule**

**Figure 7**. *MyST Markdown and Curvenote tools can help lower the barrier to entry for scientific publishing workflows for journals, research institutes, conferences, private consortiums, universities, and lab groups.*

of teams to adopt these *continuous science* practices. For example, individuals publishing static MyST Markdown sites, lab-groups creating a better archive to highlight the research contributions of their team (e.g. Applied Geophysics), conference proceedings (e.g. SciPy Proceedings), or more formalized society journals (e.g. Physiome, American Geophysical Union, Elemental Microscopy) Figure 7.

## 4. Conclusions

Scientific publishing infrastructure is disconnected from the day-to-day workflows of scientists, and at scale this slows the progress of science. These misalignments are particularly pronounced in computational disciplines, where rapid evolution of methodologies, software, and data demands equally dynamic and interconnected platforms. This gap — between the authoring/doing of research and the communicating/publishing of the research — slows the speed of research dissemination, reuse, and uptake and completely impedes "networked knowledge" and importing/reusing work in a structured way. For example, "importing" visualizations, equations or any other deeply-linked content – including provenance information – into new research articles, documentation or educational sites is completely impossible in today's research ecosystem. As a metaphor, compare open-access science to open-source programming: it would be a world without package managers to share, version, reuse, and rapidly build upon other peoples work in a structured way. The open-source ecosystem would not exist without this infrastructure.

Open infrastructure for communicating science also has to be easy to integrate into existing tools, support computational, interactive components, be archivable for the long term, and be adopted by our existing sociotechnical system of societies, journals, and institutions. There are two interconnected problems that need to be solved: (1) upgrade existing scientific authoring tools, ensuring these are integrated into both scientific and data-science ecosystems; and (2) develop radically better ways to share content as individuals, small groups, preprints, and formalized, traditional journals with existing societies and institutions. The two problems are connected, in that the authoring tools should be able to deeply integrate with publishing mediums (e.g. referencing a figure from a publication should be able to show you that figure directly as you are authoring, including all interactivity and computation).

In this article we have presented some of the goals behind features of MyST Markdown and Curvenote, to support new open-science infrastructure including authoring tools as well as publishing workflows that support checks and automation. To support every next-generation research tool on top of open access knowledge, we need access to **high-quality, structured content**, data and software — not just the scholarly metadata and citation graph. Our goal of contributing to MyST Markdown is to make the processes behind creating this structured data more accessible and affordable. These tools in the hands of researchers can also enable process changes and **continuous science** practices: where checking and automation can support rapid iterations and feedback. The analogies between continuous delivery of software and continuous science give us an opportunity to peek ahead a decade to an analogous future and draw on many learnings on how to organize and focus infrastructure to get the best out of our scientific community.

There is, of course, an *enormity* of work ahead of these tools to transform science publishing at scale. We are grateful to our society partners who are changing community practices around publishing to support HTML-first publishing, experimenting with computational articles, and implementing new peer-review workflows. Tools on their own do not make change, but can help to enable it. Improvements to scientific publishing require many diverse community efforts to improve the quality and speed of how we communicate knowledge, and ultimately to accelerate scientific progress.

## Acknowledgements

Portions of this manuscript have been previously shared in the Curvenote and MyST Markdown documentation. The authors would like to thank Lindsey Heagy, Arfon Smith, Chris Holdgraf, Greg Caporaso, Jim Colliander, Fernando Perez, J.J. Allaire, and Kristen Ratan who have provided input on versions of these ideas over the last few years. We would also like to thank the reviews Nate Jacobs, Angus Hollands, Lindsey Heagy, Stefan van der Walt, Andy Terrel, and Hongsup Shin who helped improved the manuscript. Funding for portions of this work have come from the Sloan Foundation (for MyST Markdown and *Notebooks Now!*) and Alberta Innovates (for the initial version of the Curvenote CLI, which became `mystmd`). Thank you to the growing list of MyST Markdown contributors who continue to make MyST a fantastic community project.

## References

[1] P. E. Bourne *et al.*, “Improving The Future of Research Communications and e-Scholarship (Dagstuhl Perspectives Workshop 11331),” 2012, doi: 10.4230/DAGMAN.1.1.41.

[2] M. D. Wilkinson *et al.*, “The FAIR Guiding Principles for scientific data management and stewardship,” *Scientific Data*, vol. 3, no. 1, 2016, doi: 10.1038/sdata.2016.18.

[3] A. Grossmann and B. Brembs, “Current market rates for scholarly publishing services,” *F1000Research*, vol. 10, p. 20, 2021, doi: 10.12688/f1000research.27468.2.

[4] M. Hagve, “Pengene bak vitenskapelig publisering,” *Tidsskrift for Den norske legeforening*, 2020, doi: 10.4045/tidsskr.20.0118.

[5] A. Clotworthy *et al.*, “Saving time and money in biomedical publishing: the case for free-format submissions with minimal requirements,” *BMC Medicine*, vol. 21, no. 1, 2023, doi: 10.1186/s12916-023-02882-y.

[6] A. M. Smith *et al.*, “Journal of Open Source Software (JOSS): design and first-year review,” *PeerJ Computer Science*, vol. 4, p. e147, 2018, doi: 10.7717/peerj-cs.147.

[7] Y. Xie, J. J. Allaire, and G. Grolemund, *R Markdown: The Definitive Guide*. Chapman, 2018. doi: 10.1201/9781138359444.

[8] Project Jupyter *et al.*, "Binder 2.0 - Reproducible, interactive, sharable environments for science at scale," in *Proceedings of the 17th Python in Science Conference*, in SciPy. 2018. doi: 10.25080/majora-4af1f417-011.

[9] G. Caprarelli, B. Sedora, M. Ricci, S. Stall, and M. Giampoala, "Notebooks Now! The Future of Reproducible Research," *Earth and Space Science*, vol. 10, no. 12, 2023, doi: 10.1029/2023ea003458.

[10] E. Tsang and G. Maciocci, "Welcome to a new ERA of reproducible publishing," *ELife*, 2020, [Online]. Available: https://elifesciences.org/labs/dc5acbde/welcome-to-a-new-era-of-reproducible-publishing

[11] A. Heidt, "A publishing platform that places code front and centre," *Nature*, 2024, doi: 10.1038/d41586-024-02577-1.

[12] A. Mishra and Z. Otaiwi, "DevOps and software quality: A systematic mapping," *Computer Science Review*, vol. 38, p. 100308, 2020, doi: 10.1016/j.cosrev.2020.100308.

[13] L. Leite, C. Rocha, F. Kon, D. Milojicic, and P. Meirelles, "A Survey of DevOps Concepts and Challenges," *ACM Computing Surveys*, vol. 52, no. 6, pp. 1–35, 2019, doi: 10.1145/3359981.

[14] L. Chen, "Continuous Delivery: Huge Benefits, but Challenges Too," *IEEE Software*, vol. 32, no. 2, pp. 50–54, 2015, doi: 10.1109/ms.2015.27.

[15] DORA, "Accelerate: State of DevOps 2018: Strategies for a New Economy." [Online]. Available: https://services.google.com/fh/files/misc/state-of-devops-2018.pdf

[16] A. N. Link, C. A. Swann, and B. Bozeman, "A time allocation study of university faculty," *Economics of Education Review*, vol. 27, no. 4, pp. 363–374, 2008, doi: 10.1016/j.econedurev.2007.04.002.

[17] T. M. Evans, L. Bira, J. B. Gastelum, L. T. Weiss, and N. L. Vanderford, "Evidence for a mental health crisis in graduate education," *Nature Biotechnology*, vol. 36, no. 3, pp. 282–284, 2018, doi: 10.1038/nbt.4089.

[18] R. Blinde, "DevOps Unravelled: A Study on the Effects of Practices and Technologies on Organisational Performance," Leiden, Netherlands, 2022.

[19] M. Callanan and A. Spillane, "DevOps: Making It Easy to Do the Right Thing," *IEEE Software*, vol. 33, no. 3, pp. 53–59, 2016, doi: 10.1109/ms.2016.66.

[20] C. Watson, "Rise of the preprint: how rapid data sharing during COVID-19 has changed science forever," *Nature Medicine*, vol. 28, no. 1, pp. 2–5, 2022, doi: 10.1038/s41591-021-01654-6.

[21] K. Powell, "Does it take too long to publish research?," *Nature*, vol. 530, no. 7589, pp. 148–151, 2016, doi: 10.1038/530148a.

[22] N. C. Penfold and J. K. Polka, "Technical and social issues influencing the adoption of preprints in the life sciences," *PLOS Genetics*, vol. 16, no. 4, p. e1008565, 2020, doi: 10.1371/journal.pgen.1008565.

[23] P. Jones and M. Hahnel, "How and Why Data Repositories are Changing Academia," *Against the Grain*, vol. 28, no. 1, 2016, doi: 10.7771/2380-176x.7269.

[24] A. Martinez Garcia, J. Kaye, and A. Freeman, "Enabling open research practices - connecting the Octopus platform with research institutional repositories." [Online]. Available: https://www.repository.cam.ac.uk/handle/1810/350138

[25] T. Clark, P. N. Ciccarese, and C. A. Goble, "Micropublications: a semantic model for claims, evidence, arguments and annotations in biomedical communications," *Journal of Biomedical Semantics*, vol. 5, no. 1, p. 28, 2014, doi: 10.1186/2041-1480-5-28.

[26] P. Groth, A. Gibson, and J. Velterop, "The anatomy of a nanopublication," *Information Services &amp; Use*, vol. 30, no. 1–2, pp. 51–56, 2010, doi: 10.3233/isu-2010-0613.

[27] L. Teytelman, A. Stoliartchouk, L. Kindler, and B. L. Hurwitz, "Protocols.io: Virtual Communities for Protocol Development and Discussion," *PLOS Biology*, vol. 14, no. 8, p. e1002538, 2016, doi: 10.1371/journal.pbio.1002538.

[28] B. A. Nosek, C. R. Ebersole, A. C. DeHaven, and D. T. Mellor, "The preregistration revolution," *Proceedings of the National Academy of Sciences*, vol. 115, no. 11, pp. 2600–2606, 2018, doi: 10.1073/pnas.1708274114.

[29] M. Lenharo, "Will the Gates Foundation's preprint-centric policy help open access?," *Nature*, 2024, doi: 10.1038/d41586-024-00996-8.

[30] M. A. Johansson and D. Saderi, "Open peer-review platform for COVID-19 preprints," *Nature*, vol. 579, no. 7797, p. 29, 2020, doi: 10.1038/d41586-020-00613-4.

[31] R. Cockett, "Future of Research Communication &amp; Collaboration," 2022, doi: 10.5281/ZENODO.6476040.

[32] A. Head *et al.*, "Augmenting Scientific Papers with Just-in-Time, Position-Sensitive Definitions of Terms and Symbols," in *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, in CHI '21. 2021, pp. 1–18. doi: 10.1145/3411764.3445648.

[33] B. Shneiderman, "The eyes have it: a task by data type taxonomy for information visualizations," in *Proceedings 1996 IEEE Symposium on Visual Languages*, in VL-96. 1996. doi: 10.1109/vl.1996.545307.

[34] A. Barnett and Z. Doubleday, “The growth of acronyms in the scientific literature,” *eLife*, vol. 9, 2020, doi: 10.7554/elife.60080.

[35] *European Science Editing*, vol. 45, no. 1, 2019, doi: 10.20316/ese.2019.45.18018.

[36] T. Kluyver *et al.*, “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., Netherlands, 2016, pp. 87–90. [Online]. Available: https://eprints.soton.ac.uk/403913/

[37] M. Dunn, “Single-source publishing with XML,” *IT Professional*, vol. 5, no. 1, pp. 51–54, 2003, doi: 10.1109/mitp.2003.1176491.

[38] R. Cockett, L. J. Heagy, and D. W. Oldenburg, “Pixels and their neighbors: Finite volume,” *The Leading Edge*, vol. 35, no. 8, pp. 703–706, 2016, doi: 10.1190/tle35080703.1.

Open Access | https://doi.org/10.25080/FVWM4182

SciPy 2024

July 8 - July 14, 2024

Proceedings of the 23rd Python in Science Conference
ISSN: 2575-9752

# geosnap: The Geospatial Neighborhood Analysis Package

## Open Tools for Urban, Regional, and Neighborhood Science

**Elijah Knaap**[1,2], and **Sergio Rey**[1,2]

[1]San Diego State University, [2]Center for Open Geographical Science

**Abstract**

Understanding neighborhood context is critical for social science research, public policy analysis, and urban planning. The social meaning, formal definition, and formal operationalization of "neighborhood" depends on the study or application, however, so neighborhood analysis and modeling requires both flexibility and adherence to a formal pipeline. Maintaining that balance is challenging for a variety of reasons. To address those challenges, `geosnap`, the Geospatial Neighborhood Analysis Package provides a suite of tools for exploring, modeling, and visualizing the social context and spatial extent of neighborhoods and regions over time. It brings together state-of-the-art techniques from geodemographics, regionalization, spatial data science, and segregation analysis to support social science research, public policy analysis, and urban planning. It provides a simple interface tailored to formal analysis of spatiotemporal urban data.

**Keywords** spatial-analysis, geodemographics, boundary-harmonization, neighborhood-dynamics, segregation

**Published** Jul 10, 2024

**Correspondence to**
Elijah Knaap
eknaap@sdsu.edu

## 1. Introduction

Quantitative research focusing on cities, neighborhoods, and regions is in high demand. On the practical side, city planners, Non-Governmental Organizations (NGOs), and commercial businesses all have increasing access to georeferenced datasets and spatial analyses can turn these data into more efficient, equitable, and profitable, decision-making [1], [2], [3], [4], [5], [6], [7]. On the academic side, many of the era's pressing problems are spatial or urban in form, requiring bespoke statistical methods and simulation techniques to produce accurate inferences. Thus, while data science teams in industries across the planet race to conquer geospatial data science few practitioners have expertise in the appropriate data sources, modeling frameworks, or data management techniques necessary for wrangling and synthesizing urban data.

To address these challenges we introduce `geosnap`, the geospatial neighborhood analysis package, which is a unique Python package sitting at the intersection of spatial science and practical application. In doing so, it provides a bridge between formal spatial analysis and the applied fields of neighborhood change, access to opportunity, and the social determinants of health. The package provides fast and efficient access to hundreds of socioeconomic, infrastructure, and environmental quality indicators that allow researchers to move from zero data to an informative model in a few short lines of code.

It is organized into layers for data acquisition, analysis, and visualization, and includes tools for harmonizing disparate datasets into consistent geographic boundaries, creating "geodemographic typologies" that summarize and predict multidimensional segregation over time, and network analysis tools that generate rapid (locally computed) service areas and travel isochrones. We expect the tool will be immediately useful for any analyst studying urban areas.

## 2. The Geospatial Neighborhood Analysis Package (geosnap)

`geosnap` provides a suite of tools for exploring, modeling, and visualizing the social context and spatial extent of neighborhoods and regions over time. It brings together state-of-the-art techniques from geodemographics, regionalization, spatial data science, and segregation measurement to support social science research, public policy analysis, and urban planning. It provides a simple interface tailored to formal analysis of spatiotemporal urban data, and its primary features include

- Quick access to a large database of commonly used neighborhood indicators from

U.S. providers including the U.S. Census, the Environmental Protection Agency (EPA), Longitudinal Household-Employment Dynamics (LEHD), the National Center for Education Statistics (NCES), and National Land Cover Database (NLCD), streamed efficiently from the cloud or stored for repeated use.

- Fast, efficient tooling for standardizing data from multiple time periods into a

shared geographic representation appropriate for spatiotemporal analysis

- Analytical methods for understanding sociospatial structure in neighborhoods, cities,

and regions, using unsupervised ML from scikit-learn and spatial optimization from PySAL

- Classic and spatial analytic methods for diagnosing model fit, and locating (spatial)

statistical outliers

- Novel techniques for understanding the evolution of neighborhoods over time, including

identifying hotspots of local neighborhood change, as well as modeling and simulating neighborhood conditions into the future

- Unique tools for measuring segregation dynamics over space and time
- Bespoke visualization methods for understanding travel commute-sheds, business

service-areas, and neighborhood change over time

The package is organized into four layers providing different aspects of functionality, namely `io` for data ingestion and storage, `analyze` for analysis and modeling functions, `harmonize` for geographic boundary harmonization, and `viz` for visualization.

```python
import geopandas as gpd
import matplotlib.pyplot as plt
import pandas as pd
from geosnap import DataStore
from geosnap import analyze as gaz
from geosnap import io as gio
from geosnap import visualize as gvz
from geosnap.harmonize import harmonize
```

### *2.1. Data*

Neighborhoods, cities, and regions can be characterized by a variety of data. Researchers use the social, environmental, institutional, and economic conditions of the area to understand change in these places and the people who inhabit them. The geosnap package tries to make the process of collecting and wrangling these various publicly available datasets simpler, faster, and more reproducible by repackaging the most commonly used into modern formats and serving (those we are allowed to) over fast web protocols. This makes it much faster to conduct, iterate, and share analyses for better urban research and public policy than the typical method of gathering repeated datasets from disparate providers in old formats. Of course, these datasets are generally just a starting place, designed to be used alongside additional information unique to each study region.

Toward that end, many of the datasets used in social science and public policy research in the U.S. are drawn from the same set of resources, like the Census, the Environmental Protection Agency (EPA), the Bureau of Labor Statistics (BLS), or the National Center for Education Statistics (NCES), to name a few. As researchers, we found ourselves writing the same code to download the same data repeatedly. That works in some cases, but it is also cumbersome, and can be extremely slow for even medium-sized datasets (tables with roughly 75 thousand rows and 400 columns representing Census Tracts in the USA can take several hours or even days to download from Census servers). While there are nice tools like cenpy or pyrgris, these tools cannot overcome a basic limitation of many government data providers, namely that they use old technology, e.g., File Transfer Protocol (FTP) and outdated file formats (like shapefiles).

Thus, rather than repetitively querying these slow servers, geosnap takes the position that it is preferable to store repeatedly-used datasets in highly-performant formats, because they are usually small and fast enough to store on disk. When not on disk, it makes sense to stream these datasets over S3, where they can be read very quickly (and directly). For that reason, geosnap maintains a public S3 bucket, thanks to the Amazon Open Data Registry, and uses quilt to manage data under the hood. A national-level dataset of more than 200 Census variables at the blockgroup scale (n=220333), *including geometries* can be downloaded in under two minutes, requires only 921MB of disk space, and can be read into GeoPandas in under four seconds. A comparable dataset stored on the Census servers takes more than an hour to download from the Census FTP servers in compressed format, requires more than 15GB of uncompressed disk space, and is read into a GeoDataFrame in 6.48 seconds, including only the geometries.

The `DataStore` class is a quick way to access a large database of social, economic, and environmental variables tabulated at various geographic levels. These data are not required to use the package, and `geosnap` is capable of conducting analysis anywhere in the globe, but these datasets are used so commonly in U.S.-centric research that the developers of `geosnap` maintain this database as an additional layer.

```
datasets = DataStore()
```

### 2.1.1. *Demographics:*

Over the last decade, one of the most useful resources for understanding socioeconomic changes in U.S. neighborhoods over time has been the Longitudinal Tract Database (LTDB), which is used in studies of neighborhood change [8]. One of the most recognized benefits of this dataset is that it standardizes boundaries for census geographic units over time, providing a consistent set of units for time-series analysis. A benefit of these data is the ease with which researchers have access to hundreds of useful intermediate variables computed from raw Census data (e.g. population rates by race and age). Unfortunately, the LTDB data is only available for a subset of Census releases (and only at the tract level), so geosnap includes tooling that computes the same variable set as LTDB, and it provides access to those data for every release of the 5-year ACS at both the tract and blockgroup levels (variable permitting). Note: following the Census convention, the 5-year releases are labelled by the terminal year.

Unlike LTDB or the National Historical Geographic Information System (NHGIS), these datasets are created using code that collects raw data from the Census FTP servers, computes intermediate variables according to the codebook, and saves the original geometries–and is fully open and reproducible. This means that all assumptions are exposed, and the full data processing pipeline is visible to any user or contributor to review, update, or send corrections. Geosnap's approach to data provenance reflects our view that open and transparent analysis is always preferable to black boxes, and is a much better way to promote scientific discovery. Further, by formalizing the pipeline using code, the tooling is re-run to generate new datasets each time ACS or decennial Census data is released.

To load a dataset, e.g. 2010 tract-level census data, call it as a method to return a geodataframe. Geometries are available for all of the commonly used administrative units, many of which have multiple time periods available:

- metropolitan statistical areas (MSAs)
- states
- counties
- tracts
- blockgroups
- blocks

Datasets can be queried and subset using an appropriate Federal Information Processing Series (FIPS) Code, which uniquely identifies every federally defined geographic unit in the U.S.

```
# blockgroups in San Diego county from the 2014-2018 American Community Survey
# (San Diego county FIPS is 06073)
sd = gio.get_acs(datasets, year=2018, county_fips=['06073'])
```

There is also a table that maps counties to their constituent Metropolitan Statistical Areas (MSAs). Note that blockgroups are not exposed directly, but are accessible as part of the ACS and the Environmental Protection Agency's Environmental Justic Screening Tool (EJSCREEN) data. If the dataset exists in the `DataStore` path, it will be loaded from disk, otherwise it will be streamed from S3 (provided an internet connection is available). The geosnap `io` module (aliased here as `gio`) has a set of functions for creating datasets for different geographic regions and different time scales, as well as functions that store remote datasets to disk for repeated use.

### 2.1.2. *Environment:*

The Environmental Protection Agency (EPA)'s envrironmental justice screening tool (EJSCREEN) is a national dataset that provides a wealth of environmental (and some demographic) data at a blockgroup level. For a full list of indicators and their metadata, see this EPA page. This dataset includes important variables like air toxics cancer risk, ozone concentration in the air, particulate matter, and proximity to superfund sites.

### 2.1.3. *Education:*

Geosnap provides access to two related educational data sources. The National Center for Education Statistics (NCES) provides geographic data for schools, school districts and (some time periods) of school attendance boundaries. The Stanford Education Data Archive (SEDA) is a collection of standardized achievement data, available at multiple levels (from state to district to school) with nearly complete national coverage. These data were released by researchers at Stanford to "help us—scholars, policymakers, educators, parents—learn how to improve educational opportunity for all children," and are an incredible resource for social science research. For more information on how to use the SEDA data, see their website, or the Educational Opportunity Project. These can be combined quickly to visualize or conduct further spatial analyses of educational achievement at multiple geographic scales.

### 2.1.4. *Employment:*

The LEHD Origin-Destination Employment Statistics (LODES) data contain job counts tabulated for home and workplace census block. These data are broken down by industrial sector and a handful of demographic characteristics like race and education. LODES also provides flow matrices that show the origin and destination blockgroups, and the number of commuters who travel between the two.

## *2.2. Harmonization*

A common issue in spatial analysis is that geographic tabulation units can (and often do) change over time. For instance the U.S. census boundary polygons like tracts and blocks are re-drawn with each census to accomodate population growth and change. That means it's impossible to see if a given tract has changed over time because the label assigned to each tract means something different in each time period. To help solve that problem `geosnap` leverages the PySAL `tobler` package to harmonize spatial boundaries over time. We do so by calling the `harmonize()` function.

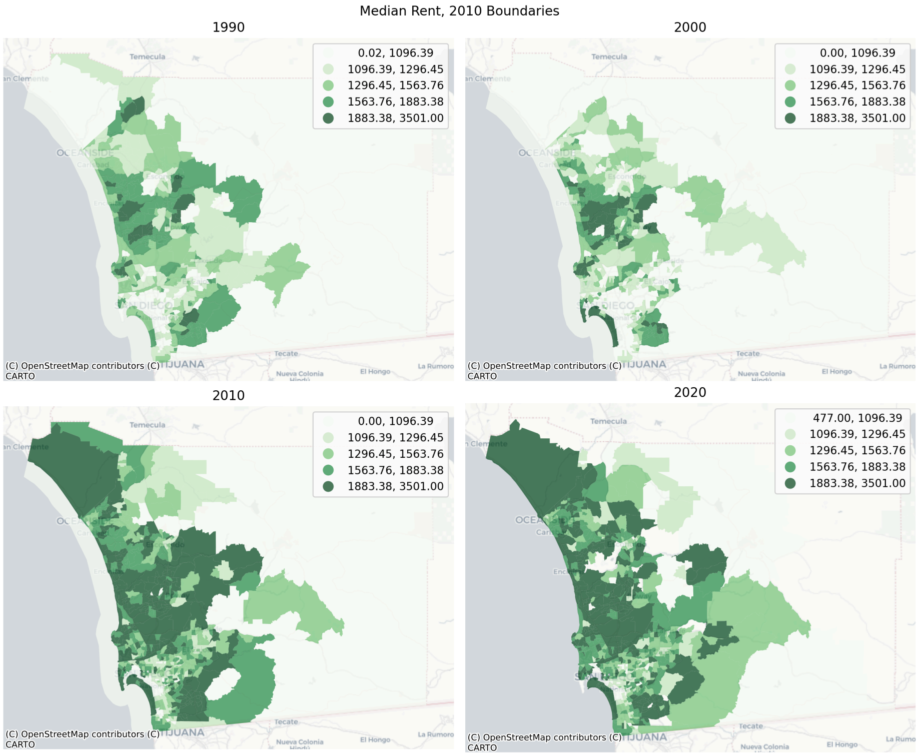
The simplest way to harmonize boundaries is to use areal interpolation, meaning we use the area of overlap between consecutive years to create a weighted sum of intersecting polygons. This approach assumes each polygon has constant density of each attribute so its most useful when the polygons are small and homogenous.

When harmonizing boundaries over time, we need to distinguish between intensive and extensive variables because each needs to be handled differently during the interpolation process. An extensive variable is one whose "magnitude is additive for subsystems, where "the system" this context, refers to the collection of geographic units (which serve as statistical sampling units), and each subsystem is a single geographic unit. Thus, geographically-extensive attributes include those such as population counts, land area, or crop yield, and geographically-intensive variables typically include ratio or interval functions of extensive variables, e.g. density (total population *per unit of land*), average income (total income *per person*), or the share of a particular population subcategory (total population in category *per unit of population*). Median income is a statistic (so intensive), whereas total population is a count (extensive), so we make sure to pass each to the appropriate list.

To standardize the San Diego Census data into consistent boundaries (specifically those defined in 2020), we can use the following code, focusing on median contract rent and the share of Hispanic/Latino population in each unit:

### 2.2.1. *Harmonizing to a Common Census Boundary:*

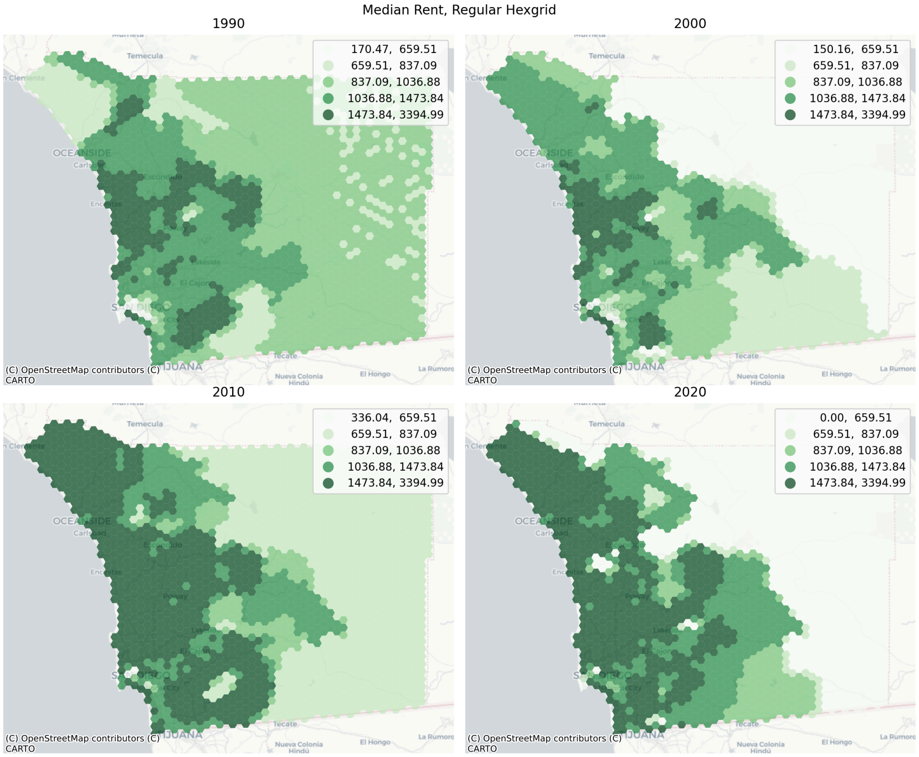
```python
sd_harm20 = harmonize(
    sd,
    intensive_variables=["median_contract_rent", "p_hispanic_persons"],
    extensive_variables=["n_total_pop"],
    weights_method="area",
    target_year=2020,
)
gvz.plot_timeseries(
    sd_harm20,
    "median_contract_rent",
    title="Median Rent, 2010 Boundaries",
    cmap="Greens",
    dpi=200,
    figsize=(12, 10),
    nrows=2,
    ncols=2,
)
plt.tight_layout()
```



### 2.2.2. *Harmonizing to an Alternative Polygon:*

In some cases, it can be useful to discard the original boundaries altogether and instead harmonize all time periods to a consistent geographic unit defined elsewhere (like a school district, or congressional district, or a regular hexgrid).

```python
from tobler.util import h3fy
# create a hexgrid that covers the surface of the san diego dataframe
sd_hex = h3fy(sd[sd.year == 2010], resolution=7)
# interpolate the census data (source) to the hexgrid (target)
sd_hex_interp = harmonize(
    sd, target_gdf=sd_hex, intensive_variables=["median_contract_rent"]
)
# plot the result
gvz.plot_timeseries(
    sd_hex_interp,
    "median_contract_rent",
    title="Median Rent, Regular Hexgrid",
    dpi=200,
    cmap="Greens",
    figsize=(12, 10),
    nrows=2,
    ncols=2,
)
plt.tight_layout()
```



After the harmonization process, each polygon represents a distinct portion of the study area whose boundaries are unchanging, and can be conceived as a discrete unit in temporal analyses. That is, unlike the original decennial census data whose polygons are different in each time period, the harmonized data can be used directly in studies of neighborhood change, for example those using Markov-based methods [9], [10], [11], [12], [13] or sequence-analysis methods [3], [14] (both of which are built into `geosnap`).

## 2.3. *Analysis*

### 2.3.1. *Geodemographic Clustering & Regionalization:*

Geodemographic analysis, which includes the application of unsupervised learning algorithms to demographic and socioeconomic data, is a widely-used technique that falls under the broad umbrella of "spatial data science" [15], [16], [17], [18], [19], [20]. Technically there is no formal *spatial analysis* in traditional geodemographics, however given its emphasis on geographic units of analysis (and subsequent mapping of the results) it is often viewed as a first (if not requisite step) in exploratory analyses of a particular study area.

The intellectual roots of geodemographics extend from analytical sociology and classic studies from factorial ecology [21], [22], [23], [24], [25] and social area analysis [26], [27], [28], [29], [30]. Today, geodemographic analysis is routinely applied in academic studies of neighborhood segregation and neighborhood change, and used extremely frequently in industry, particularly marketing where products like tapestry and mosaic are sold for their predictive power. Whereas social scientists often look at the resulting map of neighborhood types and ask how these patterns came to be, practitioners often look at the map and ask how they can use the patterns to inform better strategic decisions.

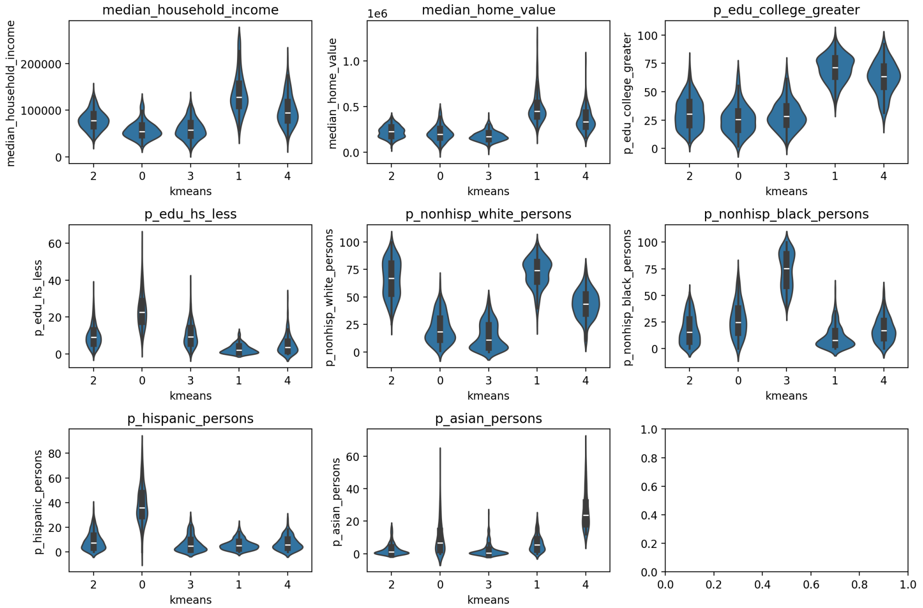
In urban social science, a primary goal is often to study the social composition of neighborhoods in a region (e.g. the racial and ethnic makeup, the population's age distribution and fertility rates, or the socioeconomic disparities or commercial and industrial mix, to name only a few), and understand whether they have changed over time (and where) and whether these neighborhood types are consistent over time and across places. That requires a common pipeline of collecting the same variable sets, standardizing them (often within the same time period, so they can be pooled with other time periods) then clustering the entire long-form dataset followed by further analysis and visualization of the results. Most often, this process happens repeatedly using different combinations of variables or different algorithms or cluster outputs (and in different places at different times). Geosnap provides a set of tools to simplify this pipeline making it trivial to construct, interrogate, and visualize a large swath of models (e.g. using different subsets of variables) for rapid exploration.

To create a simple geodemographic typology, use the `cluster` function from geosnap's `analyze` module and pass a geodataframe, a set of columns to include in the cluster analysis, the algorithm to use and the number of clusters to fit (though some algorithms require different arguments and/or discover the number of clusters endogenously. By default, this will z-standardize all the input columns, drop observations with missing values for input columns, realign the geometries for the input data, and return a geodataframe with the cluster labels as a new column (named after the clustering algorithm). Using the `return_model` argument specifies that the underlying object from scikit-learn or PySAL's `spopt` package is returned for further diagnostic inspection.

```python
# collect data for the atlanta MSA (12060) at the tract level
atl = gio.get_acs(datasets, msa_fips="12060", years=2021, level="tract")
# create a neighborhood typology with 5 clusters via kmeans
atl_kmeans, atl_k_model = gaz.cluster(
    gdf=atl, method="kmeans", n_clusters=5, columns=columns, return_model=True
)
```

To understand what the clusters are identifying, another data visualization technique is useful. Here, we rely on the Tufte-ian principle of "small multiples" (i.e. where each bivariate visualization is repeatedly displayed, conditional on a third variable [31], and create a set of violin plots that show how each variable is distributed across each of the clusters. Each of the inset boxes shows a different variable, with cluster assignments on the x-axis, and the variable itself on the y-axis The boxplot in the center shows the median and interquartile range (IQR), and the "body" of the "violin" is a kernel-density estimate reflected across the x-axis. In short, the fat parts of the violin show where the bulk of the observations are located, and the skinny "necks" show the long tails.

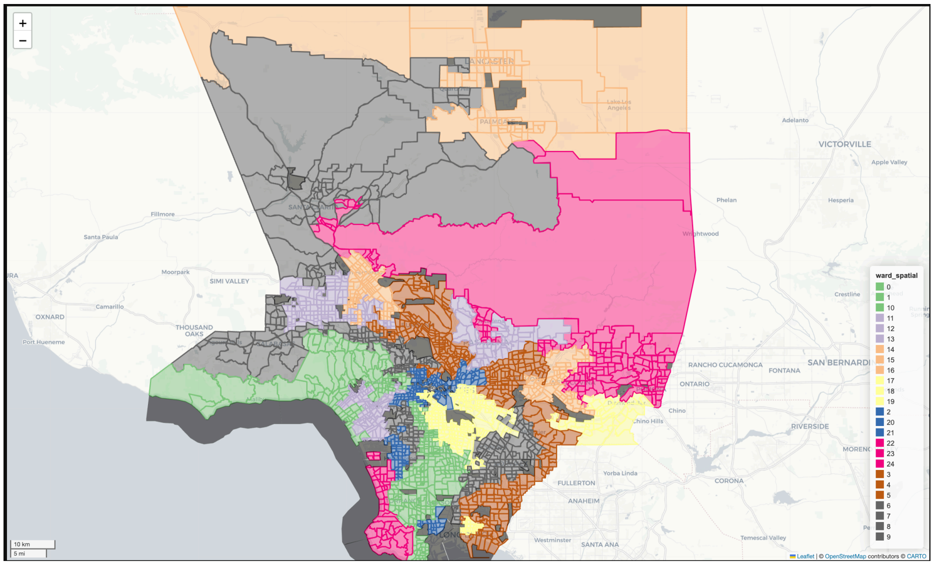
```
# plot the distribution of each input variable grouped by cluster
gvz.plot_violins_by_cluster(atl_kmeans, columns, cluster_col="kmeans")
```



Regionalization is a similar exercise that attempts to define groups of homogenous tracts, but also enforces a contiguity constraint so that similar geographic units must also be proximate to one another. As with geodemographic clustering, when carrying out a regionalization exercise, we are searching for groups of observations (census tracts in this case) which are similar in socioeconomic and demographic composition. If the goal of geodemographics is to identify neighborhood *types* that could exist anywhere in the region, the goal of regionalization is to identify *specific neighborhoods* that exist at a distinct place in the region.

Following that concept, we can use constrained clustering to develop an empirical version of geographically bounded neighborhoods, where the neighborhoods are defined by internal social homogeneity. This is similar to the historic and well-defined neighborhood zones in places like Chicago and Pittsburgh. Similar to the cluster analysis above, the following cell creates a regionalization solution, grouping observations in Los Angeles into categories that are both socially and spatially distinct using a spatially-constrained hierarchical clustering algorithm (with Ward's linkage).

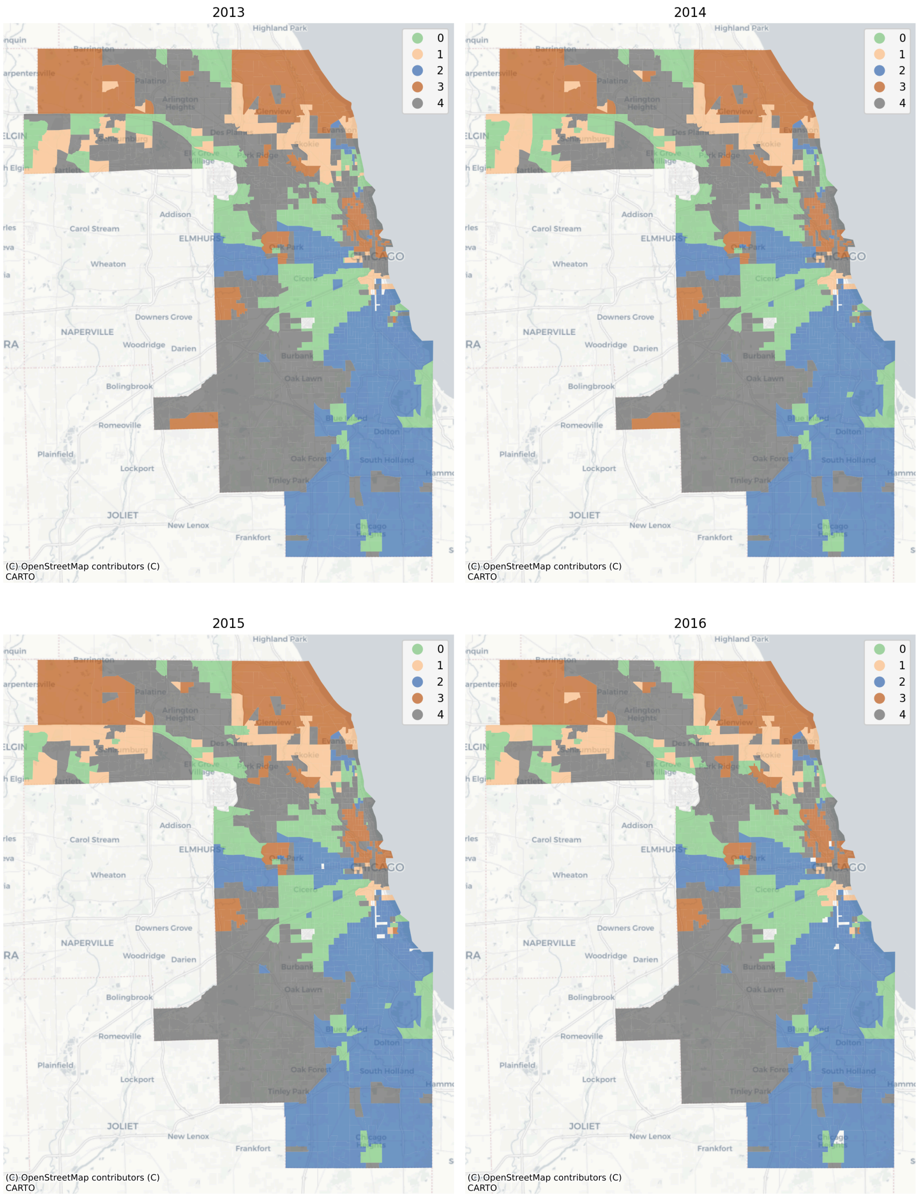
```
# collect data for Los Angeles county
la = gio.get_acs(datasets, county_fips="06037", years=2021, level="tract")
# generate a regionalization using constrained hierarchical clustering
# return both the dataframe and the ModelResults class
la_ward_reg, la_ward_model = gaz.regionalize(
    gdf=la,
    method="ward_spatial",
    n_clusters=25,
    columns=columns,
    return_model=True,
    spatial_weights="queen",
)
# generate an interactive plot showing the regionalization solution
la_ward_reg[columns + ["geometry", "ward_spatial"]].explore(
    "ward_spatial", categorical=True, cmap="Accent", tiles="CartoDB Positron"
)
```



### 2.3.2. *Transitional Dynamics:*

With geosnap, it's possible to look at temporal geodemographics without writing much code. The package provides tools for scaling each dataset within its own time period, adjusting currency values for inflation, and ensuring that times, variables, and geometries stay aligned properly. Together those tools make it easy to explore how different portions of the region transition into different neighborhood types over time, and if desired, model the evolution of neighborhood change as a spatial Markov process. The following cell creates a neighborhood typology in Chicago using four mutually exclusive race and ethnicity categories (defined by the U.S. Census) along with median home values and household incomes, which yields a set of cluster labels that change over time for each geographic unit. The `plot_timeseries` function arranges the maps of neighborhood category labels in sequential order.

```
# define a set of socioeconomic and demographic variables
columns = ['median_household_income', 'median_home_value', 'p_asian_persons', 'p_hispanic_persons',
'p_nonhisp_black_persons', 'p_nonhisp_white_persons']
# create a geodemographic typology using the Chicago data
chicago_ward = cluster(gdf=chicago, columns=columns, method='ward', n_clusters=5)
# plot the result
plot_timeseries(chicago_ward, 'ward', categorical=True, nrows=2, ncols=2, figsize=(12,16))
plt.tight_layout()
```



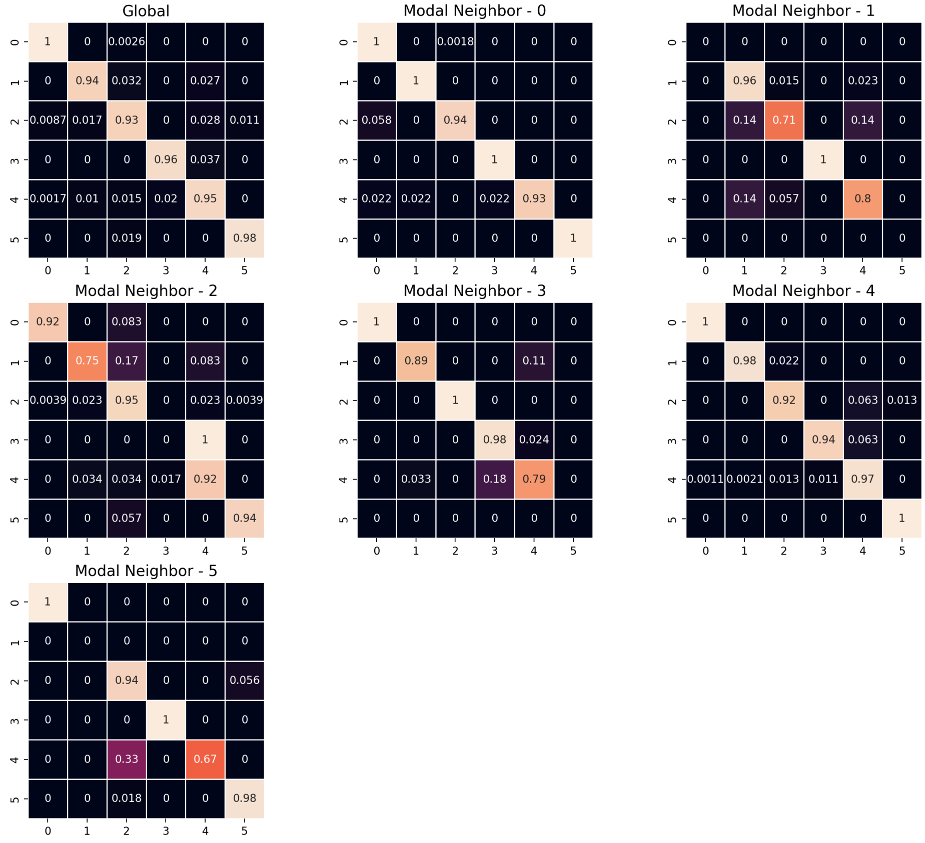
In this example, the vast majority of tracts are assigned to the same geodemographic type in each time period, but some transition into different types over time. The ones that do transition tend to be those on the edges of large contiguous groups (i.e. change tends to happen along the periphery and move inward, implying a certain kind of spatial dynamic). Following, we can also use the sequence of labels to create a spatial Markov transition model. These models examine how often one neighborhood type transitions into another type, then how these transition rates change under different conditions of spatial context.

Here, a key question of interest concerns whether there is spatial patterning in the observed neighborhood transition. If neighborhood transitions are influenced by what occurs nearby, then it suggests the potential influence of spatial spillovers. Although there is nothing 'natural' about it, this phenomenon would be akin to classic sociological models of neighborhood change from the 1920s [32], [33], [34]. Further, if there is evidence that space matters for transitions, then any attempt to understand neighborhood processes in this region *should also consider the importance of spatial interaction.*

A natural way to understand the transition matrix between two neighborhood types is to plot the observed transition rates as a heatmap. The rows of the matrix define the origin neighborhoods (i.e. the neighborhood type at $i$) and the columns define the "destination" neighborhood type (the neighborhood type at $j$), for each of the types and the values of the cells are the fraction of transitions between these two types over all time periods:

$$\hat{p}_{i,j} = \frac{n_{i,j}}{\sum_{k=1}^{k} n_{i,k}} \tag{1}$$

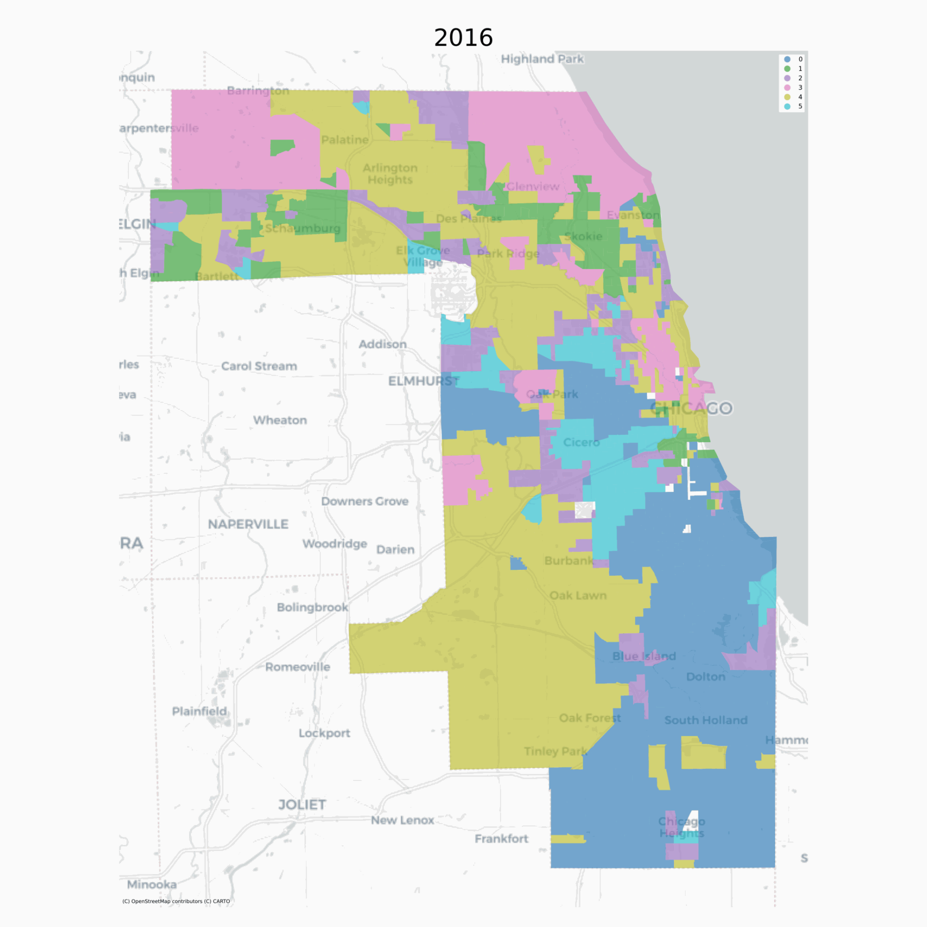
```
# plot the global and conditional transition matrices
from geosnap.visualize import plot_transition_matrix
plot_transition_matrix(chicago_ward, cluster_col='ward')
```



The "Global" heatmap in the upper left shows the overall transition rates between all pairs of neighborhood clusters, and the successive heatmaps show the transition rates conditional on different spatial contexts. The "Modal Neighbor 0" graph shows how the transition rates change when the most common unit surrounding the focal unit is Type 0. The strong diagonal across all heatmaps describes the likelihood of stability; that is, for any neighborhood type, the most common transition is remaining in its same type.

The fact that the transition matrices are not the same provides superficial evidence that conditional transition rates may differ. A formal significance test can be conducted using the returned `ModelResults` class which stores the fitted model instance from scikit-learn or `spopt` and includes some additional diagnostic convenience methods (which shows significantly different transition dynamics in this example).

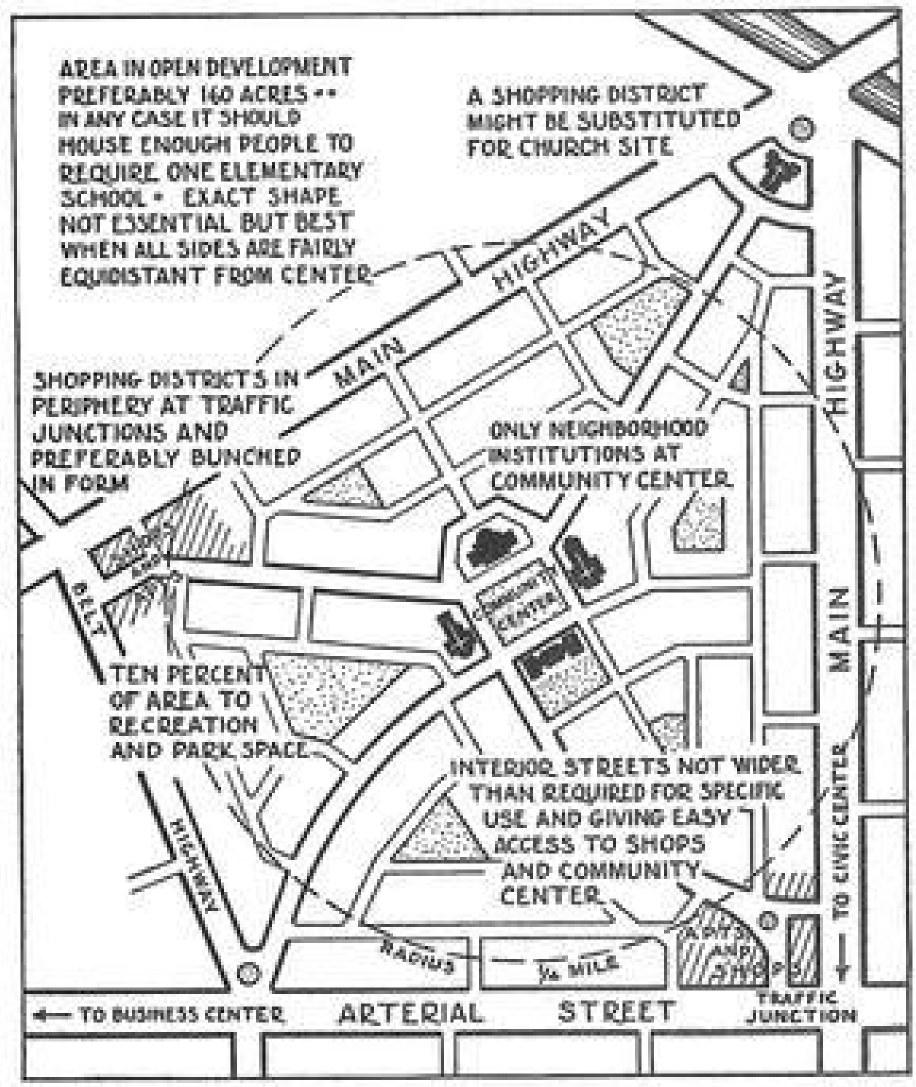
Further, the transition rates can be used to simulate future conditions (note this feature is still experimental). To simulate labels into the future, we determine the spatial context for each unit (determined by its local neighborhood via a PySAL Graph) and draw a new label from the conditional transition matrix implied by the units local neighborhood. This amounts to a simple cellular automata model on an irregular lattice, where there is only a single (stochastic) transition rule, whose transition probabilities update each round.



## 2.3.3. *Travel Isochrones:*

As a package focused on “neighborhoods”, much of the functionality in geosnap is organized around the concept of ‘endogenous’ neighborhoods. That is, it takes a classical perspective on neighborhood formation: a “neighborhood” is defined loosely by its social composition, and the dynamics of residential mobility mean that these neighborhoods can grow, shrink, or transition entirely.

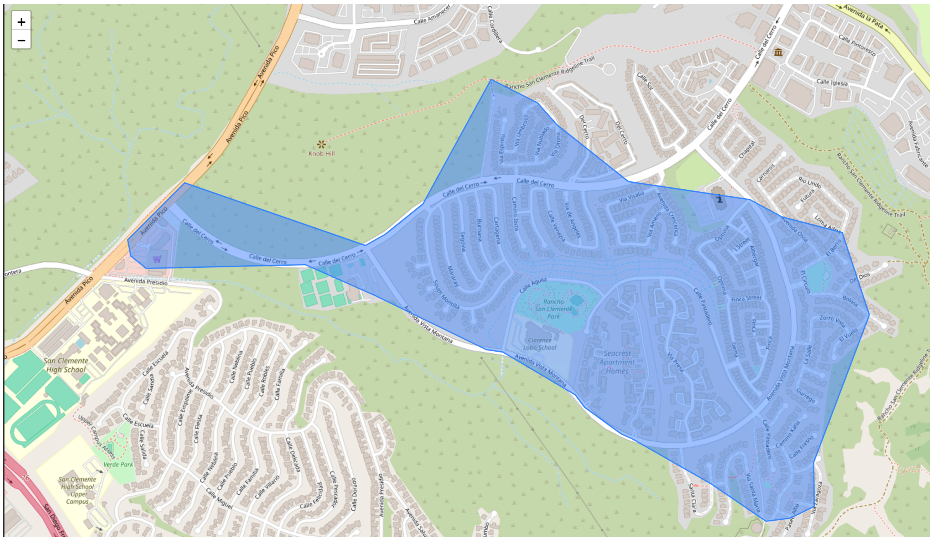
But two alternative concepts of “neighborhood” are also worth considering. The first, posited by social scientists, is that each person or household can be conceptualized as residing in *its own* neighborhood which extends outward from the person’s household until some threshold distance. This neighborhood represents the boundary inside which we might expect some social interaction to occur with one's "neighbors". The second is a normative concept advocated by architects, urban designers, and planners (arguably still the goal for new urbanists): that a neighborhood is a discrete pocket of infrastructure organized as a small, self-contained economic unit. A common shorthand today is the "20 minute neighborhood" [35].


AREA IN OPEN DEVELOPMENT PREFERABLY 160 ACRES •• IN ANY CASE IT SHOULD HOUSE ENOUGH PEOPLE TO REQUIRE ONE ELEMENTARY SCHOOL • EXACT SHAPE NOT ESSENTIAL BUT BEST WHEN ALL SIDES ARE FAIRLY EQUIDISTANT FROM CENTER
A SHOPPING DISTRICT MIGHT BE SUBSTITUTED FOR CHURCH SITE
HIGHWAY
MAIN
SHOPPING DISTRICTS IN PERIPHERY AT TRAFFIC JUNCTIONS AND PREFERABLY BUNCHED IN FORM
HIGHWAY
ONLY NEIGHBORHOOD INSTITUTIONS AT COMMUNITY CENTER
BELT
COMMUNITY CENTER
MAIN
TEN PERCENT OF AREA TO RECREATION AND PARK SPACE
INTERIOR STREETS NOT WIDER THAN REQUIRED FOR SPECIFIC USE AND GIVING EASY ACCESS TO SHOPS AND COMMUNITY CENTER
HIGHWAY
TO CIVIC CENTER
RADIUS ¼ MILE
APTS AND SHOPS
← TO BUSINESS CENTER
ARTERIAL STREET
TRAFFIC JUNCTION

The difference between these two perspectives is what defines the origin of the neighborhood (the "town center" or the person's home), and whether "neighborhood" is universal to all neighbors or unique for each resident. Both of them rely abstractly on the concept of an isochrone: the set of destinations accessible within a specified travel budget. Defining an isochrone is a network routing problem, but its spatial representation is usually depicted as a polygon that encloses the set of reachable locations. That polygon is also sometimes called a walk-shed (or bike/transit/commute etc. "shed", depending on the particular mode of travel). For the person at the center of the isochrone, whose "neighborhood" the isochrone represents, the polygon is sometimes conceived as an "egohood" or "bespoke neighborhood" [5], [36], [37].

Urban transportation planning often adopts the term "isochrone" referring to a line of equal time that demarcates an extent reachable within a certain travel budget (e.g. the "20 minute neighborhood" can be depicted as a 20-minute isochrone of equal travel time from a given origin. Travel isochrones are a useful method for visualizing these bespoke concepts of the "neighborhood", but they can be computationally difficult to create because they require rapid calculation of shortest-path distances in large travel networks. Fortunately, a travel network is a special kind of network graph, and routing through the graph can be increased dramatically using pre-processing techniques known as contraction hierarchies that eliminate inconsequential nodes from the routing problem [38]. `geosnap` uses the `pandana` library [39] which provides a Python interface to the contraction hierarchies routine and works extremely fast on large travel networks. This means geosnap can construct isochrones from massive network datasets in only a few seconds, thanks to `pandana`.

```python
# download an openstreetmap network of the San Diego region
import quilt3 as q3
b = q3.Bucket("s3://spatial-ucr")
b.fetch("osm/metro_networks_8k/41740.h5", "./41740.h5")
# create a (routeable) pandana Network object
sd_network = pdna.Network.from_hdf5("41740.h5")
# select a single intersection as an example
example_origin = 1985327805
# create an isochrone polygon
iso = isochrones_from_id(example_origin, sd_network, threshold=1600 ) # network is expressed in meters
iso.explore()
```



These accessibility isochrones can then be used as primitive units of inquiry in further spatial analysis. For example the isochrones can be conceived as "service areas" that define a reachable threshold accessible from a given provider (e.g., healthcare, social services, or healthy food).

### 2.4. Conclusion

In this paper we introduce the motivation and foundational design principles of the geospatial neighborhood analysis package `geosnap`. Intended for researchers, public policy professionals, urban planners, and spatial market analysts, among others, the package allows for rapid development and exploration of different models of "neighborhood". This includes tools for data ingestion and filtering from commonly-used public datasets in the U.S., as well as tools for boundary harmonization, transition modeling, and travel-time visualization (all of which are applicable anywhere on the globe).

## References

[1] R. X. Cortes, S. Rey, E. Knaap, and L. J. Wolf, "An Open-Source Framework for Non-Spatial and Spatial Segregation Measures: The PySAL Segregation Module," *Journal of Computational Social Science*, vol. 3, no. 1, pp. 135–166, 2020, doi: 10.1007/s42001-019-00059-3.

[2] N. Finio and E. Knaap, "Smart Growth's Misbegotten Legacy: Gentrification," in *Handbook on Smart Growth*, G. Knaap, R. Lewis, A. Chakraborty, and K. June-Friesen, Eds., Edward Elgar Publishing, 2022. doi: 10.4337/9781789904697.00026.

[3] W. Kang, E. Knaap, and S. Rey, "Changes in the Economic Status of Neighbourhoods in US Metropolitan Areas from 1980 to 2010: Stability, Growth and Polarisation," *Urban Studies*, p. 004209802110425, 2021, doi: 10.1177/00420980211042549.

[4] E. Knaap, "The Cartography of Opportunity: Spatial Data Science for Equitable Urban Policy," *Housing Policy Debate*, vol. 27, no. 6, pp. 913–940, 2017, doi: 10.1080/10511482.2017.1331930.

[5] E. Knaap and S. Rey, "Segregated by Design? Street Network Topological Structure and the Measurement of Urban Segregation," *Environment and Planning B: Urban Analytics and City Science*, p. 23998083231197956, 2023, doi: 10.1177/23998083231197956.

[6] S. Rey and E. Knaap, "The Legacy of Redlining: A Spatial Dynamics Perspective," *International Regional Science Review*, vol. 0, no. 0, 2022, doi: 10.1177/01600176221116566.

[7] R. Wei, X. Feng, S. Rey, and E. Knaap, "Reducing Racial Segregation of Public School Districts," *Socio-Economic Planning Sciences*, p. 101415, 2022, doi: 10.1016/j.seps.2022.101415.

[8] J. R. Logan, Z. Xu, and B. J. Stults, "Interpolating U.S. Decennial Census Tract Data from as Early as 1970 to 2010: A Longitudinal Tract Database," *The Professional Geographer*, vol. 66, no. 3, pp. 412–420, 2014, doi: 10.1080/00330124.2014.905156.

[9] M. Agovino, A. Crociata, and P. L. Sacco, "Proximity Effects in Obesity Rates in the US: A Spatial Markov Chains Approach," *Social Science and Medicine*, vol. 220, pp. 301–311, 2019, doi: 10.1016/j.socscimed.2018.11.013.

[10] J. L. Gallo, J. Le Gallo, J. L. Gallo, and J. Le Gallo, "Space-Time Analysis of \vphantomGDP\vphantom Disparities across \vphantomEuropean\vphantom \vphantomRegions\vphantom: A \vphantomMarkov\vphantom Chains Approach," *International Regional Science Review*, vol. 27, no. 2, pp. 138–163, 2004, doi: 10.1177/0160017603262402.

[11] W. Kang and S. J. Rey, "Conditional and Joint Tests for Spatial Effects in Discrete \vphantomMarkov\vphantom Chain Models of Regional Income Convergence," *Annals of Regional Science*, 2018, doi: 10.1007/s00168-017-0859-9.

[12] S. J. Rey, W. Kang, and L. Wolf, "The Properties of Tests for Spatial Effects in Discrete \vphantomMarkov\vphantom Chain Models of Regional Income Distribution Dynamics," *Journal of Geographical Systems*, vol. 18, no. 4, pp. 377–398, 2016, doi: 10.1007/s10109-016-0234-x.

[13] S. Rey, "Rank-Based Markov Chains for Regional Income Distribution Dynamics," *Journal of Geographical Systems*, vol. 16, no. 2, pp. 115–137, 2014, doi: 10.1007/s10109-013-0189-0.

[14] K. O. Lee, R. Smith, and G. Galster, "Neighborhood Trajectories of Low-Income U.S. Households: An Application of Sequence Analysis," *Journal of Urban Affairs*, vol. 39, no. 3, pp. 335–357, 2017, doi: 10.1080/07352166.2016.1251154.

[15] J. Abbas, A. Ojo, and S. Orange, "Geodemographics – a Tool for Health Intelligence?," *Public Health*, vol. 123, no. 1, pp. e35–e39, 2009, doi: 10.1016/j.puhe.2008.10.007.

[16] M. Adnan, P. A. Longley, A. D. Singleton, and C. Brunsdon, "Towards Real-Time Geodemographics: Clustering Algorithm Performance for Large Multidimensional Spatial Databases," *Transactions in GIS*, vol. 14, no. 3, pp. 283–297, 2010, doi: 10.1111/j.1467-9671.2010.01197.x.

[17] T. K. Anderson, "Using Geodemographics to Measure and Explain Social and Environment Differences in Road Traffic Accident Risk," *Environment and Planning A*, vol. 42, no. 9, pp. 2186–2200, 2010, doi: 10.1068/a43157.

[18] S. De Sabbata and P. Liu, "Deep Learning Geodemographics with Autoencoders and Geographic Convolution," *Proceedings of the 22nd AGILE conference on Geographic Information Science*, 2019, [Online]. Available: https://pysal.org/

[19] A. D. Singleton and P. A. Longley, "Creating Open Source Geodemographics: Refining a National Classification of Census Output Areas for Applications in Higher Education," *Papers in Regional Science*, vol. 88, no. 3, pp. 643–666, 2009, doi: 10.1111/j.1435-5957.2008.00197.x.

[20] A. Singleton and P. Longley, "Geodemographics, Visualisation, and Social Networks in Applied Geography," *Applied Geography*, vol. 29, no. 3, pp. 289–298, 2009, doi: 10.1016/j.apgeog.2008.10.006.

[21] B. J. L. Berry, "Introduction: The Logic and Limitations of Comparative Factorial Ecology," *Economic Geography*, vol. 47, no. 4, p. 209, 1971, doi: 10.2307/143204.

[22] A. A. Hunter, "Factorial Ecology: A Critique and Some Suggestions," *Demography*, vol. 9, no. 1, p. 107, 1972, doi: 10.2307/2060548.

[23] M. D. Lebowitz, "A Critical Examination of Factorial Ecology and Social Area Analysis for Epidemiological Research," *Journal of the Arizona Academy of Science*, vol. 12, no. 2, pp. 86–90, 1977, doi: 10.2307/40022220.

[24] E. D. Perle, "Variable Mix and Factor Stability in Urban Ecology," *Geographical Analysis*, vol. 11, no. 4, pp. 410–414, 1979, doi: 10.1111/j.1538-4632.1979.tb00708.x.

[25] P. H. Rees, "The Factorial Ecology of Calcutta," *American Journal of Sociology*, vol. 74, no. 5, pp. 445–491, 1969, doi: 10.1086/224681.

[26] W. Bell and S. Greer, "Social Area Analysis and Its Critics," *The Pacific Sociological Review*, vol. 5, no. 1, pp. 3–9, 1962, doi: 10.2307/1388270.

[27] T. Brindley and J. Raine, "Social Area Analysis and Planning Research," *Urban Studies*, vol. 16, no. 3, pp. 273–289, 1979, doi: 10.1080/713702552.

[28] B. S. R. Green, "Social Area Analysis and Structural Effects," *Sociology*, vol. 5, no. 1, pp. 1–19, 1971, [Online]. Available: https://www.jstor.org/stable/42851010

[29] E. Shevky and W. Bell, *Social Area Analysis; Theory, Illustrative Application and Computational Procedures*. Stanford University Press, 1955.

[30] S. E. Spielman and J.-C. Thill, "Social Area Analysis, Data Mining, and GIS," *Computers, Environment and Urban Systems*, vol. 32, no. 2, pp. 110–122, 2008, doi: 10.1016/j.compenvurbsys.2007.11.004.

[31] E. R. Tufte, *The Visual Display of Quantitative Information*, vol. 2. Graphics press Cheshire, CT, 1983.

[32] R. E. Park, E. W. Burgess, R. D. McKenzie, R. E. Park, and R. D. McKenzie, *The City*. Chicago: University of Chicago Press, 1925. [Online]. Available: http://www.jstor.org/stable/3004850?origin=crossref

[33] R. E. Park, "Succession, an Ecological Concept," *American Sociological Review*, vol. 1, no. 2, p. 171, 1936, doi: 10.2307/2084475.

[34] R. E. Park, *Human Communities; the City and Human Ecology*. 1952. doi: 10.2307/2087814.

[35] A. Calafiore, R. Dunning, A. Nurse, and A. Singleton, "The 20-Minute City: An Equity Analysis of Liverpool City Region," *Transportation Research Part D: Transport and Environment*, vol. 102, p. 103111, 2022, doi: 10.1016/j.trd.2021.103111.

[36] J. R. Hipp and A. Boessen, "Egohoods as Waves Washing Across the City: A New Measure of ``Neighborhoods''," *Criminology*, vol. 51, no. 2, pp. 287–327, 2013, doi: 10.1111/1745-9125.12006.

[37] Y.-A. Kim and J. R. Hipp, *Street Egohood: An Alternative Perspective of Measuring Neighborhood and Spatial Patterns of Crime*, no. 123456789. Springer US, 2019. doi: 10.1007/s10940-019-09410-3.

[38] R. Geisberger, P. Sanders, D. Schultes, and C. Vetter, "Exact Routing in Large Road Networks Using Contraction Hierarchies," *Transportation Science*, vol. 46, no. 3, pp. 388–404, 2012, doi: 10.1287/trsc.1110.0401.

[39] F. Foti, P. Waddell, and D. Luxen, "A Generalized Computational Framework for Accessibility: From the Pedestrian to the Metropolitan Scale," *4th Transportation Research Board Conference on Innovations in Travel Modeling (ITM)*, pp. 1–14, 2012.

Open Access | https://doi.org/10.25080/PDHK7238

# Cyanobacteria detection in small, inland water bodies with CyFi

**Emily Dorne**[1], **Katie Wetstone**[1], **Trista Brophy Cerquera**[2], and **Shobhana Gupta**[2]

[1]DrivenData, [2]NASA

**SciPy 2024**

*July 8 - July 14, 2024*

Proceedings of the 23nd
Python in Science Conference
ISSN: 2575-9752

## Abstract

Harmful algal blooms (HABs) pose major health risks to human and aquatic life. Remote sensing-based methods exist to automatically detect large, slow-moving HABs in the ocean, but fall short for smaller, more dynamic blooms in critical inland water bodies like lakes, reservoirs, and rivers.

CyFi is an open-source Python package that enables detection of cyanobacteria in inland water bodies using 10-30m Sentinel-2 imagery and a computationally efficient tree-based machine learning model. CyFi enables water quality and public health managers to conduct high level assessments of water bodies of interest and identify regions in which to target monitoring and responsive actions to protect public health.

CyFi was developed in three phases. A machine learning competition leveraged the diverse skills and creativity of data science experts to surface promising approaches for cyanobacteria detection from remote sensed data. Subsequent user interviews and model iteration resulted in a deployment-ready open-source package designed to meet user workflow needs and decision-making priorities. This process illustrates a replicable pathway for developing powerful machine learning tools in domain-specific areas.

**Keywords** Cyanobacteria, Sentinel-2, Machine Learning, Harmful Algal Blooms, Remote Sensing

## 1. Introduction

Inland water bodies provide a variety of critical services for both human and aquatic life, including drinking water, recreational and economic opportunities, and marine habitats. Harmful algal blooms (HABs) pose a significant risk to these inland bodies, producing toxins that are poisonous to humans and their pets and threatening marine ecosystems by blocking sunlight and oxygen. Such threats require water quality managers to monitor for the presence of HABs and to make urgent decisions around public health warnings and closures when they are detected.

The most common source of HABs in freshwater environments is cyanobacteria, or blue-green algae [1]. While there are established methods for using satellite imagery to detect cyanobacteria in larger water bodies like oceans, detection in small inland lakes, reservoirs, and rivers remains a challenge. Manual water sampling is accurate, but is too time and resource intensive to perform continuously at scale. Machine learning models, on the other hand, can generate estimates in seconds. Automatic detection enables water managers to better prioritize limited manual sampling resources and can provide a birds-eye view of water conditions across a region [2]. Machine learning is particularly well-suited to this task because indicators of cyanobacteria are visible in free, routinely collected satellite imagery.

**Published** Jul 10, 2024

**Correspondence to**
Emily Dorne
emily@drivendata.org

We present CyFi, short for Cyanobacteria Finder, an open-source Python package that uses satellite imagery and machine learning to estimate cyanobacteria levels in inland water bodies [3]. CyFi helps decision makers protect the public by flagging the highest-risk areas in lakes, reservoirs, and rivers quickly and easily. CyFi represents a significant advancement in environmental monitoring, providing higher-resolution detection capabilities that can pinpoint areas at risk of cyanobacterial contamination. Key strengths of CyFi compared to other tools include:

- Features derived from high-resolution Sentinel-2 satellite data
- A fast and computationally efficient boosted tree machine learning algorithm
- A straightforward command line interface
- A unique training dataset of almost 13,000 cyanobacteria ground measurements across the continental U.S.

This paper presents a detailed examination of the development of CyFi, from its origins in a machine learning competition to an open source package. The methods section explores the setup of the prize competition, the subsequent model experimentation phase that built on winning approaches, and the end user interviews that helped shape CyFi with real-world context. The results section provides insights on the machine learning methods that proved most effective for detecting inland HABs, and details CyFi's underlying methodology, core capabilities, and model performance. Finally, the discussion section reflects the primary ways CyFi can augment human decision making workflows to protect public health and notes areas for future research.

## 2. Motivation

> There are tens of thousands of lakes that matter for recreation and drinking water. Cyanobacterial blooms pose real risks in many of them, and we really don't know when or where they show up, except in the largest lakes.
>
> — Dr. Rick Stumpf, Oceanographer, NOAA, National Centers for Coastal Ocean Science[1]

Harmful algal blooms are a pressing environmental and public health issue, characterized by the rapid and excessive growth of algae in water bodies. These blooms can produce toxins, such as microcystins and anatoxins, that pose severe risks to human health, aquatic ecosystems, and local economies [5].

The toxins released by HABs can contaminate drinking water supplies, leading to acute and chronic health problems for communities [6]. Exposure to these toxins through ingestion, skin contact, or inhalation can result in a variety of acute and chronic health issues, including gastrointestinal illnesses, liver damage, neurological effects, and even death in extreme cases [7].

Ecologically, HABS can create hypoxic (low oxygen) conditions in water bodies, resulting in massive fish kills and the disruption of aquatic food webs [8]. HABs can form dense algal mats that block sunlight, inhibiting the growth of submerged vegetation essential for aquatic habitats. Furthermore, the decomposition of large algal blooms consumes significant amounts of dissolved oxygen, exacerbating oxygen depletion and leading to dead zones where most aquatic life cannot survive.

These ecological impacts can have devastating economic consequences for local industries reliant on water resources, such as fisheries, tourism, and recreation. Beaches and lakeside areas affected by algal blooms often face closures, leading to a loss of revenue. The cost

[1] DrivenData [4]

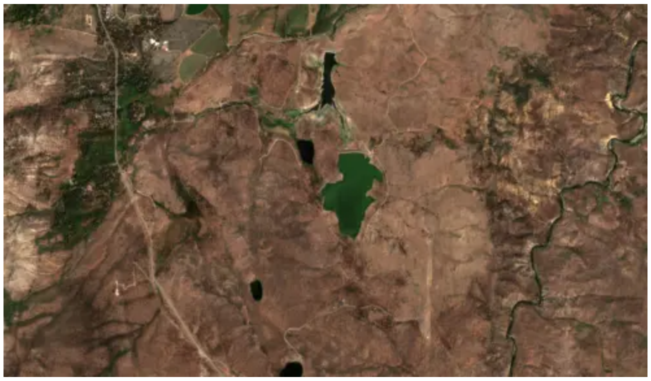


**Figure 1**. *An example of a water body at 10m resolution*

of managing and mitigating the effects of HABs, including water treatment and healthcare expenses, places additional financial burdens on affected communities [9], [10].

Despite the severe consequences of HABs, existing monitoring tools and methods are often insufficient. Traditional approaches, such as manual water sampling and laboratory analysis, are time-consuming, labor-intensive, and provide only localized snapshots of water quality.

Existing satellite-based monitoring tools offer broad coverage but fall short of the spatial resolution needed for small inland water bodies. Most are aimed at monitoring blooms in the ocean, which are larger and slower moving. Many of the leading satellite-based methods for cyanobacteria detection rely on chlorophyll estimates using the Ocean and Land Colour Instrument (OLCI) on Sentinel-3 [11]. However, the coarse 300m resolution of Sentinel-3 only recognizes approximately 5% of inland water bodies in the continental U.S. [12] and therefore is not able to provide the data needed for effective early warning and rapid response to HAB outbreaks in lakes, reservoirs, and rivers. In addition, chlorophyll is an imperfect proxy for cyanobacteria as all types of algae contain chlorophyll-a, including non-harmful blooms [13].

The effects of climate change are likely to increase both the frequency and severity of HABs in inland water bodies. Warmer water temperatures, higher concentrations of carbon dioxide, runoff and coastal upswelling related to extreme weather events, and sea level rise can all contribute to making freshwater nutrients levels and environmental conditions more conducive to the growth of harmful algae [14].

Effectively monitoring inland HABs and protecting public health requires developing new innovative tools that capture a higher spatial resolution, can be run quickly and frequently,

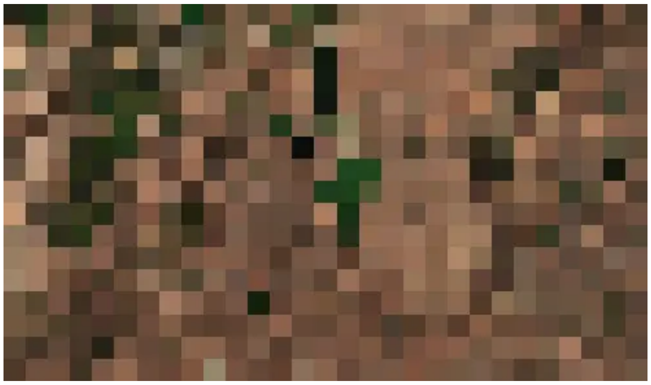


**Figure 2**. *An example of the Figure 1 image at 300m resolution*

and are accessible to decision makers. CyFi aims to fill this gap by incorporating higher-resolution satellite imagery, an efficient tree-based model, and a user-friendly command line interface.

## 3. Methods

### *3.1. Machine learning competition*

The machine learning approach in CyFi was originally developed as part of the Tick Tick Bloom: Harmful Algal Detection Challenge, which ran from December 2022 to February 2023 [4]. Machine learning competitions can harness the power of community-driven innovation and rapidly test a wide variety of possible data sources, model architectures, and features [15], [16], [17]. Tick Tick Bloom was created by DrivenData on behalf of NASA and in collaboration with NOAA, the U.S. Environmental Protection Agency, the U.S. Geological Survey, the U.S. Department of Defense Defense Innovation Unit, Berkeley AI Research, and Microsoft AI for Earth.

In the Tick Tick Bloom challenge, over 1,300 participants competed to detect cyanobacteria blooms in small, inland water bodies using publicly available satellite, climate, and elevation data. Models were trained and evaluated using a set of manually collected water samples that had been analyzed for cyanobacteria density. Labels were sourced from 14 data providers across the U.S., shown in Figure 3. The full dataset containing 23,570 in situ cyanobacteria measurements is publicly available through the SeaBASS data archive [18]. Each observation in the dataset is a unique combination of date, latitude, and longitude.

The labels were divided into train and set sets, where 17,060 train labels were provided to participants and 6,510 test labels were used to evaluate model performance and kept confidential from participants. Lakes in close proximity can experience similar bloom-forming conditions, presenting a risk of leakage. To address this, clustering methods were used to maximize the distance between every train set point and every test set point, decreasing the likelihood that participants could gain insight into any test point density based on the training set. Scikit-learn's DBSCAN algorithm [19], [20] was used to divide all data points into spatial clusters. Each cluster was then randomly assigned to either the train or test dataset, such that no test data point was within 15 kilometers of a train data point.

Participants predicted a severity category for a given sampling point as shown in Table 1. These ranges were informed by EPA and WHO guidelines [21], [22].

Predictions were evaluated using region-averaged root mean squared error. Averaging across regions incentivized models to perform well across the continental U.S., rather than in certain states that were over-represented in the competition dataset (such as California

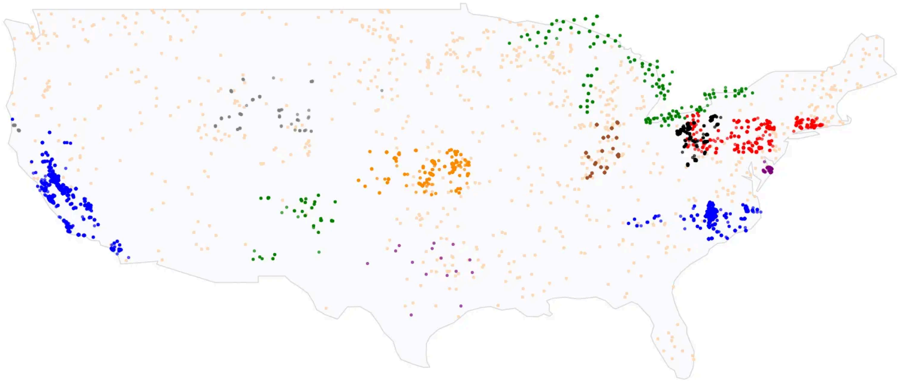


**Figure 3**. *Labeled samples used in the Tick Tick Bloom competition colored by dataset provider.*

**Table 1**. *Severity categories used in the Tick Tick Bloom competition*

| Severity level | Cyanobacteria density range (cells/mL) |
|---|---|
| 1 | <20,000 |
| 2 | 20,000 – <100,000 |
| 3 | 100,000 – <1,000,000 |
| 4 | 1,000,000 – <10,000,000 |
| 5 | ≥10,000,000 |

and North Carolina). Over 900 submissions across 115 teams were made over the course of the competition.

### 3.2. *Carrying forward competition models*

Machine learning competitions are excellent for crowd-sourcing top approaches to complex predictive modeling problems. Over a short period of time, a large community of solvers tests a broad feature space including possible data sources, model architectures, and model features. The result is an open-source github repository with code from the most effective approaches, trained model weights, and write-ups of winning methods.

However, transforming this research code into production code requires significant additional work. There are a few key differences between competition-winning research approaches and deployable code:

1. The competition relies on static data exported and processed once while deployment requires repeated, automatic use with new data.
2. Winning models are relatively unconstrained by the size and cost of their solutions. For ongoing use, efficiency matters.
3. Competition code is validated once with anticipated, clean data and static versions of Python package dependencies. In the real world things break and change; use requires basic robustness, testing and configurability.
4. There is substantial variability in the clarity and organization of competition-winning code. Usable code requires others to be able to understand, maintain, and build on the codebase.

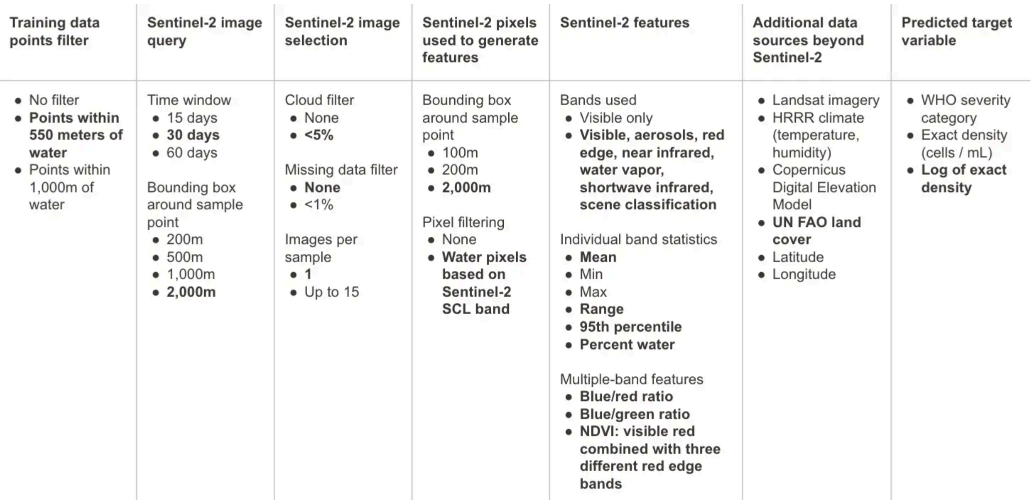
The end goal is regularly generated predictions of cyanobacteria levels surfaced in user-friendly dashboards to water quality managers. To achieve this, the intermediate requirement is a deployment-ready code package. This package is CyFi, a configurable, open-source Python library capable of generating cyanobacteria predictions on new input data.

#### 3.2.1. *Model experimentation:*

CyFi was developed through an additional model experimentation phase, which combined and iterated on the most useful pieces from competition-winning models, and simplified and restructured code to transform it into a runnable pipeline.[2] Additional model testing helped determine which winning approaches were the most robust, accurate, and generalizable outside of the competition setting.

The table below summarizes the matrix of experiments that were conducted. Model experimentation informed key decisions around which data sources were used, how satellite imagery was selected and processed, and which target variable was predicted.

[2] The authors would like to thank Yang Xu, Andrew Wheeler, and Raphael Kimina, the Tick Tick Bloom competition winners whose modeling approaches provided the foundation for CyFi.



| Training data points filter | Sentinel-2 image query | Sentinel-2 image selection | Sentinel-2 pixels used to generate features | Sentinel-2 features | Additional data sources beyond Sentinel-2 | Predicted target variable |
|---|---|---|---|---|---|---|
| ● No filter<br>● **Points within 550 meters of water**<br>● Points within 1,000m of water | Time window<br>● 15 days<br>● **30 days**<br>● 60 days<br><br>Bounding box around sample point<br>● 200m<br>● 500m<br>● 1,000m<br>● **2,000m** | Cloud filter<br>● None<br>● **<5%**<br><br>Missing data filter<br>● **None**<br>● <1%<br><br>Images per sample<br>● **1**<br>● Up to 15 | Bounding box around sample point<br>● 100m<br>● 200m<br>● **2,000m**<br><br>Pixel filtering<br>● None<br>● **Water pixels based on Sentinel-2 SCL band** | Bands used<br>● Visible only<br>● **Visible, aerosols, red edge, near infrared, water vapor, shortwave infrared, scene classification**<br><br>Individual band statistics<br>● **Mean**<br>● Min<br>● Max<br>● **Range**<br>● **95th percentile**<br>● **Percent water**<br><br>Multiple-band features<br>● **Blue/red ratio**<br>● **Blue/green ratio**<br>● **NDVI: visible red combined with three different red edge bands** | ● Landsat imagery<br>● HRRR climate (temperature, humidity)<br>● Copernicus Digital Elevation Model<br>● **UN FAO land cover**<br>● Latitude<br>● Longitude | ● WHO severity category<br>● Exact density (cells / mL)<br>● **Log of exact density** |

**Figure 4**. *Model experimentation summary, with final selections in bold.*

During experimentation the Section 3.1 train-test split was maintained, but samples prior to the launch of Sentinel-2 as well as sampling points more than 550m away from a water body were removed, as outlined in Table 3. This resulted in a final train set size of 8,979 points and a test set size of 4,035 points.

`StratifiedGroupKFold` was used to create five train and validation folds stratified on region (cyfi/pipeline.py) and a model was trained on each fold. Predictions are averaged across the five models. Early stopping based on performance on the validation set was used to prevent overfitting.

Performance on the test set was evaluated based on a combination of root mean squared error, mean absolute error, mean absolute percentage error, and regional root mean squared error, along with manual review and visualizations of predictions. Standard best practices were used to inform hyperparameters tuning for the final model.

### 3.2.2. *User interviews:*

To design a package that optimally addresses on-the-ground user needs, we conducted human-centered design (HCD) interviews with subject matter experts and end users. Interviewees included water quality and public health experts from California, New York, Georgia, Louisiana, Florida, and Michigan. Representatives from these states were selected to understand workflows and priorities, and capture a diversity of geographic locations, number of water bodies in the region, HAB severity, investment in HABs monitoring, and technical sophistication of current approaches. User interviews focused on understanding current water quality decision-making processes, including the data and tools used to support those decisions. Learnings were used to inform the format for surfacing predictions, priorities in model performance, and computational constraints. Table 6 summarizes the core design decisions for CyFi that were rooted in insights from user interviews.

## 4. Results

### *4.1. Competition takeaways*

The overarching goal of the Section 3.1 was to identify the most useful data sources, features, and modeling methods for cyanobacteria estimation in small, inland water bodies. There was particular interest around the use of Sentinel-2 data, which has significantly higher

**Table 2.** *Data sources used by Tick Tick Bloom competition winners*

| | **Landsat**<br>***Satellite*** | **Sentinel 2**<br>***Satellite*** | **HRRR**<br>***Climate data*** | **Copernicus DEM**<br>***Elevation*** | **Metadata**<br>***Time, location*** |
|---|---|---|---|---|---|
| 1st Place | | ✓<br>Color value statistics | ✓<br>Temperature | | ✓<br>Region<br>Location |
| 2nd Place | | ✓<br>Color value statistics | | ✓ | ✓<br>Clustered location |
| 3rd Place | ✓<br>Color value statistics | ✓<br>Color value statistics | ✓<br>Temperature<br>Humidity | | ✓<br>Longitude |

resolution than Sentinel-3 and is more suited to smaller water bodies. However, Sentinel-2 does not contain the bands historically used to derive cyanobacteria estimates.

The competition showed that Sentinel-2 bands contain sufficient information for generating accurate cyanobacteria estimates with machine learning. Below is a summary of which datasets were used by winners.

All winners used Level-2 satellite imagery instead of Level-1, likely because it already includes useful atmospheric corrections. Sentinel-2 data is higher resolution than Landsat, and proved to be more useful in modeling.

All winners also used gradient boosted decision tree models such as LightGBM [23], XGBoost [24], and CatBoost [25]. First place explored training a CNN model but found the coarse resolution of the satellite imagery overly constraining, particularly when using Landsat imagery. Despite deep learning models often being thought of as the go-to for image data, decision tree models can work particularly well with satellite imagery where the task is point estimation rather than segmentation.[3] Decision tree models also have a number of advantages over deep learning models as they boast faster training and inference, do not require a GPU, and provide greater interpretability. This enables more rapid experimentation and iteration in training and supports near real-time inference using limited compute.

### 4.2. *Model experimentation takeaways*

The Section 3.2.1 phase did not explore alternate model architectures given how clearly the competition surfaced the success of a gradient boosted tree model [26]. It did however extensively iterate on other parts of the pipeline. Over 30 configurations were tested to identify the optimal setup for training a robust, generalizable model. Below are the core decisions that resulted from model experimentation and retraining.

#### 4.2.1. *Data decisions:*

One of the risks in a machine learning competition is overfitting to the test set. Competition models may pick up on patterns specific to the competition data, rather than patterns of environmental cyanobacteria conditions that generalize outside of the competition. The experimentation phase worked to identify and remove competition artifacts that would hamper the generalizability of the model in an open source package. For example, all winning solutions used a “longitude” feature in their models, which captured some underlying differences in sampling procedures by the 14 data providers for the competition. For example, data-providing organizations in California only conduct toxin analysis for suspected

[3] The success of decision tree models in the Tick Tick Bloom competition is consistent with other DrivenData competitions where the task was point-based prediction from satellite imagery (i.e., estimating the amount of water in snowpack and estimating levels of air pollution).

**Table 3**. *Data decisions from model experimentation*

| Decision | Explanation |
|---|---|
| Filter points farther than 550m from a water body | A small amount of noise in the competition dataset was caused by a combination of human error, GPS device error, or a lack of adequate precision in recorded latitude and longitude. Excluding points that are farther than 500m from a water body helps ensure that the model learns from real-world environmental characteristics of cyanobacteria blooms rather than patterns in human error (see below for additional details). Applying this filter decreased the train set size from 17,060 to 11,299 and the test set size from 6,510 to 4,938. |
| Use Sentinel-2 as the sole satellite source | Landsat data primarily only added value for the time period prior to July 2015, when Sentinel-2 data became available. Most applications of CyFi will be forward looking, meaning Sentinel-2 data will be available. As the slowest part of the prediction process is downloading satellite data, incorporating Landsat as a second data source would impose a significant efficiency cost. To rely only on Sentinel-2, any samples prior to the launch of Sentinel-2 were removed from the training and evaluation sets. This further decreased the train set size from 11,299 to 8,979 and the test set size from 4,938 to 4,035. |
| Exclude climate and elevation features | Climate and elevation features primarily provided value for data points prior to the launch of Sentinel-2 and so are not used in the final CyFi model. Climate and elevation likely do have an impact on how cyanobacteria blooms form, and more sophisticated feature engineering with these data sources may add value in the future. This is a direction for Section 5.0.2. |
| Incorporate land cover | Including a land cover map, even at a coarse 300m resolution, aided model accuracy. The land cover map captures farmland areas with fertilizer runoff that contributes to blooms, among other features. A static map from 2020 is used rather than a real-time satellite-derived product, as this reduces the compute time and patterns in land use do not fluctuate daily. Land cover is also an effective balance between reflecting regional characteristics, and avoiding overfitting to the small number data providers in the training set. |

blooms, leading to an over-representation of high density samples among competition data points in California. Predicting high severity for all points in California served well in the competition setting, but would not generalize to the real world. As a result, geographic features like longitude, state, and region were not used for the deployed CyFi model.

Competitions can also surface data quality issues. A number of competition winners pointed out that upon inspection of satellite imagery, some competition data points appeared to be outside of any water body. We believe that the small amount of noise in the competition dataset was caused by a combination of human error, GPS device error, and a lack of adequate precision in recorded latitude and longitude.[4]

While it is important to remove points that have erroneous GPS coordinates, it is also the case that sampling point locations are often recorded from a dock or parking lot near where the water sample was taken. In these cases, the bounding box around the sampling point used to generate features would still pick up on relevant water-based characteristics. Filtering out samples that are far from any water body, and keeping points that are on land but *near* water pixels, is the best method to separate relevant data from incorrect coordinates.

The distance between each sample and the nearest water body was calculated using the European Space Agency (ESA) WorldCover 10m 2021 product on Google Earth Engine. Samples farther than 550m from a water body were excluded to help ensure that the relevant water body fell within the portion of the satellite image from which features were calculated. The WorldCover dataset was chosen over Sentinel-2's scene classification band as the water classification appeared to be more reliable based on visual review of samples.

### 4.2.2. *Satellite feature engineering decisions:*

### 4.2.3. *Target variable decisions:*

## *4.3. User interview takeaways*

Technical experimentation alone is insufficient to build a tool that effectively addresses a real-world problem. Understanding user needs and day-to-day processes helps enable integration with existing workflows and increases the likelihood of adoption. The table below synthesizes key insights gleaned from Section 3.2.2, and outlines how each insight supported the development of a user-friendly package.

**Table 4**. *Satellite feature engineering decisions from model experimentation*

| Decision | Explanation |
|---|---|
| Filter to water area and use a large bounding box | Land pixels are filtered out because they are usually greener than water areas, and can generate falsely high cyanobacteria estimates. Sentinel-2's scene classification band is not perfectly accurate, but is sufficient for masking non-water pixels. Since ground sampling points are often on land but *near* water (taken from the shore or the dock), a large bounding box of 2,000m is used to ensure that relevant water pixels are included. |
| Use a large look-back window and filter to images with almost no clouds | When selecting relevant imagery, CyFi uses Sentinel-2's scene classification band to calculate the percent of clouds in the bounding box. Any imagery that has greater than 5% clouds is not used. CyFi combines a relatively large look-back window of 30 days before the sample with this strict cloud threshold to increase the chances of finding a cloud-free image. |
| Use only one image per sample point | Some winning solutions averaged predictions over multiple satellite images within a specified range. We find that this favors static blooms. We use only the most recent cloud-free image to better detect short-lived blooms. |

[4] At the equator, a longitude value to 2 decimal degrees is only accurate to around a 1km distance [27].

**Table 5**. *Target variable decisions from model experimentation*

| Decision | Explanation |
|---|---|
| Estimate density instead of severity | We learned during user interviews that states use different thresholds for action, so predicting density instead of severity categories supports a broader range of use cases. The winning competition models were trained to predict severity. During experimentation, we validated that there was sufficient signal to predict at the higher granularity of exact density. |
| Train the model to predict log density | We find transforming density into a log scale for model training and prediction yields better accuracy, as the underlying data is highly skewed. About 75% of samples have a density less than 400,000 cell/mL, but there are extreme densities into the tens of millions cells/mL. A log scale helps the model learn that incorrectly estimating a density of 100,000 when the true density is 0 is much more important than incorrectly estimating a density of 1,100,000 when the true density is 1,000,000. The estimate a user sees has been converted back into (non-log) density. |

## 4.4. *CyFi*

The culmination of the machine learning competition, subsequent model experimentation, and user interviews is CyFi. CyFi, short for Cyanobacteria Finder, is an open-source Python package that uses satellite imagery and machine learning to detect cyanobacteria levels, one type of HAB. CyFi can help decision makers protect the public by flagging the highest-risk areas in lakes, reservoirs, and rivers quickly and easily. CyFi incorporates open-source best practices, including tests and continuous integration, and is ready for use in state-level dashboards and decision-making processes.

### 4.4.1. *Data sources:*

CyFi relies on two data sources as input:

1. Sentinel-2 satellite imagery

**Table 6**. *CyFi design decisions rooted in HCD interviews*

| Interview insight | CyFi design decision |
|---|---|
| States tend to have designated sampling locations or locations of reported blooms. Coverage of the full area of a water body is nice but not necessary. | CyFi will expect sampling points as input rather than polygons, and the output will be point-estimates rather than a gridded heatmap. |
| Thresholds are not universal and actions vary by state. | Prediction will be a density value rather than severity category. |
| While blooms in small water bodies can change quickly, the maximum cyanobacteria estimation cadence is daily. | A sampling point will be a unique combination of date, latitude, and longitude. Additional time granularity is not needed. |
| Many states include a visual review of imagery (satellite or submitted photo) as part of the decision-making process. | CyFi will include a way to see the underlying satellite data for a given prediction point, to help users build confidence and intuition around the CyFi model. |
| States have their own tools for managing water quality data (e.g. ground samples and lab results). | CyFi will output a simple CSV file that includes identifying columns for joining with external data. |

2. Land cover classifications

**Sentinel-2** is a wide-swath, high-resolution, multi-spectral imaging mission. The Sentinel-2 Multispectral Instrument (MSI) samples 13 spectral bands: four bands at 10 meters, six bands at 20 meters, and three bands at 60 meters spatial resolution. The mission provides global coverage of the Earth's land surface every 5 days. Sentinel-2 data is accessed through Microsoft's Planetary Computer.

CyFi uses high-resolution Sentinel-2 satellite imagery (10-30m) to focus on smaller water bodies with rapidly changing blooms. This is a significant improvement in resolution over Sentinel-3, which is used by most existing satellite-based cyanobacteria detection tools and has a resolution of 300-500m.

The Climate Research Data Package **Land Cover Gridded Map** (2020) categorizes land surface into 22 classes, which have been defined using the United Nations Food and Agriculture Organization's Land Cover Classification System (LCCS). The map is based on data from the Medium Resolution Imaging Spectrometer (MERIS) sensor on board the polar-orbiting Envisat-1 environmental research satellite by the European Space Agency. CyFi accesses the data using the CCI-LC database hosted by the ESA Climate Change Initiative's Land Cover project.

### 4.4.2. *Feature processing:*

Each observation (or "sampling point") is a unique combination of date, latitude, and longitude. Feature generation for each observation is as follows:

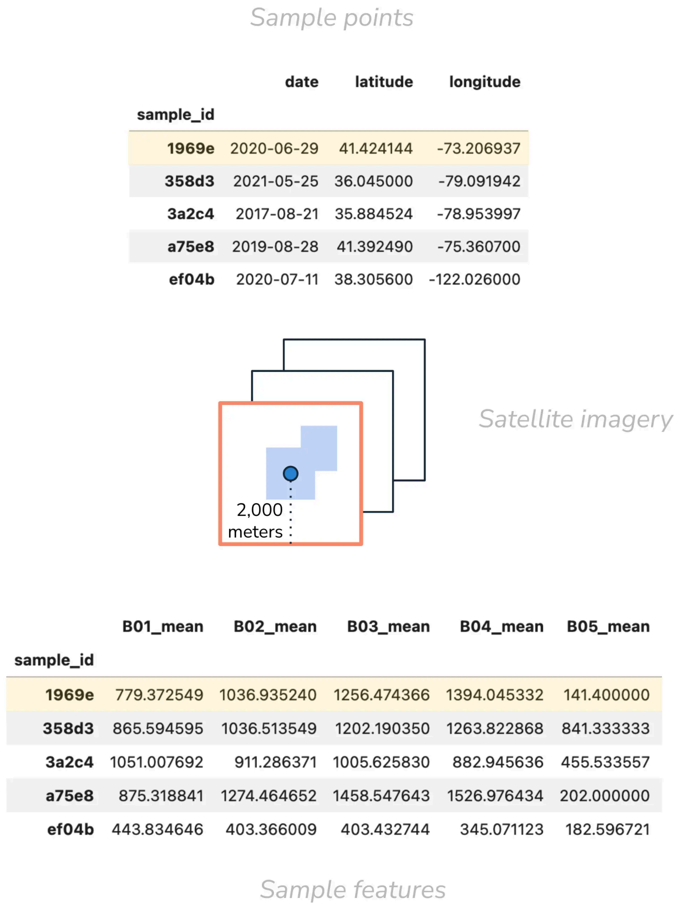
1. Download up to 15 relevant Sentinel-2 tiles based on a bounding box of 2,000m around the sampling point and a time range of 30 days prior to (and including) the sampling date.
2. Select the most recent image that has a bounding box containing less than 5% cloud pixels. If none of the images meet this criteria, no prediction is made for that sampling point.
3. Filter the pixels in the bounding box to the water area using the scene classification (SCL) band.
4. Generate band summary statistics (e.g., mean, 95th percentile) and ratios (e.g, green-blue ratio, NDVI) using 15 different Sentinel-2 bands. The full list of satellite image features is here: cyfi/config.py
5. Calculate two satellite metadata features: 1) the month of selected satellite image and 2) the number of days between the sampling date and the satellite image capture.
6. Look up static land cover map data for the sampling point, and combine land cover information with satellite features.

### 4.4.3. *Model:*

Cyanobacteria estimates are generated by a gradient-boosted decision tree algorithm built with LightGBM [23]. The hyperparameters can be found here: cyfi/config.py.

The model was trained and evaluated using "in situ" labels collected manually by many organizations across the U.S. The train-test split was maintained from the Section 3.1, with Section 4.2.1 to remove samples prior to the launch of Sentinel-2 or more than 550m from a water body.

CyFi was ultimately trained on 5,721 of the 8,979 train set observations due to satellite



| sample_id | date | latitude | longitude |
|---|---|---|---|
| 1969e | 2020-06-29 | 41.424144 | -73.206937 |
| 358d3 | 2021-05-25 | 36.045000 | -79.091942 |
| 3a2c4 | 2017-08-21 | 35.884524 | -78.953997 |
| a75e8 | 2019-08-28 | 41.392490 | -75.360700 |
| ef04b | 2020-07-11 | 38.305600 | -122.026000 |

| sample_id | B01_mean | B02_mean | B03_mean | B04_mean | B05_mean |
|---|---|---|---|---|---|
| 1969e | 779.372549 | 1036.935240 | 1256.474366 | 1394.045332 | 141.400000 |
| 358d3 | 865.594595 | 1036.513549 | 1202.190350 | 1263.822868 | 841.333333 |
| 3a2c4 | 1051.007692 | 911.286371 | 1005.625830 | 882.945636 | 455.533557 |
| a75e8 | 875.318841 | 1274.464652 | 1458.547643 | 1526.976434 | 202.000000 |
| ef04b | 443.834646 | 403.366009 | 403.432744 | 345.071123 | 182.596721 |

**Figure 5.** *Mock up of satellite data selection and processing. The dot represents the sample point; the square represents the 2,000m bounding box around the sample point. The multiple squares outlined in black represents the multiple satellite image contenders within the lookback period. The orange outlined square indicates the selected, most-recent satellite image. The blue shaded area indicates the water pixels in the bounding box from which features are calculated.*

*Note that not all features are represented in the columns. The table above shows a few features calculated based on the B01 (aerosol), B02 (blue), B03 (green), B04 (red), and B05 (red edge) Sentinel-2 bands.*

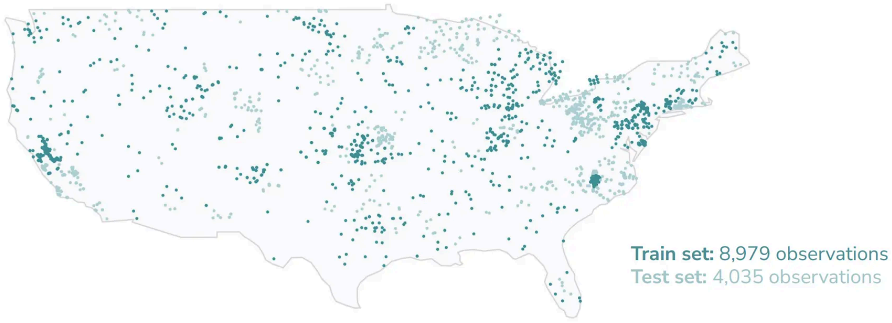


**Figure 6.** *Location and distribution of training and evaluation data for CyFi.*

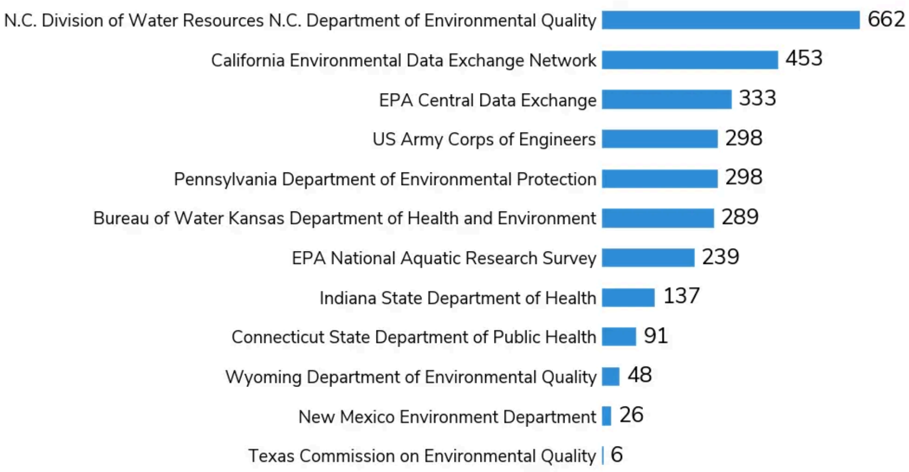


**Figure 7**. *Counts of cyanobacteria measurements by data provider in the evaluation set.*

imagery availability.[5]

### 4.4.4. *Performance:*

CyFi was evaluated using 2,880 ground measurements from 12 data providers spanning the time range August 2015 to December 2021.[6] Given that CyFi relies on Sentinel-2 imagery, the earliest date in the evaluation set aligns with the launch of Sentinel-2 (mid 2015). Of these points, 1,153 were low severity, 504 were moderate severity, and 1,223 were high severity according to ground measurement data. Some states only conduct toxin analysis when blooms are suspected, which may account for the large number of high-severity observations in the evaluation set.

We use the following categories based on World Health Organization [22] for evaluation:

- **Non-bloom:** Cyanobacteria density is less than 20,000 cells/mL
- **Bloom:** Cyanobacteria density is at least 20,000 cells/mL
- **Severe bloom:** Cyanobacteria density is greater than 100,000 cells/mL. Severe blooms are a subset of blooms

On this evaluation dataset, CyFi detects 48% of **non-blooms** with 63% precision. Being able to detect places *not* likely to contain blooms enables ground sampling staff to de-prioritize low-risk sampled locations and better allocate limited resources.

CyFi detects 81% of **blooms** with 70% precision. Based on user interviews, moderate blooms are important to identify because they should be prioritized for sampling. There may be negative public health impacts and more precise toxin analysis is needed.

Lastly, CyFi detect 53% of **severe blooms** with 71% precision. These locations pose the highest risk of severe negative health impacts, and are critical to flag for decision makers to prioritize for public health action (e.g., issuing advisories). In the most severe cases, additional visual inspection of the satellite imagery used by CyFi may be sufficient to issue an advisory without additional sampling. CyFi enables this step with its Section 4.4.7 functionality.

[5] CyFi did not use the remaining 3,258 points in the train set due to a lack of valid satellite data. In order to generate features, there must be at least one satellite image within 30 days prior to the sampling date where cloud pixels account for less than 5% of the pixels in the bounding box around the sampling point.

[6] CyFi did not produce cyanobacteria estimates for the remaining 1,155 points in the test set due to a lack of valid satellite data. In order to produce an estimate, there must be at least one satellite image within 30 days prior to the sampling date where cloud pixels account for less than 5% of the pixels in the bounding box around the sampling point.

**Table 7**. *CyFi estimates compared to ground truth advisory level for out-of-sample California data from 2023*

*This table shows the promise of CyFi in relative ordering of points by severity level as well as the limitations of absolute predicted values. Predicted density correctly increases with the severity of the advisory level, although absolute density is generally overestimated. For example, the expected density range is 0-20,000 cells/mL for "No bloom" and 20,000-100,000 cells/mL for "Caution" based on World Health Organization guidelines. California data in the training set was heavily biased toward severe blooms due to California's sampling protocols, likely driving this overestimation. A promising mitigation strategy is the inclusion of more true negative points from California in the training data.*

| Ground truth advisory level | Median predicted cyanobacteria density |
|---|---|
| No bloom | 92,560 cells/mL |
| Caution | 206,923 cells/mL |
| Warning / Danger | 295,558 cells/mL |

Model accuracy can vary based on bloom severity as well as location and other attributes of the sampling point, so the performance metrics above will vary based on the distribution in the evaluation set. After concluding the model experimentation phase, we conducted a small out-of-sample evaluation using new data collected by California during summer 2023 (231 total observations). We found that estimated cyanobacteria densities increased with the severity of the advisory level that was issued. While the relative ordering of points based on estimated severity was promising, absolute cyanobacteria densities were consistently overestimated. This reinforces that the main immediate use case of CyFi is to identify comparatively higher and lower priority areas, and to inform rather than replace ground sampling activities.

### 4.4.5. *Benchmark comparison:*

An apples-to-apples comparison with one of the leading tools for cyanobacteria estimation from satellite imagery is provided as a more objective benchmark of performance. The Cyanobacteria Index was developed through the Cyanobacteria Assessment Network (CyAN). It is based on a spectral shape algorithm that relates to chlorophyll absorption, initially using Sentinel-3's MERIS and now using OLCI [28], [29], [30], [31].

Due to lower resolution of satellite imagery as well as missing data, CyAN's Cyanobacteria Index is only able to provide estimates for 30% of points in the evaluation set described in Figure 7 (756 points). A major advantage of CyFi is coverage of small water bodies. Over half of the points in the evaluation set were identified as "land" by CyAN due to the coarse resolution of Sentinel-3 imagery. An additional 18% of points had "no data" likely due to clouds or bad imagery.

Among the portion of the evaluation set captured by CyAN, CyFi detects blooms with slightly higher accuracy. Using a cutoff of 10,000 cells/mL per S. Mishra, R. P. Stumpf, B. Schaeffer, P. J. Werdell, K. A. Loftin, and A. Meredith [32], we find CyFi has a presence/absence accuracy of 72% compared to 66% for CyAN. The improved accuracy is largely due to a higher correct classification of true positive cases (blooms).

CyFi correctly identifies 306 of 389 blooms, while CyAN correctly identifies 169 (true positives). CyFi correctly identifies 241 of 367 non-blooms, while CyAN correctly identifies 332 (true negatives).

This performance shows that CyFi offers similar accuracy as a leading tool but at 10m resolution instead of 300m. This dramatic increase in spatial granularity enables remote sensing-based bloom detection in small, inland water bodies across the U.S.

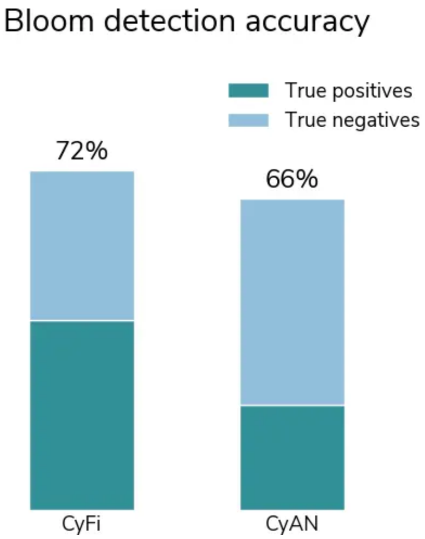


**Figure 8.** *A comparison of CyFi and CyAN model accuracy on 756 ground sampled data points from across the U.S. A true positive (bloom presence) is where cyanobacteria density > 10,000 cells/mL.*

### 4.4.6. *Using CyFi:*

Comprehensive instructions for using CyFi can be found in the CyFi docs. The below provides an overview of some of CyFi's key functionality.

CyFi is designed to be simple to use. To get started, users can install CyFi with pip.[7]

```
$ pip install cyfi
```

Cyanobacteria predictions can then be generated with a single command. The only information needed to generate a prediction is a location (latitude and longitude) and date.

```
$ cyfi predict-point --lat 35.6 --lon -78.7 --date 2023-09-25

SUCCESS  | Estimate generated:
date                    2023-09-25
latitude                      35.6
longitude                    -78.7
density_cells_per_ml        22,836
severity                  moderate
```

For each sampling point, CyFi downloads recent cloud-free Sentinel-2 data from Microsoft's Planetary Computer, calculates a set of summary statistics using the spectral bands for the portion of the image around the sampling point, and passes those features to a LightGBM model which produces an estimated cyanobacteria density.

CyFi also makes it easy to generate cyanobacteria estimates for many points at once. Users can input a CSV with columns for date, latitude, and longitude.

A CSV input can also be processed with a single command.

[7] Alternatively, CyFi can be installed with conda (`conda install -c conda-forge cyfi`), which is recommended for M1 Mac users as of July 2024 due to a known issue with the pip installation of LightGBM.

**Table 9.** *CyFi outputted csv (`preds.csv`) containing predictions*

| sample_id | date | latitude | longitude | density_cells_per_ml | severity |
|---|---|---|---|---|---|
| 7ff4b4a56965d80f6aa501cc25aa1883 | 2023-06-22 | 41.424144 | -73.206937 | 34,173 | moderate |
| 882b9804a3e28d8805f98432a1a9d9af | 2023-07-01 | 36.045 | -79.0919415 | 7,701 | low |
| 10468e709dcb6133d19a230419efbb24 | 2023-08-04 | 35.884524 | -78.953997 | 4,053 | low |

**Table 8.** *Example input csv (`samples.csv`) containing the sampling points where cyanobacteria estimates are needed*

| latitude | longitude | date |
|---|---|---|
| 41.424144 | -73.206937 | 2023-06-22 |
| 36.045 | -79.0919415 | 2023-07-01 |
| 35.884524 | -78.953997 | 2023-08-04 |

```
$ cyfi predict samples.csv

SUCCESS  | Loaded 3 sample points (unique combinations of date, latitude, and longitude) for prediction
SUCCESS  | Downloaded satellite imagery
SUCCESS  | Cyanobacteria estimates for 4 sample points saved to preds.csv
```

Cyanobacteria estimates are saved out as a CSV that can be plugged into any existing decision-making process. For each point, the model provides an estimated density in cells per mL for detailed analysis. Densities are also discretized into severity levels based on World Health Organization guidelines [22].

### 4.4.7. *CyFi Explorer:*

CyFi also comes with a visualization tool called CyFi Explorer. CyFi Explorer surfaces the corresponding Sentinel-2 imagery for each cyanobacteria estimate. The explorer runs a Gradio app locally on the user's machine and is intended to enable visual inspection of where the model is performing well, as well as edge cases or failure modes. It is not intended to replace more robust data analytics tools and decision-making workflows.

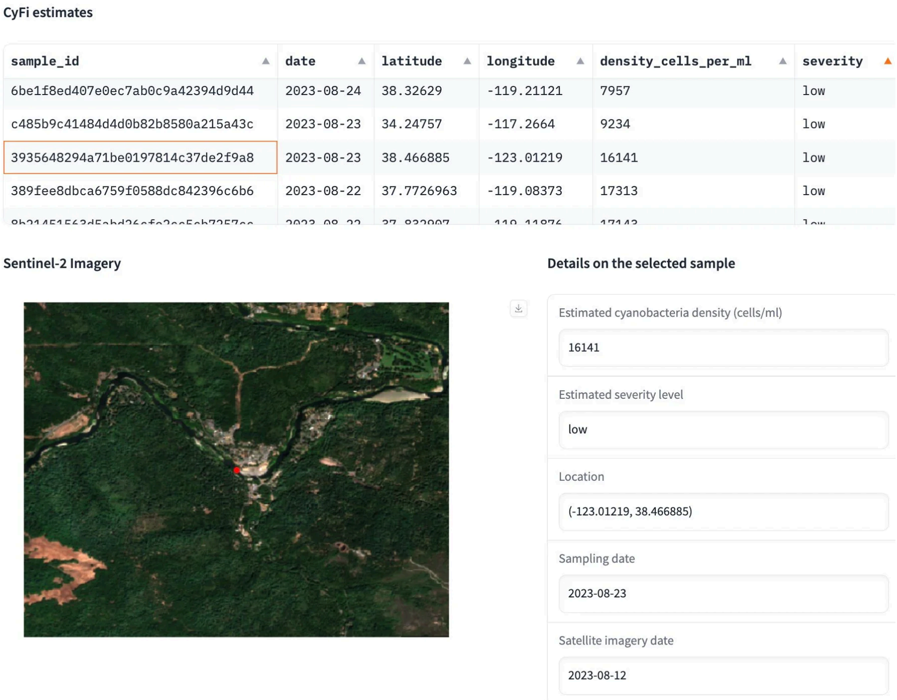
## 5. Discussion

CyFi's progression from a machine learning competition that surfaced promising approaches, through subsequent user interviews and model iteration, to a deployment-ready open source package illustrates a replicable pathway for developing powerful machine learning tools in domain-specific areas.

We find that CyFi performs at least as well as a leading Sentinel-3 based tool, but has significantly greater coverage of water bodies across the U.S. due to the higher resolution

**Table 10.** *WHO Recreational Guidance/Action Levels for Cyanobacteria [22]*

| Relative Probability of Acute Health Effects | Cyanobacteria (cells/mL) |
|---|---|
| Low | < 20,000 |
| Moderate | 20,000–100,000 |
| High | > 100,000–10,000,000 |
| Very High | > 10,000,000 |



CyFi estimates

| sample_id | date | latitude | longitude | density_cells_per_ml | severity |
|---|---|---|---|---|---|
| 6be1f8ed407e0ec7ab0c9a42394d9d44 | 2023-08-24 | 38.32629 | -119.21121 | 7957 | low |
| c485b9c41484d4d0b82b8580a215a43c | 2023-08-23 | 34.24757 | -117.2664 | 9234 | low |
| 3935648294a71be0197814c37de2f9a8 | 2023-08-23 | 38.466885 | -123.01219 | 16141 | low |
| 389fee8dbca6759f0588dc842396c6b6 | 2023-08-22 | 37.7726963 | -119.08373 | 17313 | low |
| 8b21451563d5abd26cfe2ce5cb7257ce | 2023-08-22 | 37.832907 | -119.11876 | 17143 | low |

Sentinel-2 Imagery

Details on the selected sample

Estimated cyanobacteria density (cells/ml)
16141

Estimated severity level
low

Location
(-123.01219, 38.466885)

Sampling date
2023-08-23

Satellite imagery date
2023-08-12

**Figure 9**. *Screenshot of CyFi Explorer, a visualization tool that surfaces the underlying satellite imagery used to generate the cyanobacteria estimate.*

of Sentinel-2 data. This dramatically expands the applicability of remote sensing-based estimates as a tool for management of HABs.

### 5.0.1. *Implications for use:*

CyFi works best as an enhancement to existing decision-making processes through its ability to surface high and low priority areas. At its current accuracy level, we believe CyFi should be used to inform human workflows rather than triggering automatic actions.

Based on discussions with end users, a few common use cases for CyFi are listed below. Overall, CyFi supports more widespread and timely public health actions, better allocation of ground sampling resources, and more informed impairment and regulatory monitoring.

1. Flag high severity blooms for public health action

High concentrations of cyanobacteria often merit public health interventions. Having daily estimates at designated sampling points can quickly and easily surface worrisome conditions. States have the flexibility to design their own processes for how to use this information. For example, some states may choose to prioritize these locations for ground sampling where advisory levels are dependent upon toxin analysis results. Other states may choose to take action such as issuing a press release based on visual review of imagery alone.

1. Identify locations where ground sampling can be deprioritized

Identifying water bodies that are *not* experiencing blooms can be just as helpful as identifying water bodies that are. Ground sampling is time and labor intensive, and CyFi enables water quality managers to deprioritize sampling in the areas least likely to contain blooms.

1. Confirm publicly reported blooms with more data

Multiple states rely on visual inspection of a submitted photo to confirm a bloom. CyFi can both generate a cyanobacteria density estimate and show the most recent, cloud free 10m satellite imagery for that location.

1. Provide a birds-eye view of lake conditions across the state

Many states track impaired and threatened waters in accordance with the Clean Water Act and develop total maximum daily loads (TMDLs), which specify the maximum amount of pollutant allowed to enter a water body. Routine predictions from CyFi can help monitor the progression in water bodies where cyanobacteria is a primary concern.

### 5.0.2. *Future directions:*

While CyFi represents a significant step forward in detecting cyanobacteria from satellite imagery, challenges remain. CyFi is the least reliable for the following cases:

- In very narrow or small waterways
- When there are clouds obscuring the area around a sampling point
- Where multiple water bodies are nearby in the same satellite image

Model performance could be improved by retraining with additional ground measurements for true negative cases, adding water body segmentation to exclude pixels from non-contiguous water bodies, and adding cloud segmentation to remove cloud pixels from feature calculations. Additionally, incorporating more sophisticated time-series climate features may enhance model accuracy. To support users who desire comprehensive estimates across an entire water body, a pre-processing step could be added that accepts a water body polygon as input and transforms this into a grid of sample points.

As decision-makers begin experimenting with CyFi, we recommend calculating historical estimates and comparing these against prior ground measurements to get a baseline accuracy for CyFi's performance. Using CyFi Explorer to review predictions can provide further insight into water bodies that may be particularly challenging for CyFi.

## 6. Conclusion

CyFi is a powerful tool for identifying high and low levels of cyanobacteria, and enables humans to make more timely and targeted decisions when issuing public health guidance around current cyanobacteria levels. Areas with low-density cyanobacteria counts can be excluded from ground sampling to better prioritize limited resources, while areas with high-density cyanobacteria counts can be prioritized for public health action. The development of CyFi illustrates the utility of machine learning competitions as a first step toward open source tools. CyFi's primary use cases show how machine learning can be incorporated into human workflows to enable more efficient and more informed decision making.

### Acknowledgements

This project was supported by funding from National Aeronautics and Space Administration Science Mission Directorate's Earth Science, Applied Sciences, Health and Air Quality, and the NASA Prizes, Challenges, and Crowdsourcing Programs. The project was led by DrivenData and managed by the NASA Tournament Lab, part of the Prizes, Challenges, and Crowdsourcing Program in NASA's Space Technology Mission Directorate for the NASA Open Innovation Services 2 (NOIS2) contract 80JSC020D0041.

The authors express their gratitude to collaborators, advisors, and data providers at the following organizations, whose support made this study possible: National Oceanic and Atmospheric Administration, Environmental Protection Agency, Alaska Department of Environmental Conservation, Arizona Department of Environmental Quality, Bureau of Water Kansas Department of Health and Environment, California Environmental Data Exchange Network, Centers for Disease Control and Prevention, Connecticut State Department of Public Health, Delaware National Resources and the University of Delaware's Citizen Monitoring Program, EPA Central Data Exchange, EPA National Aquatic Research Survey, EPA Ohio, EPA Water Quality Data Portal, Indiana State Department of Health, Iowa Department of Natural Resources, Louisiana Department of Environmental Quality, Maine Bureau of Water Quality - Division of Environmental Assessment, N.C. Division of Water Resources N.C. Department of Environmental Quality, New Jersey Department of Environmental Protection, New Mexico Environment Department, New York State Department of Environmental Protection, North Dakota Department of Environmental Quality, Pennsylvania Department of Environmental Protection, Rhode Island Department of Environmental Management - Office of Water Resources, South Carolina Department of Health and Environmental Control, State of Georgia - Environmental Protection Division, State of Michigan - Lake Michigan Unit - Surface Water Assessment Section - Water Resources Division - Department of Environment Great Lakes and Energy, Tennessee Department of Environment and Conservation, Texas Commission on Environmental Quality, UMRBA (Upper Mississippi River Basin Association), U.S. Army Corps of Engineers, USGS Water Quality Data Portal - Harmful Algal Bloom Science in Texas, Utah Department of Environmental Quality - Division of Water Quality, Vermont Department of Health - Health and the Environment, Virginia Department of Health, West Virginia Department of Environmental Protection, Wisconsin Department of Natural Resources, Wyoming Department of Environmental Quality. The authors would like to thank Yang Xu, Andrew Wheeler, and Raphael Kimina, the Tick Tick Bloom competition winners whose modeling approaches provided the foundation for CyFi. The authors also thank the reviewers who provided constructive comments and suggestions to improve the quality of the manuscript.

Correspondence regarding the CyFi Python package should be directed to Emily Dorne at emily@drivendata.org. Correspondence regarding this open innovation project should be directed to Shobhana Gupta at shobhana.gupta@nasa.gov. Correspondence regarding the NASA Tournament Lab should be directed to Ryon Stewart at ryon.stewart@nasa.gov. Mention of trade names or commercial products does not constitute endorsement or recommendation for use by the U.S. Government. The views expressed in this article are those solely of the authors and do not necessarily reflect the views or policies of the U.S. Government.

## References

[1] CDC, "Harmful Algal Bloom (HAB)-Associated Illness." [Online]. Available: https://www.cdc.gov/harmful-algal-blooms/about/index.html

[2] M. Papenfus, B. Schaeffer, A. I. Pollard, and K. Loftin, "Exploring the potential value of satellite remote sensing to monitor chlorophyll-a for US lakes and reservoirs," *Environmental Monitoring and Assessment*, vol. 192, no. 12, 2020, doi: 10.1007/s10661-020-08631-5.

[3] DrivenData, "CyFi: Cyanobacteria Finder." [Online]. Available: https://cyfi.drivendata.org/

[4] DrivenData, "Tick Tick Bloom: Harmful Algal Bloom Detection Challenge: Results." [Online]. Available: https://www.drivendata.org/competitions/143/tick-tick-bloom/

[5] O. M. Pulido, "Phycotoxins by Harmful Algal Blooms (HABS) and Human Poisoning: An Overview," *International Clinical Pathology Journal*, vol. 2, no. 6, 2016, doi: 10.15406/icpjl.2016.02.00062.

[6] G. Treuer, C. Kirchhoff, M. C. Lemos, and F. McGrath, "Challenges of managing harmful algal blooms in US drinking water systems," *Nature Sustainability*, vol. 4, no. 11, pp. 958–964, 2021, doi: 10.1038/s41893-021-00770-y.

[7] A. Lad *et al.*, "As We Drink and Breathe: Adverse Health Effects of Microcystins and Other Harmful Algal Bloom Toxins in the Liver, Gut, Lungs and Beyond," *Life*, vol. 12, no. 3, p. 418, 2022, doi: 10.3390/life12030418.

[8] S. B. Watson *et al.*, "The re-eutrophication of Lake Erie: Harmful algal blooms and hypoxia," *Harmful Algae*, vol. 56, pp. 44–66, 2016, doi: 10.1016/j.hal.2016.04.010.

[9] P. Hoagland *et al.*, "The Costs of Respiratory Illnesses Arising from Florida Gulf Coast Karenia brevis Blooms," *Environmental Health Perspectives*, vol. 117, no. 8, pp. 1239–1243, 2009, doi: 10.1289/ehp.0900645.

[10] P. Hoagland, D. M. Anderson, Y. Kaoru, and A. W. White, "The economic effects of harmful algal blooms in the United States: Estimates, assessment issues, and information needs," *Estuaries*, vol. 25, no. 4, pp. 819–837, 2002, doi: 10.1007/bf02804908.

[11] R. M. Khan, B. Salehi, M. Mahdianpari, F. Mohammadimanesh, G. Mountrakis, and L. J. Quackenbush, "A Meta-Analysis on Harmful Algal Bloom (HAB) Detection and Monitoring: A Remote Sensing Perspective," *Remote Sensing*, vol. 13, no. 21, p. 4347, 2021, doi: 10.3390/rs13214347.

[12] J. M. Clark *et al.*, "Satellite monitoring of cyanobacterial harmful algal bloom frequency in recreational waters and drinking water sources," *Ecological Indicators*, vol. 80, pp. 84–95, 2017, doi: 10.1016/j.ecolind.2017.04.046.

[13] Ohio Environmental Protection Agency, "Developing a Harmful Algal Bloom (HAB) Treatment Optimization Protocol." [Online]. Available: https://dam.assets.ohio.gov/image/upload/epa.ohio.gov/Portals/28/documents/habs/TreatmentOptimizationProtocol.pdf

[14] United States Environmental Protection Agency, "Climate Change and Freshwater Harmful Algal Blooms." [Online]. Available: https://www.epa.gov/habs/climate-change-and-freshwater-harmful-algal-blooms

[15] P. Bull, I. Slavitt, and G. Lipstein, "Harnessing the Power of the Crowd to Increase Capacity for Data Science in the Social Sector." [Online]. Available: https://arxiv.org/abs/1606.07781

[16] V. Da Poian *et al.*, "Leveraging open science machine learning challenges for data constrained planetary mission instruments," *RAS Techniques and Instruments*, vol. 3, no. 1, pp. 156–165, 2024, doi: 10.1093/rasti/rzae009.

[17] P. A. Johnson *et al.*, "Laboratory earthquake forecasting: A machine learning competition," *Proceedings of the National Academy of Sciences*, vol. 118, no. 5, 2021, doi: 10.1073/pnas.2011362118.

[18] S. Gupta, E. Gelbart, R. Gupta, K. Wetstone, and E. Dorne, "Cyanobacteria Aggregated Manual Labels Dataset." [Online]. Available: http://dx.doi.org/10.5067/SeaBASS/CAML/DATA001

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

[20] M. Ester, Kriegel Hans-Peter, J. Sander, and X. Xu, "A density-based algorithm for discovering clusters in large spatial databases with noise," in *KDD'96: Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, 1996, pp. 226–231.

[21] Office of Water, "Recommendations for Cyanobacteria and Cyanotoxin Monitoring in Recreational Waters," 2019. [Online]. Available: https://www.epa.gov/sites/default/files/2019-09/documents/recommend-cyano-rec-water-2019-update.pdf

[22] World Health Organization, "Guidelines for safe recreational water environments. Volume 1: Coastal and fresh waters," 2003. [Online]. Available: https://iris.who.int/bitstream/handle/10665/42591/9241545801.pdf

[23] G. Ke *et al.*, "LightGBM: A highly efficient gradient boosting decision tree," in *Advances in neural information processing systems*, 2017, pp. 3149–3157.

[24] T. Chen and C. Guestrin, "XGBoost: A Scalable Tree Boosting System," 2016, doi: 10.48550/ARXIV.1603.02754.

[25] A. V. Dorogush, V. Ershov, and A. Gulin, "CatBoost: gradient boosting with categorical features support." [Online]. Available: https://arxiv.org/abs/1810.11363

[26] DrivenData, "Meet the winners of the Tick Tick Bloom challenge." [Online]. Available: https://drivendata.co/blog/tick-tick-bloom-challenge-winners

[27] "Decimal degrees." [Online]. Available: https://en.wikipedia.org/wiki/Decimal_degrees#Precision

[28] E. A. Urquhart, B. A. Schaeffer, R. P. Stumpf, K. A. Loftin, and P. J. Werdell, "A method for examining temporal changes in cyanobacterial harmful algal bloom spatial extent using satellite remote sensing," *Harmful Algae*, vol. 67, pp. 144–152, 2017, doi: 10.1016/j.hal.2017.06.001.

[29] T. T. Wynne, R. P. Stumpf, M. C. Tomlinson, and J. Dyble, "Characterizing a cyanobacterial bloom in Western Lake Erie using satellite imagery and meteorological data," *Limnology and Oceanography*, vol. 55, no. 5, pp. 2025–2036, 2010, doi: 10.4319/lo.2010.55.5.2025.

[30] S. Mishra, R. P. Stumpf, B. A. Schaeffer, P. J. Werdell, K. A. Loftin, and A. Meredith, "Measurement of Cyanobacterial Bloom Magnitude using Satellite Remote Sensing," *Scientific Reports*, vol. 9, no. 1, 2019, doi: 10.1038/s41598-019-54453-y.

[31] T. T. Wynne *et al.*, "Relating spectral shape to cyanobacterial blooms in the Laurentian Great Lakes," *International Journal of Remote Sensing*, vol. 29, no. 12, pp. 3665–3672, 2008, doi: 10.1080/01431160802007640.

[32] S. Mishra, R. P. Stumpf, B. Schaeffer, P. J. Werdell, K. A. Loftin, and A. Meredith, "Evaluation of a satellite-based cyanobacteria bloom detection algorithm using field-measured microcystin data," *Science of The Total Environment*, vol. 774, p. 145462, 2021, doi: 10.1016/j.scitotenv.2021.145462.

Open Access | https://doi.org/10.25080/JFYN3740

# Funix - The laziest way to build GUI apps in Python

**Forrest Sheng Bao**[1], **Mike Qi**[2], **Ruixuan Tu**[3], and **Erana Wan**[4]

[1]Founder, Textea, Inc., [2]Engineer, Textea, Inc., [3]Undergraduate student, Dept. of Computer Sciences, University of Wisconsin-Madison, [4]MS student, Dept. of Computer Science, University of Southern California

SciPy 2024

July 8 - July 14, 2024

Proceedings of the 23rd Python in Science Conference

ISSN: 2575-9752

**Published** Jul 10, 2024

**Correspondence to**
Forrest Sheng Bao
forrest.bao@gmail.com

## Abstract

The rise of machine learning (ML) and artificial intelligence (AI), especially the generative AI (GenAI), has increased the need for wrapping models or algorithms into GUI apps. For example, a large language model (LLM) can be accessed through a string-to-string GUI app with a textbox as the primary input. Most of existing solutions require developers to manually create widgets and link them to arguments/returns of a function individually. This low-level process is laborious and usually intrusive. Funix automatically selects widgets based on the types of the arguments and returns of a function according to the type-to-widget mapping defined in a theme, e.g., `bool` to a checkbox. Consequently, an existing Python function can be turned into a GUI app without any code changes. As a transcompiler, Funix allows type-to-widget mappings to be defined between any Python type and any React component and its `props`, liberating Python developers to the frontend world without needing to know JavaScript/TypeScript. Funix further leverages features in Python or its ecosystem for building apps in a more Pythonic, intuitive, and effortless manner. With Funix, a developer can make it (a functional app) before they (competitors) fake it (in Figma or on a napkin).

**Keywords** type hints, docstrings, transcompiler, frontend development

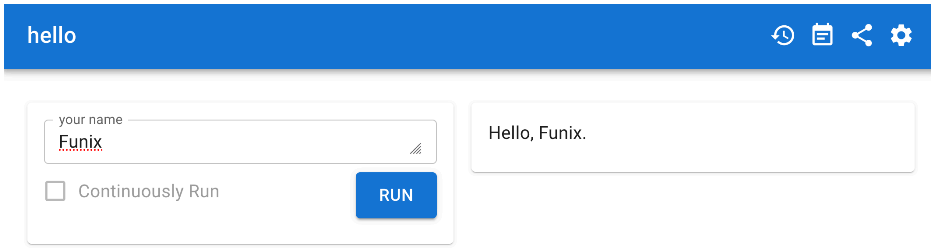
## 1. Introduction

Presenting a model or algorithm as a GUI application is a common need in the scientific and engineering community. For example, a large language model (LLM) is not accessible to the general public until it is wrapped with a chat interface, consisting of a text input and a text output. Since most scientists and engineers are not familiar with frontend development, which is JavaScript/TypeScript-centric, there have been many solutions based on Python, one of the most popular programming languages in scientific computing, especially AI. Examples include ipywidgets, Streamlit, Gradio, Reflex, Dash, and PyWebIO. Most of these solutions follow the conventional GUI programming philosophy, requiring developers to manually select widgets from a widget library and associate them with the arguments and returns of an underlying function, commonly referred to as the “callback function.”

This approach has several drawbacks. **First**, it is manual and repetitive. A developer needs to manually create widgets, align them with the signature of the callback function, and maintain the alignment manually should any of the two changes. **Second**, the choice of widgets is limited to those provided by a specific GUI library, as expanding the widget library requires significant knowledge and effort that the target users are unlikely to possess. **Third**, these solutions do not leverage the features of the Python language itself to automate or streamline the process. For example, most existing solutions require developers to manually specify the labels of widgets, even though such information is often already available in the `Parameters` or `Args` sections of a function’s Docstrings, which are common in Python

```python
# hello.py
def hello(your_name: str) -> str:
    return f"Hello, {your_name}."
```

**Program 1**. *A hello, world! example in Funix. It is an ordinary Python function with nothing special to Funix. The corresponding GUI app is shown in Figure 1.*



Hello, Funix.

**Figure 1**. *The app generated from Program 1 by the command* `funix hello.py` *via Funix. Funix can be installed as simple as* `pip install funix`*.*

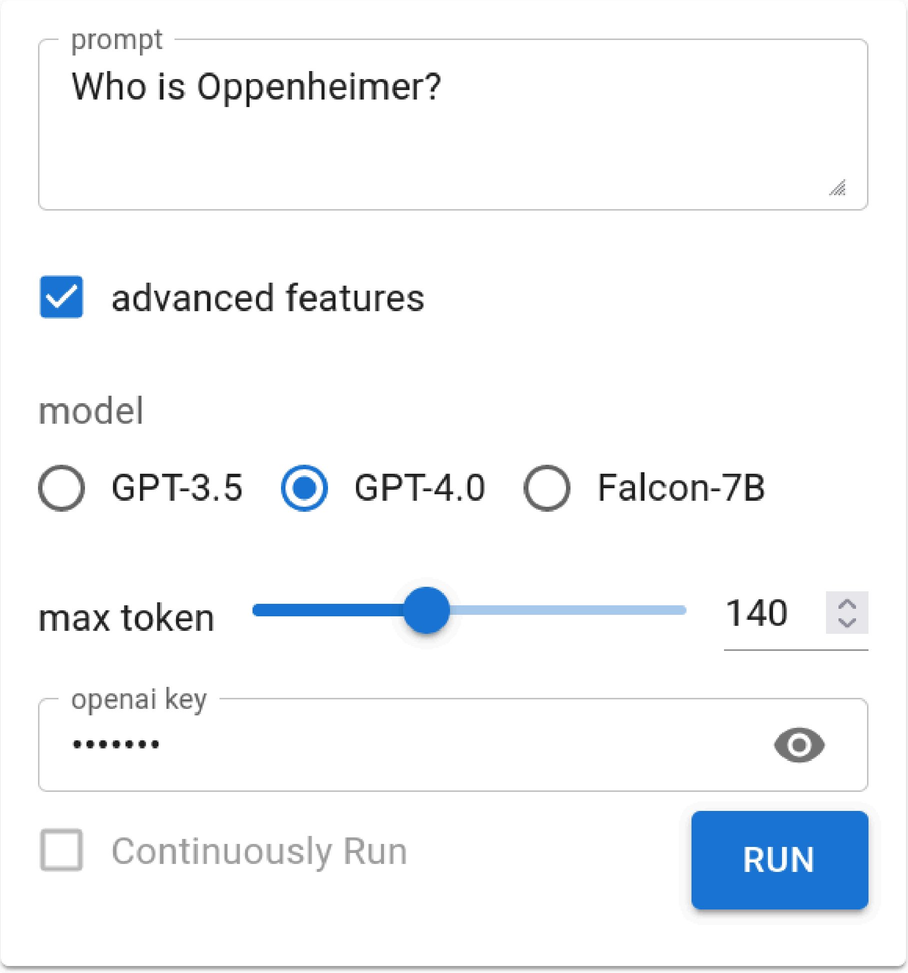
development. As a result, scientific developers, such as geophysicists, neurobiologists, or machine learning engineers, whose jobs are not building apps, are not able to quickly fire up apps to present their models, algorithms, or discoveries to the world.

To address these drawbacks, Funix was created to automatically launch apps from existing Python functions. We observed that the choice of a widget is correlated with the type of the function's input/output that it is associated with. For example, a checkbox is only suitable for Boolean types. Funix automatically selects widgets based on the types of the function's arguments and returns. In the default theme of Funix, Python native types such as `str`, `bool`, and `Literal` are mapped to an input box, a checkbox, and a set of radio buttons in the MUI library, respectively. Common scientific types like `pandas.DataFrame` and `matplotlib.figure.Figure` are mapped to tables (MUI's `DataGrid`) and charts (in `mpld3`). A variable's type can be specified via type-hinting, which is a common practice in Python development, or inferred, which Funix plans to support in the future.

Unlike many of its peers, Funix does not have its own widget library. Any React component can be bound to a Python type. A type-to-widget mapping can be redefined or created [on-the-fly](#Supporting a new type on the fly using the new funix type decorator) or via a reusable [theme](#Defining and using themes). Additionally, the properties (i.e., `props`) of the frontend widget can be configured in Python via JSON. In this sense, Python becomes a surface language for web development. The frontend development world, dominated by

```python
# hello.py
import typing # Python native
import ipywidgets  # popular UI library

def input_widgets_basic(
    prompt: str = "Who is Oppenheimer?",
    advanced_features: bool = True,
    model: typing.Literal['GPT-3.5', 'GPT-4.0', 'Falcon-7B'] = 'GPT-4.0',
    max_token: range(100, 200, 20) = 140,
    openai_key: ipywidgets.Password = "1234556",
    )  -> str:
    pass
```

**Program 2**. *An advanced input widgets example in Funix. The input panel of the corresponding GUI app is shown in Figure 2. In this example, Funix not only uses Python native types but also types from* `ipywidgets`*, a popular UI library in Jupyter.*



prompt
Who is Oppenheimer?

☑ advanced features

model
○ GPT-3.5 ● GPT-4.0 ○ Falcon-7B

max token 140

openai key
•••••••

□ Continuously Run

**Figure 2.** *The input panel of the app generated from Program 2 by Funix, showing a variety of input widgets.*

JavaScript/TypeScript, is now accessible to Python developers who previously couldn't tap into it.

The centralized management of appearances using themes is an advantage of Funix over many of its peers. A theme allows for the automatic launching of apps and ensures consistent GUI appearances across different apps. To change the appearance of an app, simply select a different theme. For further customization, a theme can be extended using the popular JSON format. Often, a Funix developer can leverage themes developed by others to support new types or widgets, similar to how most scientists use LaTeX classes or macros created by others to properly typeset their papers.

More than a GUI generator, Funix is a transcompiler – a program that translates source code from one language to another – which generates both the backend and frontend of an application. Funix wraps the Python function into a Flask app for the backend and generates React code for the frontend. The two ends communicate via WebSocket.

In addition to types, Funix leverages other features of the Python language and ecosystem to further automate app building. Docstrings, commonly used in Python, are utilized by Funix to control the UI appearance. For example, the annotation of an argument in the `Args` (Google-style) or `Parameters` (Numpy-style) section of a docstring is used as the label or tooltip to explain the argument to the app user. Funix also assigns new meanings to certain keywords and types in Python within the context of app building. For instance, `global` is used for states and sessions, `yield` for streaming, and each class becomes a multi-page app where pages share data via the `self` variable. These concepts are detailed in Section Section 5.

In summary, Funix has the following cool features for effortless app building in Python:

1. Automatic, type-based GUI generation controlled by themes
2. Exposing any React component to Python developers
3. Leveraging Python's native features to make app building more intuitive, minimizing the need to learn new concepts specific to Funix.

## 2. Motivation: the demand to rapidly launch low-interactivity and plain-looking apps at scale

When it comes to GUI app development, there is a trade-off between simplicity and versatility. JavaScript/TypeScript-based web frontend frameworks like React, Angular, and Vue.js offer great versatility. However, their versatility is often beyond the reach of most scientists and engineers, except for frontend or full-stack developers, and is usually overkill for most scientific and engineering applications.

As machine learning researchers, we have observed that a significant number of scientific applications share two common features:

1. The underlying logic is a straightforward input-output process – thus complex interactivity, such as dynamically updating an input widget based on user input in another input widget, is not needed.
2. The app itself is not the end goal but a means to it – thus it is not worthy to spend time on building the app.

Existing Python-based solutions such as Streamlit or Gradio do a great job addressing the first feature but are still too complicated for the second one, requiring developers to read their documentation and add code before an app can be launched. In particular, a developer needs to manually select and configure widgets and/or link them to the arguments and returns of a function. This low-level process is laborious. Since versatility is already sacrificed for simplicity in Python-based app building, why not trade it further for even more simplicity?

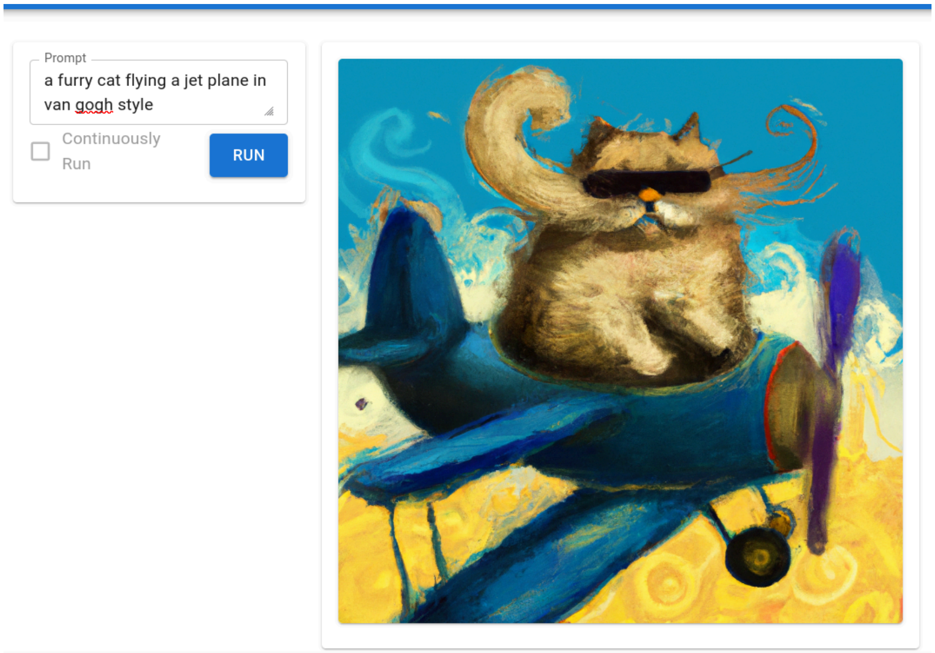
Funix pushes simplicity to the extreme. In the Program 1 example above, an app is launched without requiring any learning or code modification. To achieve this simplicity, Funix leverages common Python development practices, such as type hints (or type inference) and docstrings, to save developers from extra work or learning. Funix does not aim to become a Domain Specific Language (DSL), because it treats the Python programming language itself (including the typing hint syntax and docstring styles) as a surface language for GUI app building.

Because Funix is designed for quickly firing up apps that model straightforward input-output processes, a Funix-generated app consists of two panels: the arguments of the underlying function on the left, input panel and the Section 5.3 on the right, output panel (Program 1 and Figure 1). A more complex process can be decomposed into simple input-output processes and embodied into Section 5.5. The underlying or callback function will be called after the user plugs in the arguments and click the "Run" button. The result will be displayed in the output panel.

We would also like to argue that the rise of Generative AI (GenAI) is simplifying GUIs, as natural languages are becoming a prominent interface between humans and computers. For example, a text-to-image generation app (Figure 3) only needs a string input and an

```
import openai  # pip install openai
import IPython

def dalle(Prompt: str = "a flying cat on a jet plane")
            -> IPython.display.Image:
    client = openai.OpenAI() # defaults to os.environ.get("OPENAI_API_KEY")
    response = client.images.generate(Prompt)
    return response.data[0].url
```

**Program 3.** *Source code for the Dall-E app in Funix. The corresponding GUI app is shown in Figure 3.*



Prompt: a furry cat flying a jet plane in van gogh style

□ Continuously Run

**Figure 3.** *A Dall-E app generated by Funix by simply wrapping OpenAI's image generation API with a* `str`*-to-*`Image` *function. Source code in Program 3.*

image output. In this sense, Funix and its Python-based peers will be able to meet many needs, both in scientific computing and more general applications, in the future.

## 3. Funix's default mapping from Python types to React components

Like CSS, Funix controls the GUI appearance based on the types. Under the hood, Funix transcompiles the signature of a function into React code. Currently, Funix depends on the type hint for every argument or return of a function. In the future, it will support type inference or tracing.

The default mapping from basic Python types to React components is given in Table 1. In particular, we leverage the semantics of `Literal` and `List[Literal]` for single-choice and multiple-choice selections. Two apps exhibiting the diversity of widgets are shown in Figure 2 and Figure 9.

Because Funix is a transcompiler, it leverages multimedia types already defined in popular Python libraries such as `ipywidgets` (Jupyter's input widgets), `IPython` (Jupyter's display system), `pandas`, and `matplotlib`. `ipywidgets` and `IPython` types are mapped to MUI components rather than their respective components for being React compatible. Figure 4 illustrates a data-plot-from-tabular-data app that maps a `pandas.DataFrame` to a table and a `matplotlib.figure.Figure` to a chart.

**Table 1.** *Default mapping from basic Python types to components.*

| Python type | As an input (argument) or output (return) | Widget |
|---|---|---|
| `str` | Input | MUI TextField |
| `bool` | Input | MUI Checkbox or Switch |
| `int` | Input | MUI TextField |
| `float` | Input | MUI TextField or Slider |
| `Literal` | Input | MUI RadioGroup if number of elements is below 8; Select otherwise |
| `range` | Input | MUI Slider |
| `List[Literal]` | Input | An array of MUI Checkboxes if the number of elements is below 8; AutoComplete otherwise |
| `str` | Output | Plain text |
| `bool` | Output | Plain text |
| `int` | Output | Plain text |
| `float` | Output | Plain text |

**Table 2.** *Default mapping from common multimedia/MIME types to components.*

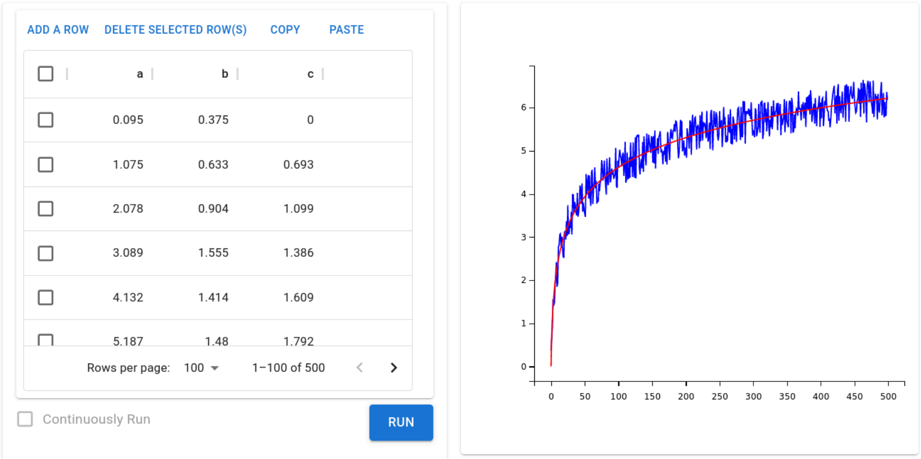
| Python type | As an input (argument) or output (return) | Widget |
|---|---|---|
| `ipywidgets.Password` | Input | MUI TextField with `type="password"` |
| `ipywidgets.Image` | Input | React Dropzone combine with MUI Components |
| `ipywidgets.Video` | Input | React Dropzone combine with MUI Components |
| `ipywidgets.Audio` | Input | React Dropzone combine with MUI Components |
| `ipywidgets.FileUpload` | Input | React Dropzone combine with MUI Components |
| `IPython.display.HTML` | Output | Raw HTML |
| `IPython.display.Markdown` | Output | React Markdown |
| `IPython.display.JavaScript` | Output | Raw JavaScript |
| `IPython.display.Image` | Output | MUI CardMedia with `component=img` |
| `IPython.display.Video` | Output | MUI CardMedia with `component=video` |
| `IPython.display.Audio` | Output | MUI CardMedia with `component=audio` |
| `matplotlib.figure.Figure` | Output | mpld3 |
| `pandas.DataFrame` & `pandera.typing.DataFrame` | Input & Output | MUI DataGrid |

## 4. Customizing the type-to-widget mapping

> **Note**
>
> Introducing a new type-to-widget mapping or modifying an existing one should not be the job of most Funix users but advanced users or user interface specialists. This is like

```
import pandas, matplotlib.pyplot
from numpy import arange, log
from numpy.random import random

def table_and_plot(
   df: pandas.DataFrame = pandas.DataFrame({
       "a": arange(500) + random(500)/5,
       "b": random(500)-0.5 + log(arange(500)+1),
       "c": log(arange(500)+1) })
   ) -> matplotlib.figure.Figure:

   fig = matplotlib.pyplot.figure()
   matplotlib.pyplot.plot(df["a"], df["b"], 'b')
   matplotlib.pyplot.plot(df["a"], df["c"], 'r')

   return fig
```

**Program 4.** *A Python functions with a `pandas.DataFrame` input and a `matplotlib.figure.Figure` output. The corresponding GUI app is shown in Figure 4. The default values populate the table with random numbers.*



| □ | a | b | c |
|---|---|---|---|
| □ | 0.095 | 0.375 | 0 |
| □ | 1.075 | 0.633 | 0.693 |
| □ | 2.078 | 0.904 | 1.099 |
| □ | 3.089 | 1.555 | 1.386 |
| □ | 4.132 | 1.414 | 1.609 |
| □ | 5.187 | 1.48 | 1.792 |

Rows per page: 100 1–100 of 500

**Figure 4.** *A data-plot-from-tablular-data app generated from Program 4 by Funix. The input panel contains a table (`pandas.DataFrame`) and the output panel contains a chart (`matplotlib.figure.Figure`). Both the table and the chart are interactive/editable. As far as we know, no other Python-based solutions supports editable tables as inputs.*

> most scientists who write papers in LaTeX do not develop the LaTeX classes or macros but just use them.

The default mapping from Python types to React components is detailed in the section above. To expand or modify an existing mapping, Funix provides two approaches: the on-the-fly, Section 4.1 and the reusable, Section 4.2.

As a transcompiler that generates React code, Funix does not have its own widget library. Instead, developers can choose any React component on the market (as for now, only some MUI components are supported.) to use as the widget for a Python type. Additionally, Funix allows configuring the properties (`props`) of the frontend widget in Python via JSON. In this way, Funix bridges the Python world with the frontend world, making Python or JSON the surface language for React-based frontend development.

### *4.1. Supporting a new type on the fly using the `new_funix_type` decorator*

Program 5 defines a new type `blackout`, which a special case (as indicated by the inheritance) of `str` and binds it with a widget. Following the convention in the frontend world, Funix identifies a widget by its module specifier in `npm`, the de facto package manager in the

```
from funix import new_funix_type
@new_funix_type(
  widget = {
      "widget": "@mui/material/TextField",
      "props": {
          "type": "password",
          "placeholder": "Enter a secret here."
      }
  }
)
class blackout(str):
   def print(self):
       return self + " is the message."

def hoho(x: blackout = "Funix Rocks!") -> str:
   return x.print()
```

**Program 5**. *An example of introducing a new type, binding it to a widget, and using it.*

frontend world. In Program 5, the widget is identified as `@mui/material/TextField`. Properties of the widget supported by its library for configuration can be modified in the `new_funix_type` decorator as well. As mentioned earlier, this allows a Pythonista to tap into a React component without frontend development knowledge.

The on-the-fly approach is only applicable when introducing a new type, e.g., a custom class. To modify the widget choice for an existing type, a theme must be used.

### *4.2. Defining and using themes*

A type-to-widget mapping can be reused and centralized managed via a theme, which is a simple JSON file. An example is given in Program 6 below where the Python's native types `str`, `int`, and `float` are bound to three widgets. In this exmaple, besides using `npm` module specifier, Funix shorthand strings `inputbox` and `slider` are also used.

There are two ways to apply a theme: script-wide and function-wide. The script-wide approach (Program 7) applies a default theme to all functions in a script. The function-wide approach (Program 8) applies a theme to a specific function. In either case, the theme can be referred to by a web URL, a local file path, or a name/alias. If no theme is specified, Funix uses its default theme.

To refer to a theme by its name or alias, it must be imported. The alias can be set when importing. A theme can be imported from a web URL, a local file path, or a JSON dictionary defining a theme (Program 9).

```
{
 "name": "grandma's secret theme", // space and punctuation allowed
 "widgets": {
   "str": "inputbox",        // Funix' shorthand, non-parametric
   "int": "slider[0,100,2]", // Funix' shorthand, parametric
   "float": {
       "widget": "@mui/material/Slider",
       // using MUI's widget
       // https://mui.com/material-ui/api/slider
       "props": {
           // config props of the frontend widget
           "min": 0,
           "max": 100,
           "step": 0.1
       }
   }
 }
}
```

**Program 6**. *An example theme.*

```
import funix

funix.set_default_theme("http://example.com/sunset_v2.json") # from web URL

funix.set_default_theme("../../sunset_v2.json") # from local file

funix.set_default_theme("grandma's secret theme") # from a name/alias
```

**Program 7**. *Three ways to apply a theme script-wide.*

```
import funix

@funix.funix(theme = "http://example.com/sunset.json") # from web URL
def foo():
  pass

@funix.funix(theme = "../../themes/sunset.json") # from local file
def foo():
  pass

@funix.funix(theme = "grandma's secret theme") # from a name/alias
def foo():
  pass
```

**Program 8**. *Three ways to apply a theme function-wide.*

## 5. Building apps Pythonically

Funix leverages the language features of Python and common practices in Python development to make app building in Python more intuitive and efficient.

### *5.1. Default values as placeholders*

Python supports default values for keyword arguments. Funix directly uses them as the placeholder values for corresponding widgets. For example, default values in Program 2 are prefilled in Figure 2. A more complex example is the `pandas.DataFrame` initated with `numpy` columns in Program 4 are prefilled into the table in Figure 4. In contrast, Funix' peer solutions require developers to provide the placeholder values the second time in the widget initiation.

```
funix.import_theme(
  "http://example.com/my_themes.json",  # from web URL
  alias = "my_favorite_theme"           # alias is optional
)

funix.import_theme(
  "../../themes/my_themes.json",        # from local file
  alias = "my_favorite_theme"           # alias is optional
)

theme_json = { # a Funix theme definition
  "name": "grandma's secret theme"
  "widgets" : {
    "range" : "inputbox"
  }
}

funix.import_theme(
    theme_json,                        # from a JSON theme definition
    alias = "granny's secret theme"    # alias is optional
)
```

**Program 9**. *Three ways to import a theme. Note that theme importing is optional. The only benefit is to refer to a theme by its name or alias for easy switching.*

```python
def foo(x: str, y: int) -> str:
    """## What happens when you multiply a string with an integer?

    Try it out below.

    Parameters
    ----------
    x : str
        A string that you want to repeat.
    y : int
        How many times you want to repeat.

    Examples
    ----------
    >>> foo("hello ", 3)
    "hello hello hello "
    """
    return x * y
```

**Program 10**. *An example of a function with a Google-style docstring. The corresponding GUI app is shown in Figure 5.*

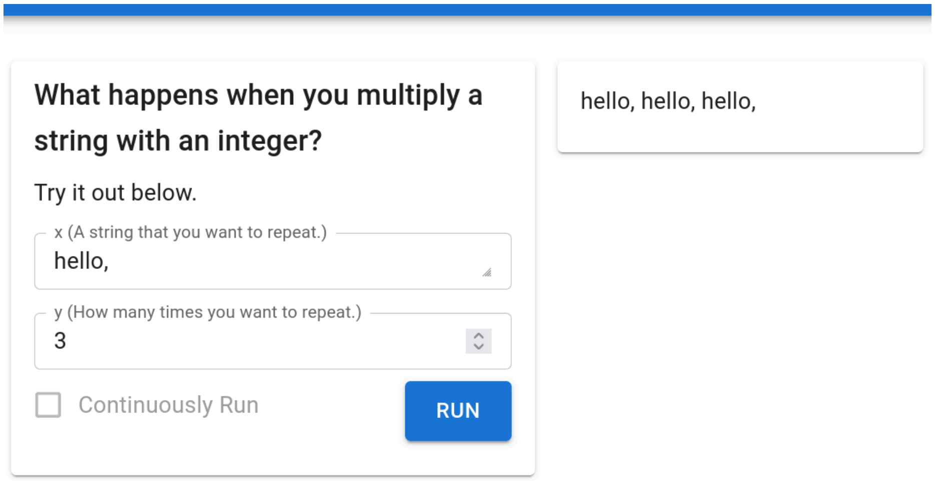
## 5.2. *Making use of docstrings*

Docstrings are widely used in the Python community. Information in docstrings is often needed in the GUI of an app. For example, the annotation of each argument can be displayed as a label or tooltip to explain the meaning of the argument to the app user. Therefore, Funix automatically incorporates selected information from docstrings into apps, eliminating the need for developers to manually add it, as required by other solutions.

Different sections of docstrings will be transformed into various types of information on the frontend. The docstring above the first headed section (e.g., `Parameters` in Program 10) will be rendered as Markdown. Argument annotations in the `Args` section will become labels in the UI. The `Examples` section will provide prefilled example values for users to try out. Funix currently supports only Google-style and Numpy-style docstrings. Supports for sections beyond `Args`/`Parameters` and `Examples` and styles will be added in the future.

## 5.3. *Output layout in `print` and `return`*

In Funix, by default, the return value of a function becomes the content of the output panel. A user can control the layout of the output panel by returning strings, including f-strings, in Markdown and HTML syntaxes. Markdown and HTML strings must be explicitly specified as `IPython.display.Markdown` and `IPython.display.HTML`, respectively. Otherwise, the



What happens when you multiply a string with an integer?

Try it out below.

x (A string that you want to repeat.)
hello,

y (How many times you want to repeat.)
3

□ Continuously Run

hello, hello, hello,

**Figure 5**. *An app with input panel customized by Docstrings.*

```python
from IPython.display import Markdown, HTML
from typing import Tuple
from funix import funix

@funix(print_to_web=True)
def foo(income: int = 200000, tax_rate: float= 0.45) -> Tuple[Markdown, HTML]:
    print (f"## Here is your tax breakdown: \n")

    tax_table = (
        " | Item | $$$ | \n"
        " | --- | --- |  \n"
        f"| Income | {income} | \n"
        f"| Tax | -{tax_rate * income : .3f} | \n"
        f"| Net Income | {income - tax_rate * income : .3f} | \n\n"
    )

    return tax_table, "If you have any question, contact <a href='http://irs.gov'>IRS</a>."
```

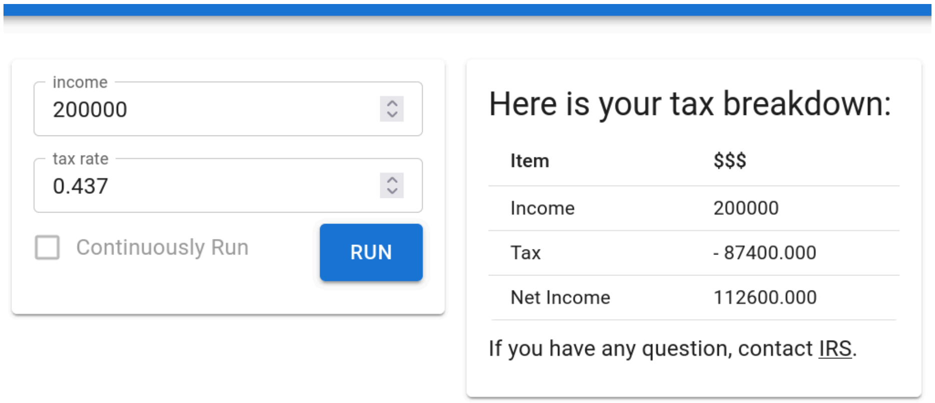
**Program 11**. *Use `print` and `return` to control the output layout. The corresponding GUI app is shown in Figure 6.*

raw strings will be displayed. Since Python supports multiple return values, you can mix Markdown and HTML strings in the return statement.

Quite often, we need to print out some information before a function reaches its return statement. The `print` function, a built-in feature of Python, is frequently used for this purpose. Funix extends this convenience by redirecting the output of `print` to the output panel of an app. Printout strings in Markdown or HTML syntax will be automatically rendered after syntax detection. To avoid conflicting with the default behavior of printing to `stdout`, printing to the web needs to be explicitly enabled using a Boolean decorator parameter `print_to_web` (See Program 11).

### 5.4. *Streaming based on `yield`*

In the GenAI era, streaming is a common method for returning lengthy text output from an AI model. Instead of inventing a new mechanism to support streaming, Funix repurposes the `yield` keyword in Python to stream a function's output. The rationale is that `return` and `yield` are highly similar, and `return` is already used to display the final output in a Funix-powered app.



income: 200000

tax rate: 0.437

Here is your tax breakdown:

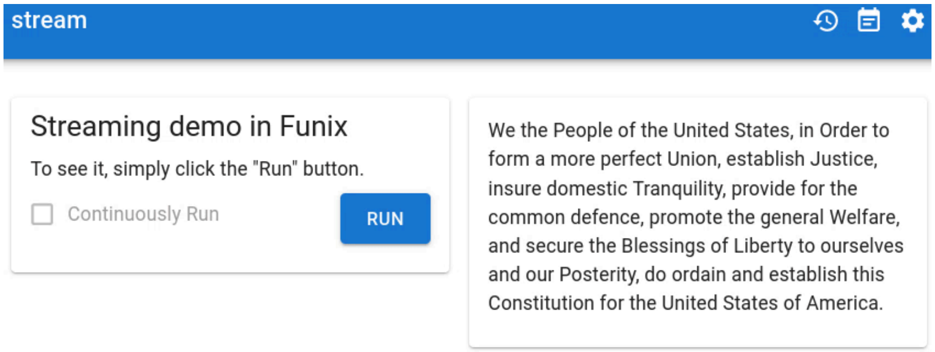
| Item | $$$ |
|---|---|
| Income | 200000 |
| Tax | - 87400.000 |
| Net Income | 112600.000 |

If you have any question, contact IRS.

**Figure 6**. *An app with output panel customized by `print` and `return`. Source code in Program 11.*

```
import time

def stream() -> str:
    """
    ## Streaming demo in Funix

    To see it, simply click the "Run" button.
    """
    message = "We the People of the United States, in Order to form a more perfect Union, establish
Justice, insure domestic Tranquility, provide for the common defence, promote the general Welfare, and
secure the Blessings of Liberty to ourselves and our Posterity, do ordain and establish this Constitution
for the United States of America. "

    for i in range(len(message)):
        time.sleep(0.01)
        yield message[0:i]
```

**Program 12.** *Python keyword* `yield` *is repurposed for streaming in Funix. The corresponding GUI app is shown in Figure 7.*



Streaming demo in Funix

To see it, simply click the "Run" button.

We the People of the United States, in Order to form a more perfect Union, establish Justice, insure domestic Tranquility, provide for the common defence, promote the general Welfare, and secure the Blessings of Liberty to ourselves and our Posterity, do ordain and establish this Constitution for the United States of America.

**Figure 7.** *The streaming demo in Funix. Source code in Program 12.*

## *5.5. States, sessions, and multi-page apps*

### 5.5.1. *A simple approach by using* `global`*:*

In Funix, maintaining states is as simple as updating a `global` variable. By leveraging the semantics of `global` variables which are a native feature of the Python language, Funix saves developers the burden of learning something new in Funix.

A security risk is that there is only one backend server for the Funix app and consequently a `global` variable is accessible by all browser sessions of the app. To eliminate this risk, Funix provides a simple command-line flag `-t` at the launch of the Funix app to sessionize

```
from IPython.display import Markdown

secret_word = "funix"
used_letters = [] # a global variable to maintain the state/session

def guess_letter(Enter_a_letter: str) -> Markdown:
    letter = Enter_a_letter # rename
    global used_letters # state/session as global
    used_letters.append(letter)
    answer = "".join([
        (letter if letter in used_letters else "_")
            for letter in secret_word
        ])
    return f"### Hangman \n `{answer}` \n\n ---- \n ### Used letters \n {', '.join(used_letters)}"
```

**Program 13.** *A simple Hangman game in Funix that uses the* `global` *keyword to maintain the state. This solution is much shorter than using peer solutions, such as in Gradio.*

all `global` variables. If the developer on purpose wants to share the data among different connections, the `-t` flag can be omitted.

### 5.5.2. *Multi-page apps from classes:*

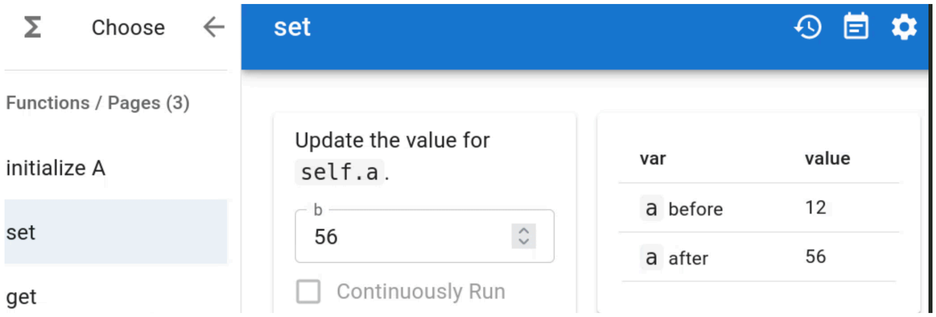
A special but useful case that requires maintaining states is multi-page apps. A multi-page app consists of multiple pages or tabs that share the same variable space. Values of variables set on one page can be accessed on another page.

Without reinventing the wheel, Funix supports this need by turning a Python class into a multi-page app. Each member function of the class becomes a page of the multi-page app, and pages can exchange data via the `self` variable. Specifically, the constructor (`__init__`) becomes the landing page of the multi-page app. Since each instance of the class is independent, the multi-page app is automatically sessionized for different connections from browsers without needing to set the `-t` flag.

Program 14 is a Python class of three member methods, which are turned into three pages of the GUI app by Funix (Figure 8). The GIF shows that values of `self.a` set in either the contructor or updated in the `set` method can be accessed in the `get` method. In this approach, a Funix developer does not need to learn anything new and can easily build a mulit-page app from the OOP principles they are already familiar with.

```python
from funix import funix_method
from IPython.display import Markdown, HTML

class A:
    @funix_method(print_to_web=True)
    def __init__(self, a: int):
        self.a = a
        print(f"`self.a` has been initialized to {self.a}")

    def set(self, b: int) -> Markdown:
        """Update the value for `self.a`. """
        old_a  = self.a
        self.a = b
        return (
                "| var | value |\n"
                "| ----| ------|\n"
            f"| `a` before | {old_a} |\n"
            f"| `a` after | {self.a} |"
            )

    def get(self) -> HTML:
        """Check the value of `self.a`. """
        return f"The value of <code>self.a</code> is <i>{self.a}</i>. "
```

**Program 14.** *A simple multi-page app in Funix leveraging OOP. The corresponding GUI app is shown in Figure 8.*



Update the value for `self.a`.

b: 56

| var | value |
|---|---|
| a before | 12 |
| a after | 56 |

**Figure 8.** *A multiplage app generated by Funix from a class of three member methods including the constructor. Source code in Program 14*

```
from funix import funix
from typing import Literal, List

@funix(
    widgets={
        "animal": "inputbox",
        "activities": "inputbox",
    }
)
def sentence_builder(
    count: range(2, 21) = 4,
    animal: Literal["cat", "dog", "bird"] = "cat",
    countries: List[Literal["USA", "Japan", "Pakistan"]] = ["USA", "Pakistan"],
    location: Literal["park", "zoo", "road"] = "park",
    activities: List[Literal["ran", "swam", "ate", "slept"]] = ["swam", "slept"],
    in_morning: bool = False
) -> str:
    return f"""The {count} {animal}s from {" and ".join(countries)} went to the {location} where they
{" and ".join(activities)} until the {"morning" if in_morning else "night"}"""
```

**Program 15**. *The Funix implementation of a sentence builder. The* `funix` *decorator overwrites the theme-based widgets choices of two arguments.*

## 6. The Funix decorators

Although Funix relies on the Python language (including type hints and docstrings) to define GUI apps, there are still some aspects of an app's appearance and behavior that remain uncovered. This is where the `@funix` decorator comes into play. One example, as mentioned above (Section 5.3), is redirecting the `print` output from `stdout` to the output panel of an app. Here, we provide a few more examples. For full details on Funix decorators, please refer to the Funix reference manual.

### 6.1. *Overriding the type-based widget choice*

Funix uses types to determine the widgets. However, there may be needs to manually pick a widget. The `@funix` dectorator has a `widgets` parameter for this purpose.

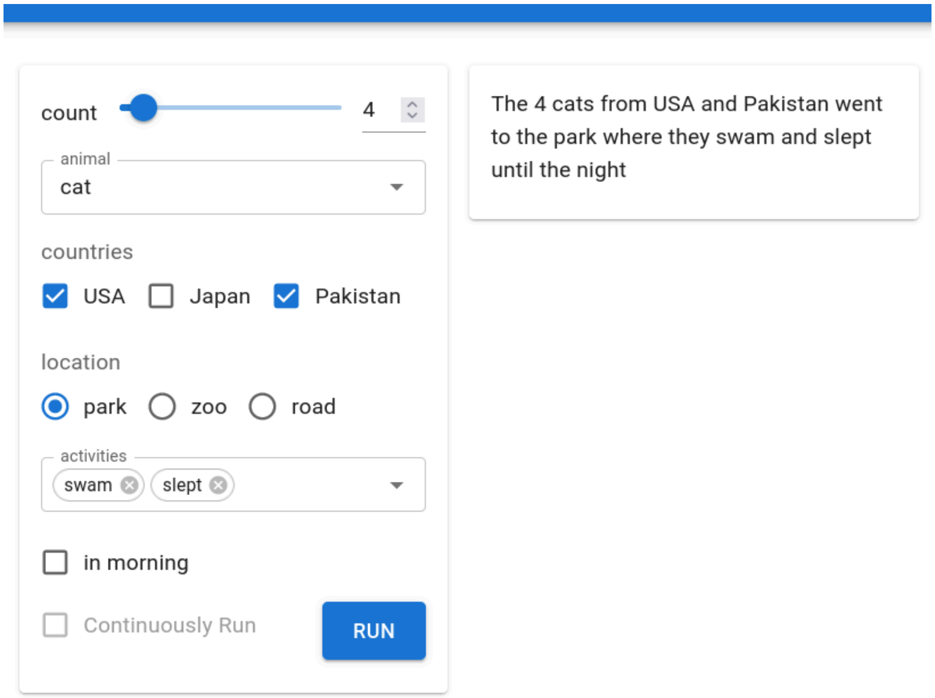
Program 15 is an example to temporarily override the widget choice for two variables of the types `Literal` and `List[Literal]` respectively. The corresponding app (Figure 9) is a sentence builder. The Funix-based code is much shorter and more human-readable than its Gradio-based counterpart, thanks to leveraging the Python-native features like automatic rendering the `return` strings or default values.

### 6.2. *Automatic re-run triggered by input changes*

As mentioned earlier, Funix is suitable for straightforward input-output processes. Such a process is triggered once when the “Run” button is clicked. This may work for many cases but in many other cases, we may want the output to be updated following the changes in the input end automatically. To do so, simply toggle on the `autorun` parameter in the `@funix` decorator. This will activate the “continuously run” checkbox on the input panel.

### 6.3. *Conditional visibility*

Although interactivity is not a strong suit of Funix for reasons aforementioned, Funix still supports some common interactivity features. One of them is “conditional visibility” which reveal some widgets only when certain conditions are met (Program 17 and Figure 11).



count 4

animal: cat

countries

☑ USA □ Japan ☑ Pakistan

location

● park ○ zoo ○ road

activities: swam ⓧ slept ⓧ

□ in morning

□ Continuously Run

RUN

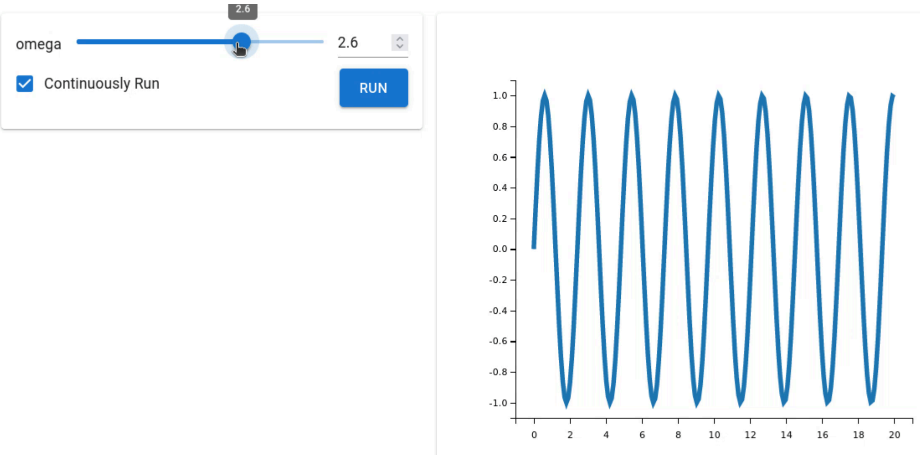
The 4 cats from USA and Pakistan went to the park where they swam and slept until the night

**Figure 9**. *The sentence builder app in Funix. Source code in Program 15. Gradio-based version here.*

## 6.4. Rate limiting

```python
import matplotlib.pyplot, matplotlib.figure
from ipywidgets import FloatRangeSlider
import numpy
from funix import funix

@funix(autorun=True)
def sine(omega: FloatRangeSlider[0, 4, 0.1]) -> matplotlib.figure.Figure:
    fig = matplotlib.pyplot.figure()
    x = numpy.linspace(0, 20, 200)
    y = numpy.sin(x * omega)
    matplotlib.pyplot.plot(x, y, linewidth=5)
    return fig
```

**Program 16**. *A sine wave plotter that re-visualizes the function whenever input changes, kept on using the `autorun` parameter in `@funix` decorator. The corresponding GUI app is shown in Figure 10.*



omega 2.6

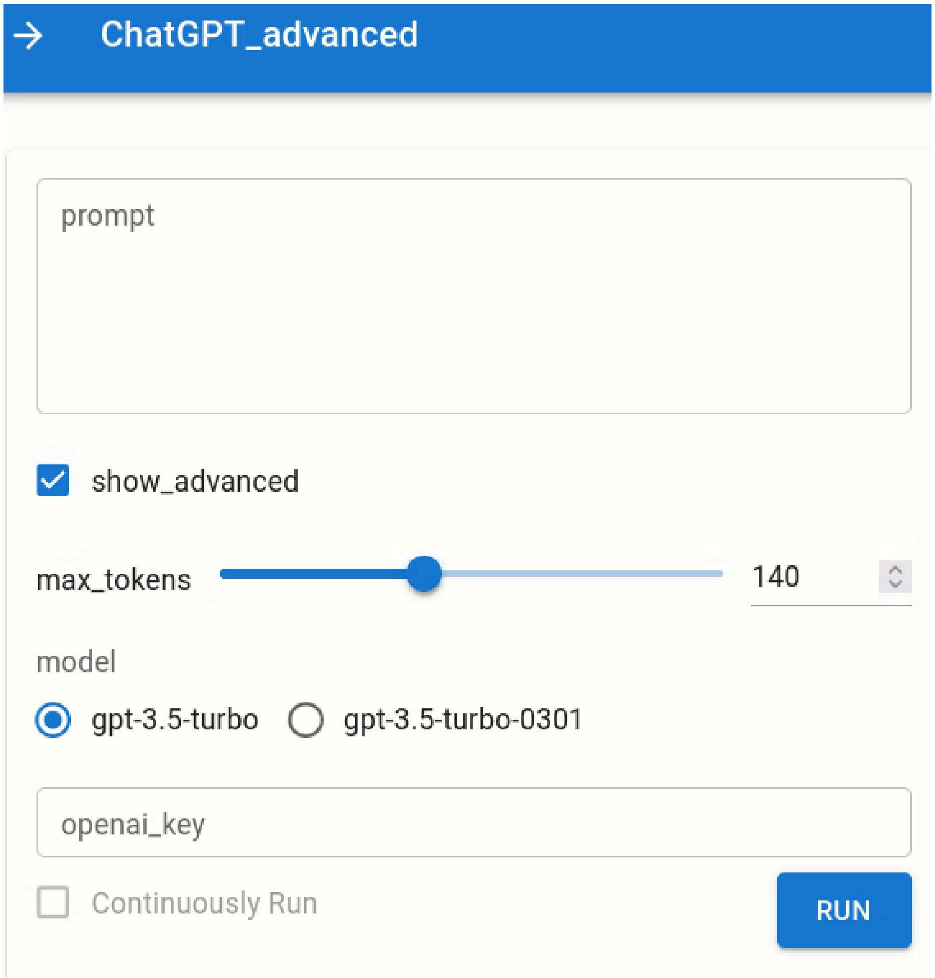
☑ Continuously Run

RUN

**Figure 10**. *A sine wave generator with the `autorun` parameter toggled on. Source code in Program 16.*

```python
import typing
import openai
import funix

@funix.funix(
  conditional_visible=[
    {
      "when": {"show_advanced": True,},
      "show": ["max_tokens", "model", "openai_key"]
    }
  ]
)
def ChatGPT_advanced(
  prompt: str,
  show_advanced: bool = False,
  model : typing.Literal['gpt-3.5-turbo', 'gpt-3.5-turbo-0301']= 'gpt-3.5-turbo',
  max_tokens: range(100, 200, 20)=140,
  openai_key: str = ""
) -> str:
  completion = openai.ChatCompletion.create(
    messages=[{"role": "user", "content": prompt}],
    model=model,
    max_tokens=max_tokens,
  )
  return completion["choices"][0]["message"]["content"]
```

**Program 17.** *Conditional visibility in* `@funix` *decorator. App in action is shown in Figure 11.*



☑ show_advanced

max_tokens 140

model

● gpt-3.5-turbo ○ gpt-3.5-turbo-0301

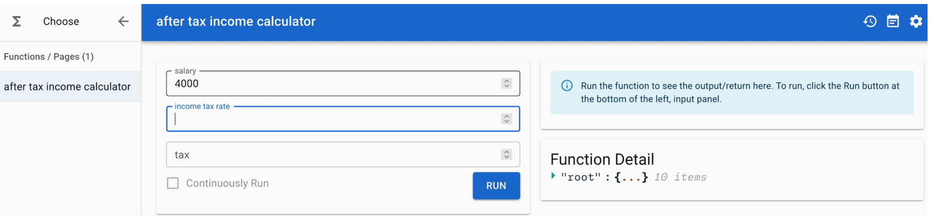
openai_key

□ Continuously Run

**Figure 11.** *An advanced ChatGPT app that only displays advanced options when the* `show_advanced` *checkbox is checked. Source code in Program 17.*

```
from funix import funix

def __compute_tax(salary: float, income_tax_rate: float) -> int:
    return salary * income_tax_rate

@funix(
  reactive={"tax": __compute_tax}
)
def after_tax_income_calculator(
    salary: float,
    income_tax_rate: float,
    tax: float) -> str:
    return f"Your take home money is {salary - tax} dollars,\
    for a salary of {salary} dollars, \
    after a {income_tax_rate*100}% income tax."
```

**Program 18.** *A reactive app.*



salary
4000

Run the function to see the output/return here. To run, click the Run button at the bottom of the left, input panel.

"root" : {...} 10 items

**Figure 12.** *A reactive app in Funix. Source code in Program 18.*

When an app is exposed, a common concern is how to avoid abuses. Rate limiting is a common measure to this. Funix's `@funix` decorator supports rate limiting based on both browser sessions and time.

### 6.5. *Reactive apps*

Funix can dynamically prefill widgets based on information from other widgets. We call this "reactive." An example is given in Program 18. The `tax` argument of the function is populated automatically based on the values of `salary` and `income_tax_rate` as the user enters.

### 6.6. *Showing source code*

Lastly, togging on `show_source` parameter in `@funix` can enable the source code of your app to be displayed.

## 7. Jupyter support

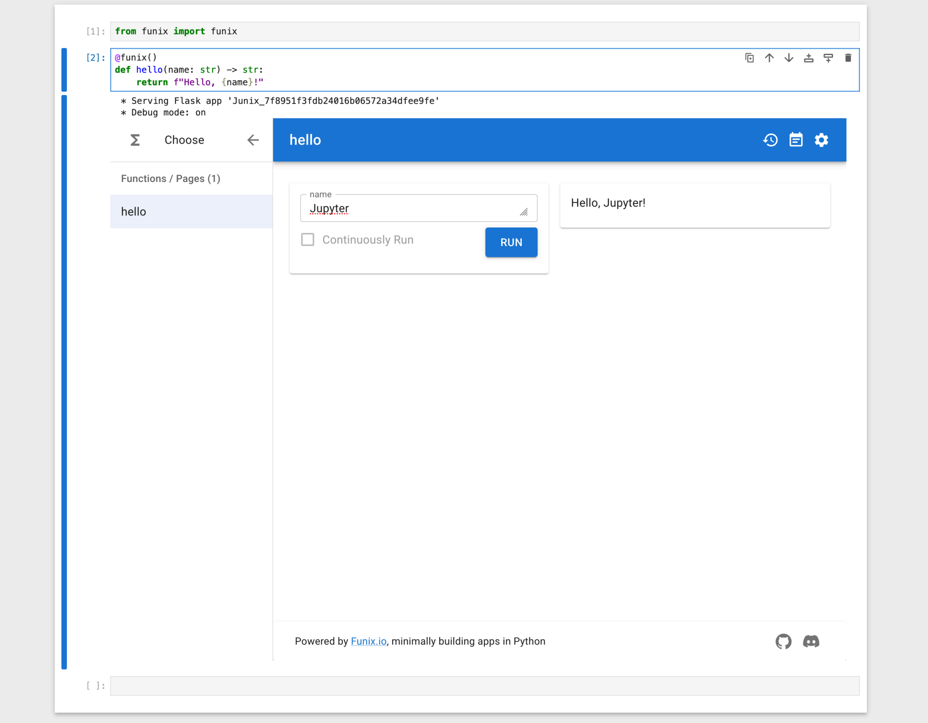
Jupyter is a popular tool for Python development. Funix supports turning a Python function/ class defined in a Jupyter cell into an app inside Jupyter. To do so, simply add the `@funix` decorator to the function/class definition and run the cell (Figure 13).

## 8. Showcases

Lastly, please allow us to use some examples to demonstrate the convenient and power of Funix in quickly prototyping apps. If there is any frontend knowledge needed, it is only HTML.

### 8.1. *Wordle*

The source code can be found here. In Funix, only simple HTML code that changes the background colors of tiles of letters according to the rules of the game Wordle is needed. A GIF showing the game in action is in Figure 14.



```
[1]: from funix import funix
```

```
[2]: @funix()
     def hello(name: str) -> str:
         return f"Hello, {name}!"
```

```
 * Serving Flask app 'Junix_7f8951f3fdb24016b06572a34dfee9fe'
 * Debug mode: on
```

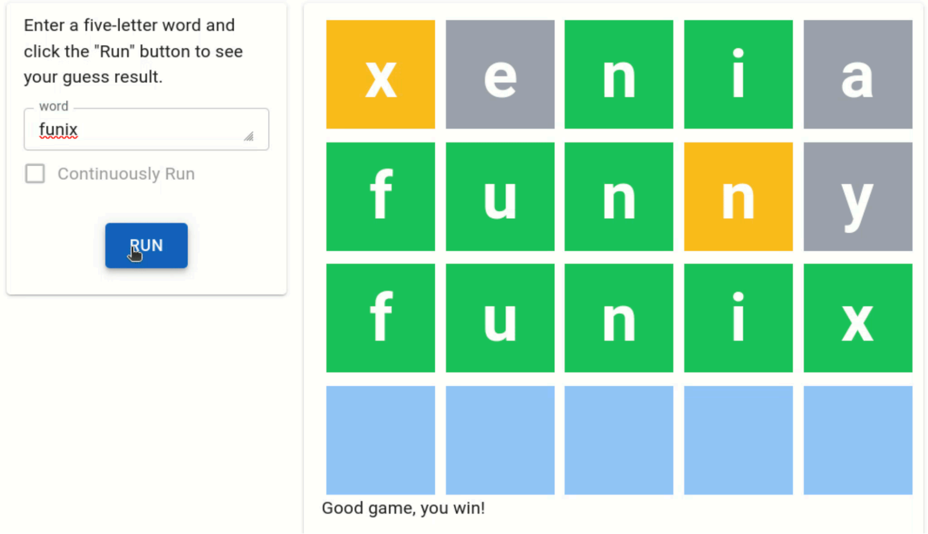
name: Jupyter

Hello, Jupyter!

**Figure 13**. *Funix working in Jupyter.*

## *8.2. ChatGPT multi-turn*

Funix does not have a chat widget, because it is so easy (less than 10 lines in Program 19) to build one using simple alignment controls in HTML. The only thing Funix-specific in the code is using the `@funix` decorator to change the arrangement of the input and output panels from the default left-right to top-down for a more natural chat experience.



Enter a five-letter word and click the "Run" button to see your guess result.

word: funix

Good game, you win!

**Figure 14**. *The Wordle game implemented in Funix. Source code here.*

```
import IPython
from openai import OpenAI
import funix

client = OpenAI()

messages  = []  # list of dicts, dict keys: role, content, system. Maintain the conversation history.

def __print_messages_html(messages):
    printout = ""
    for message in messages:
        if message["role"] == "user":
            align, left, name = "left", "0%", "You"
        elif message["role"] == "assistant":
            align, left, name = "right", "30%", "ChatGPT"
            printout += f'<div style="position: relative; left: {left}; width: 70%"><b>{name}</b>:
{message["content"]}</div>'
    return printout

@funix.funix(
    direction="column-reverse",
)
def ChatGPT_multi_turn(current_message: str)  -> IPython.display.HTML:
    current_message = current_message.strip()
    messages.append({"role": "user", "content": current_message})
    completion = client.chat.completions.create(messages=messages)
    chatgpt_response = completion.choices[0].message.content
    messages.append({"role": "assistant", "content": chatgpt_response})

    return __print_messages_html(messages)
```

**Program 19**. *Multiturn chatbot using Funix.*

### 8.3. *Multimodal inputs*

Funix extends the support to `ipywidgets.{Image, Audio, File, Video}` to allow drag-and-drop of multimedia files or push-to-capture audio or video from the computer's microphone or webcam.

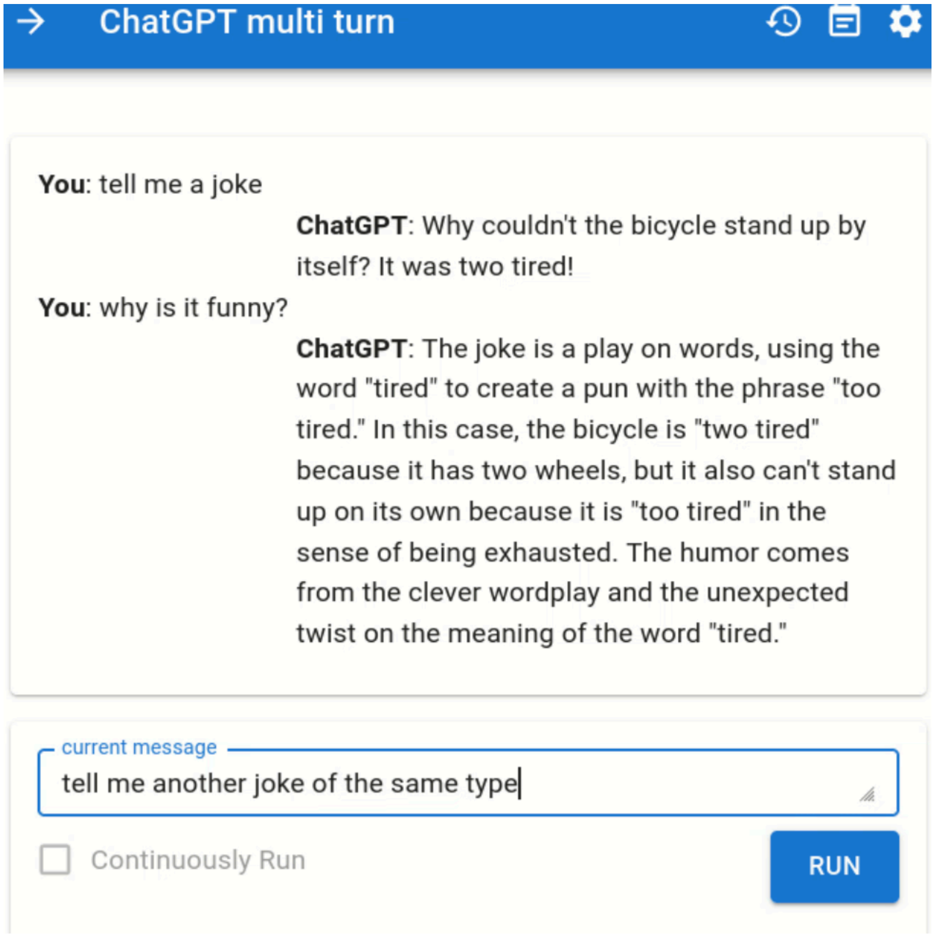
### 8.4. *Vector stripping in bioinformatics*

A vector is a nucleotide sequence that is appended to a nucleotide sequence of interest for easy handling or quality control. It is added before the sequencing process and should be removed after the sequence is read. Vector stripping is the process of removing vectors. A vector stripping app only involves simple data structures, such as strings, lists of strings, and numeric parameters. This is a sweet spot of Funix.

Because the bioinformatics part of vector stripping is lengthy, we only show the interface function in Program 21 and the full source code can be found here. `pandas.DataFrame`'s are used in both the input and output of this app, allowing biologists to batch process vector stripping by copy-and-pasting their data to Excel or Google Sheets, or uploading/downloading CSV files.

## 9. Conclusion

In this paper, we introduce the philosophy and features of Funix. Funix is motivated by the observations in scientific computing that many apps are straightforward input-output processes and the apps are meant to be disposable at a large volume. Therefore, Funix' goal is to enable developers, who are experts in their scientific domains but not in frontend development, to build apps by continue doing what they are doing, without code modification or learning anything new. To get this goal, Funix leverages the language features of the Python language, including docstrings and keywords, to automatically generate the GUIs for apps and control the behaviors of the app. Funix tries to minimize reinventing the wheel



**You**: tell me a joke

**ChatGPT**: Why couldn't the bicycle stand up by itself? It was two tired!

**You**: why is it funny?

**ChatGPT**: The joke is a play on words, using the word "tired" to create a pun with the phrase "too tired." In this case, the bicycle is "two tired" because it has two wheels, but it also can't stand up on its own because it is "too tired" in the sense of being exhausted. The humor comes from the clever wordplay and the unexpected twist on the meaning of the word "tired."

current message: tell me another joke of the same type

**Figure 15.** *A multi-turn chatbot in Funix in action. Source code in Program 19.*

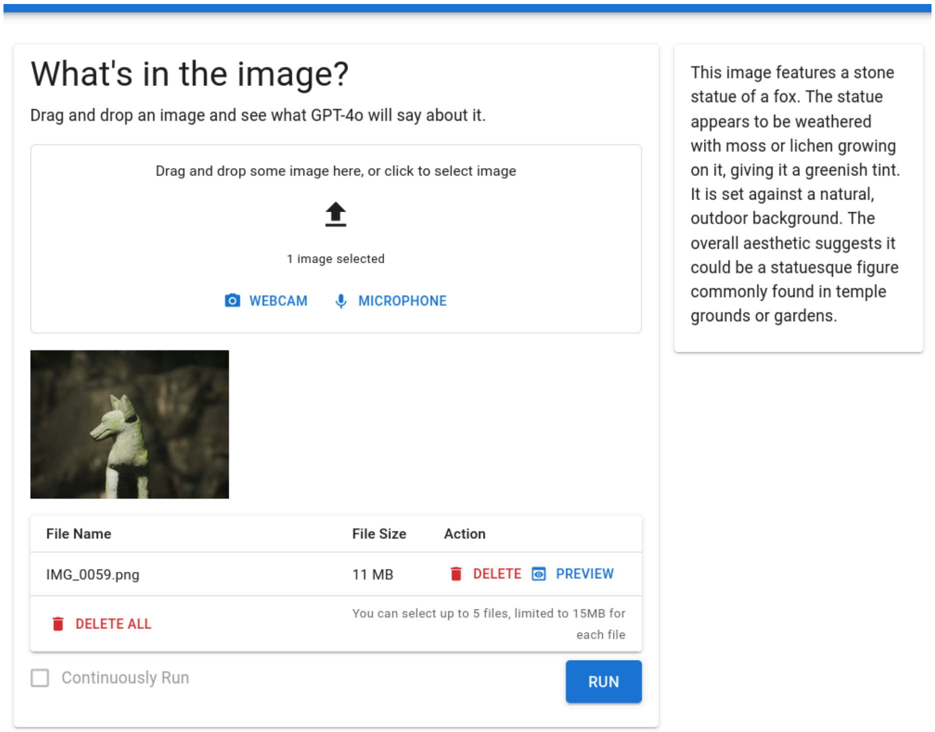
by being a transcompiler between the Python word and the React world. Not only does it expose developers to the limitless resources in the frontend world, but it also minimizes the learning curve. Funix is still a very early-stage project. As an open-source project, we welcome feedback and contributions from the community.

## Acknowledgments

Funix is not the first to exploit variable types for automatical UI generation. Python Fire by Google is a Python library that automatically generates command line interfaces (CLIs) from the signatures of Python functions. Funix extends the idea from CLI to GUIs. `interact` in ipywidgets infers types from default values of keyword arguments and picks widgets accordingly. But it only supports five types/widgets (`bool`, `str`, `int`, `float`, and Dropdown menus) and is not easy to expand the support. We'd like to thank the developers of these projects for their inspiration.

```
import openai
import base64
from ipywidgets import Image

client = openai.OpenAI()

def image_reader(image: Image) -> str:
"""
# What's in the image?

Drag and drop an image and see what GPT-4o will say about it.
"""

# Based on https://platform.openai.com/docs/guides/vision
# with only one line of change
response = client.chat.completions.create(
  model="gpt-4o",
  messages=[
    {
      "role": "user",
      "content": [
          {"type": "text", "text": "What's in this image?"},
          {"type": "image_url",
          "image_url": {
            "url":f"data:image/png;base64,{base64.b64encode(image).decode()}",
          },
          },
        ],
    }
  ],
)
return response.choices[0].message.content
```

**Program 20**. *A multimodal input demo in Funix built by simply wrapping OpenAI's GPT-4o demo code into a function with an* `ipywidgets.Image` *input and a* `str` *output. The corresponding GUI app is shown in Figure 16.*



What's in the image?

Drag and drop an image and see what GPT-4o will say about it.

Drag and drop some image here, or click to select image
1 image selected
WEBCAM
MICROPHONE

| File Name | File Size | Action |
|---|---|---|
| IMG_0059.png | 11 MB | DELETE PREVIEW |

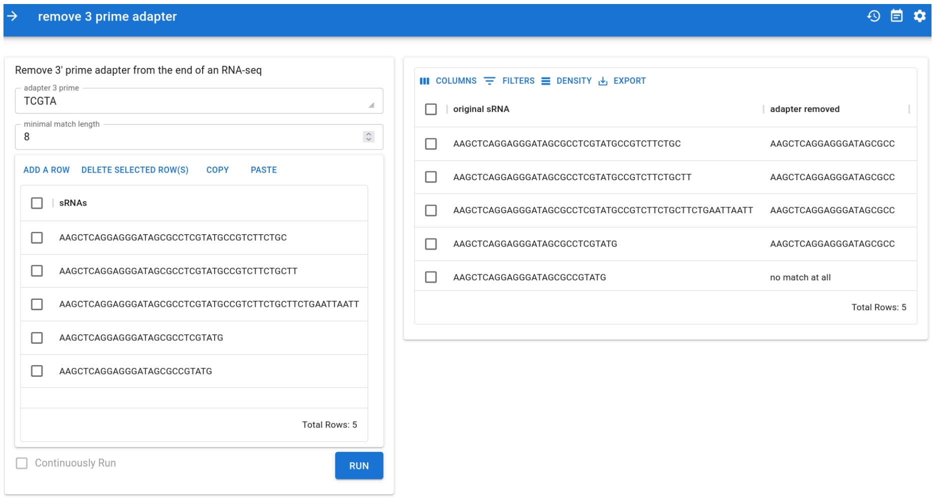
DELETE ALL
You can select up to 5 files, limited to 15MB for each file
Continuously Run
RUN

This image features a stone statue of a fox. The statue appears to be weathered with moss or lichen growing on it, giving it a greenish tint. It is set against a natural, outdoor background. The overall aesthetic suggests it could be a statuesque figure commonly found in temple grounds or gardens.

**Figure 16**. *Funix maps a* `ipywidgets.{Image, Audio, Video, File}`*-type arguments to a drag-and-drop file uploader with push-to-capture ability from the microphone or webcam of the computer. The corresponding source code is in Program 20.*

```python
def remove_3_prime_adapter(
    adapter_3_prime: str="TCGTATGCCGTCTTCTGCTT",
    minimal_match_length: int = 8,
    sRNAs: pandas.DataFrame = pandas.DataFrame(
        {
            "sRNAs": [
                "AAGCTCAGGAGGGATAGCGCCTCGTATGCCGTCTTCTGC",  # shorter than full 3' adapter
                "AAGCTCAGGAGGGATAGCGCCTCGTATGCCGTCTTCTGCTT",  # full 3' adapter
                # additional seq after 3' adapter,
                "AAGCTCAGGAGGGATAGCGCCTCGTATGCCGTCTTCTGCTTCTGAATTAATT",
                "AAGCTCAGGAGGGATAGCGCCTCGTATG",  # <8 nt io 3' adapter
                "AAGCTCAGGAGGGATAGCGCCGTATG",  # no match at all
            ]
        }
    ),
    # ) -> pandera.typing.DataFrame[OutputSchema]:
) -> pandas.DataFrame:

    ## THE BODY HIDDEN

    return pandas.DataFrame(
        {"original sRNA": sRNAs["sRNAs"], "adapter removed": list(return_seqs)}
    )
```

**Program 21**. *The function that is turned into a vector stripping app by Funix.*



Remove 3' prime adapter from the end of an RNA-seq

adapter 3 prime: TCGTA

minimal match length: 8

| □ | sRNAs |
|---|---|
| □ | AAGCTCAGGAGGGATAGCGCCTCGTATGCCGTCTTCTGC |
| □ | AAGCTCAGGAGGGATAGCGCCTCGTATGCCGTCTTCTGCTT |
| □ | AAGCTCAGGAGGGATAGCGCCTCGTATGCCGTCTTCTGCTTCTGAATTAATT |
| □ | AAGCTCAGGAGGGATAGCGCCTCGTATG |
| □ | AAGCTCAGGAGGGATAGCGCCGTATG |

Total Rows: 5

□ Continuously Run

| □ | original sRNA | adapter removed |
|---|---|---|
| □ | AAGCTCAGGAGGGATAGCGCCTCGTATGCCGTCTTCTGC | AAGCTCAGGAGGGATAGCGCC |
| □ | AAGCTCAGGAGGGATAGCGCCTCGTATGCCGTCTTCTGCTT | AAGCTCAGGAGGGATAGCGCC |
| □ | AAGCTCAGGAGGGATAGCGCCTCGTATGCCGTCTTCTGCTTCTGAATTAATT | AAGCTCAGGAGGGATAGCGCC |
| □ | AAGCTCAGGAGGGATAGCGCCTCGTATG | AAGCTCAGGAGGGATAGCGCC |
| □ | AAGCTCAGGAGGGATAGCGCCGTATG | no match at all |

Total Rows: 5

Open Access | https://doi.org/10.25080/NVYF1037

# Ecological and Spatial Influences on the Genetics of Cumacea (Crustacea: Peracarida) in the Northern North Atlantic

**Justin Gagnon**[1], and **Nadia Tahiri**[2]

[1]Department of Biology, University of Sherbrooke, 2500, boul. de l'Université, Sherbrooke, Quebec, J1K 2R1 Canada, [2]Department of Computer Science, University of Sherbrooke, 2500, boul. de l'Université, Sherbrooke, Quebec, J1K 2R1 Canada

SciPy 2024

July 8 - July 14, 2024

Proceedings of the 23rd Python in Science Conference
ISSN: 2575-9752

**Published** Jul 10, 2024

**Correspondence to**
Justin Gagnon
Justin.Gagnon2@USherbrooke.ca

## Abstract

The peracarid taxon Cumacea is an essential indicator of benthic quality in marine ecosystems. This study investigated the influence of environmental (i.e., biological or ecosystemic), climatic (i.e., meteorological or atmospheric), and spatial (i.e., geographic or regional) variables on their genetic variability and adaptability in the Northern North Atlantic, focusing on Icelandic waters. We analyzed partial sequences of the 16S rRNA mitochondrial gene from 62 Cumacea specimens. Using the *aPhyloGeo* software, we compared these sequences with relevant variables such as latitude (decimal degree) at the end of sampling, wind speed (m/s) at the start of sampling, $O_2$ concentration (mg/L), and depth (m) at the start of sampling.

Our analyses revealed variability in spatial and biological variables, reflecting the diversity of ecological requirements and benthic habitats. The most common Cumacea families, Diastylidae and Leuconidae, suggest adaptations to various marine environments. Phylogeographic analysis showed a divergence between specific genetic sequences and two habitat variables: wind speed (m/s) at the start of sampling and $O_2$ concentration (mg/L). This observation may indicate the possibility of varying local adaptations in response to these fluctuating conditions.

These results reinforce the importance of further research into the relationship between Cumacea genetics and global environmental variables to interpret the evolutionary dynamics and adaptation of these deep-sea organisms. This study sheds much-needed light on the acclimatization of invertebrates to climate change, anthropogenic pressures, and marine habitat management, potentially contributing to the evolution of more effective conservation strategies and policies to protect these vulnerable ecosystems.

The *aPhyloGeo* Python package is freely and publicly available on GitHub and PyPi, providing an invaluable tool for future research.

**Keywords** Adaptation, Atlantic, Bioinformatics, Biology, Cumacea, Iceland, Phylogeography

## 1. Introduction

The North Atlantic and Subarctic regions, particularly the Icelandic waters, are of ecological interest due to their diverse water masses and unique oceanographic features [1], [2], [3]. These areas form vital benthic habitats[1] [4] and enhance our understanding of deep-sea ecosystems and biodiversity patterns [3], [5], [6]. The IceAGE project and its predecessors,

[1] Areas at the bottom of oceans, lakes, or rivers, including sediments and organisms that live in them.

BIOFAR and BIOICE, provide invaluable data for studying the impacts of climate change and seabed mining, especially in the Greenland, Iceland, and Norwegian (GIN) seas [7].

Cumacea, a crustacean taxon within Peracarida, are major indicators of marine ecosystem health due to their sensitivity to environmental fluctuations [8] and their contribution to benthic food webs [9]. Despite their ecological importance, the evolutionary history of deep-sea benthic invertebrates remains uncharted, notably in the North Atlantic [10]. Analyzing the genetic and distribution patterns of these deep-sea organisms is crucial for predicting their responses to climate change [10] and anthropogenic pressures [7], while also advancing our understanding of their adaptive mechanisms within deep-sea ecosystems.

Given the urgency of the aforementioned factors, this study aims to analyze the effects of ecological (climatic and environmental) and spatial variables on the genetic variation and adaptation of Cumacea in the Northern North Atlantic. More specifically, we will examine whether there is a genetic adaptation between the genetic structure of a region represented by a partial sequence of the 16S rRNA mitochondrial gene of the Cumacea species included in our analyses and their habitat variables. If so, we will identify which variables show the greatest divergence from a specific segment (i.e., window) of this partial sequence and further investigate the potential associated protein using bioinformatics tools to interpret its biological relevance. Our approach includes confirming various phylogeographic models[2] and updating a Python package (currently in beta), *aPhyloGeo*, to facilitate these analyses.

This paper is organized as follows: Section 2 reviews pertinent studies on the biodiversity and biogeography of deep-sea benthic invertebrates; Section 3 summarizes the aims and contributions of this study, highlighting aspects relating to the conservation and adaptation of marine invertebrates to climate change; Section 4 describes data collection, preprocessing and phylogeographic analyses of partial genetic sequence and habitat variables; Section 4.7 describes the metrics used to evaluate the phylogeographic models; Section 5 presents the results; finally, Section 6 discusses their implications for future research and conservation efforts.

## 2. Related Works

Assessing and quantifying the biodiversity of deep-sea benthic invertebrates has become increasingly important since it was discovered that their species richness may be underestimated [11]. Subsequent research has highlighted the need for large-scale distribution models to interpret the diversity of these organisms across their ecological and evolutionary contexts [12]. Consequently, recent efforts have focused on mapping, managing, and studying the seabed [13]. Advanced technologies, such as acoustic detection are improving our knowledge of benthic ecosystem complexity [13]. Integrating genetic and habitat variables provides a better insight into how ecosystemic, meteorological, and spatial variables influence the genetic variation, distribution, biodiversity, and resilience of deep-sea benthic organisms [14].

However, the relationship between genetics and the environment is complex, involving gene-environment interactions and factors related to natural selection, which makes it difficult to identify clear causal relationships [15]. Additionally, the distinction between the direct and indirect effects of the environment on genetics presents further challenges [16], [17]. The limitations of current methods for measuring genetic and ecological variables, combined with logistical constraints, often limit the scope of such studies [16], [18]. This complexity may explain why the relationship between the environment and genetics of

[2] Phylogeographic models are computational tools that analyze relationships between the genetic structures of populations and their geographic distributions. In our case, by incorporating regional, biological, and atmospheric variables, we can analyze and interpret their impact on the genetic adaptation and spatial patterns of Cumacea species.

Cumacea has been less studied, despite their importance for understanding how deep-sea invertebrates adapt to fluctuating environmental conditions.

## 3. Our Contribution

Our study focuses on the genetic fluctuation of a partial sequence of the 16S rRNA mitochondrial gene in Cumacea communities in response to variations in their habitat, a topic that has been little explored in previous studies [11], [19]. We aim to refine the natural selection hypothesis by identifying specific divergent genetic regions and the potentially associated proteins using bioinformatics tools, such as protein structure modeling and functional annotation databases, to reveal the potential functions that these proteins may have in the adaptation of Cumacea to habitat fluctuations. By linking this partial sequence to habitat variables using robust analytical methods, such as dissimilarity calculations and phylogenetic reconstructions, we can better interpret the selection effects at the molecular level of this Cumacea sequence, which could confer survival advantages in the harsh environments of the Northern North Atlantic. This represents a major advance over previous research, which has often struggled to integrate genetic and biological data in the context of deep-sea invertebrates [14], [20] or has faced difficulties in linking genetics and environment [21], [22].

Furthermore, our genetic and environmental data highlight habitats of high conservation interest that can be considered for establishing marine protected areas [4]. These results are essential for developing informed conservation strategies in the context of climate change. Finally, our study paves the way for further research on other invertebrate species across different geographic regions. By extending this research to diverse environments and taxonomic groups, scientists will be better able to assess the adaptation and resilience of marine invertebrates to changing conditions.

## 4. Materials and Methods

This section describes our data and introduces the main data preprocessing steps, the *aPhyloGeo* software, the distance metrics used, and how the figures were created. A flow chart, constructed with the diagram software draw.io, summarizes this section (see Figure 1).

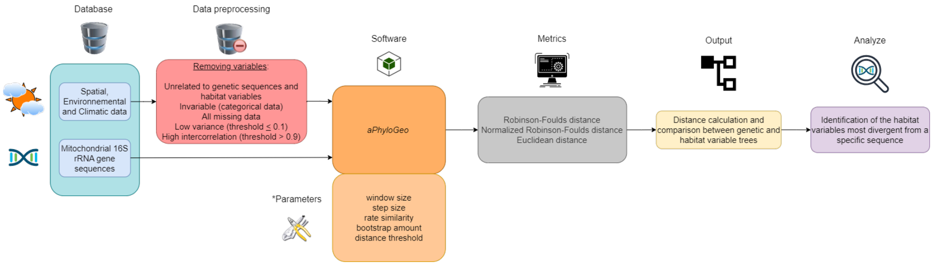

Database
Spatial, Environnemental and Climatic data
Mitochondrial 16S rRNA gene sequences
Data preprocessing
Removing variables:
Unrelated to genetic sequences and habitat variables
Invariable (categorical data)
All missing data
Low variance (threshold ≤ 0.1)
High intercorrelation (threshold > 0.9)
Software
aPhyloGeo
*Parameters
window size
step size
rate similarity
bootstrap amount
distance threshold
Metrics
Robinson-Foulds distance
Normalized Robinson-Foulds distance
Euclidean distance
Output
Distance calculation and comparison between genetic and habitat variable trees
Analyze
Identification of the habitat variables most divergent from a specific sequence

**Figure 1**. *Flow chart summarizing the Materials and Methods section workflow. Six different colors highlight the blocks. The first block (blue) represents our database. The second block (in red) is data preprocessing, which consists of deleting certain variables. The third and fourth blocks (orange) implement the* aPhyloGeo *software and its parameters for our phylogeographic analyses. The fifth block (gray) applies distance metrics to the genetic and habitat variable trees produced. The sixth block (yellow) calculates and compares distance metrics between genetic and habitat variable trees. The seventh block (purple) identifies the most divergent habitat variables of a specific region of the partial sequence of the 16S rRNA mitochondrial gene based on the results of tree comparisons. *See YAML files on GitHub for more details on these parameters.*

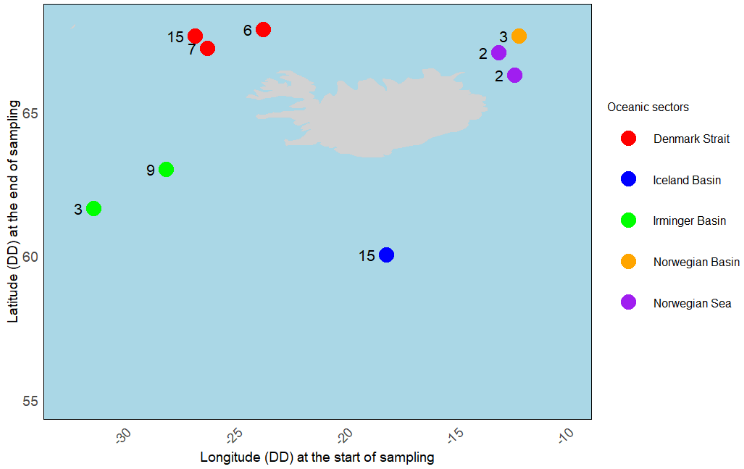


**Figure 2.** *Distribution map of Cumacea specimens included in our analyses according to the oceanic sector where they were sampled. The grey area represents Iceland, and the number next to the point is the number of specimens found at that sampling point.*

### *4.1. Description of the data*

The study area is located in the Northern part of the North Atlantic, including the Denmark Strait, the Iceland Basin, the Irminger Basin, and the Norwegian Basin and Sea (see Figure 2). The specimens analyzed were collected as part of the IceAGE project (Icelandic marine Animals: Genetic and Ecology; Cruise ship M85/3 in 2011), which focused on the deep continental slopes and abyssal waters around Iceland [7]. The sampling period for the included specimens was from August 30 to September 22, 2011, and they were collected at depths ranging from 315.9 m to 2567.7 m. Detailed protocols concerning the sampling plan, sample processing, DNA extraction steps, PCR amplification, sequencing, and aligned DNA sequences are available in [3].

### *4.2. Data preprocessing*

We used data from the article [3], the IceAGE project, and related data from the BOLDSystem database, as described in [3]. Given these databases' enormous variety of variables, we applied a selective reduction procedure. Variables with no variability (categorical data) were excluded from our study, for which all data were missing and were not linked to genetic sequences or spatial, environmental, and climatic variables. Out of the 495 available in the IceAGE dataset, we considered 62 specimens for which partial 16S rRNA mitochondrial gene sequences were available.

Next, we calculated the variance ($S^2$) using the $var()$ function in RStudio Desktop 4.3.2 for each of the selected variables (numerical and categorical). This step aimed to eliminate variables with low variation, as they are unlikely to provide essential data for analysis. We set a variance threshold of ≤ 0.1 to exclude uninformative variables. The latter retains variables whose variability is reasonably sufficient for our analyses while rejecting those with little variation. Only water salinity was eliminated based on this criterion ($S^2_{\text{Salinity}} = 0.02146629$). The formula (see Equation Equation 1) and code (Program 1) used to calculate the variance of our final variables, available in the data file on GitHub, are provided below:

$$S^2 = \frac{\sum_{i=1}^{n} (x_i - | \, x)^2}{n-1} \tag{1}$$

where $S^2$ is the variance of the variable, $x_i$ represents each value of the variable, $| \, x$ is the average of the values for this variable, and $n$ is the number of values for this variable in the dataset.

```r
# Import data from the CSV file
Data <- read.csv(file="Final_Data_Article.csv", header=TRUE, sep=";")

# Define a function to calculate entropy for categorical variables
calculate_entropy <- function(x) {
  # Calculate the frequency of each category
  freq_table <- table(x)

  # Calculate probabilities
  probabilities <- freq_table / sum(freq_table)

  # Calculate entropy using the probabilities
  entropy_value <- -sum(probabilities*log(probabilities), na.rm=TRUE)

  return(entropy_value)
}

# Calculate variance
variances <- sapply(Data, function(x) {
  # Check if the column is numeric
  if (is.numeric(x)) {
    # Compute variance, excluding NA values
    var(x, na.rm=TRUE)
  } else if (is.factor(x) || is.character(x)) {
    # If the column is categorical, compute entropy
    calculate_entropy(x)
  } else {
    NA  # Return NA for other types of columns
  }
})

# Display variances/entropies
print(variances)
```

**Program 1**. *RStudio script to calculate the variance of each numerical and categorical variables in our final dataset.*

We calculated the Pearson correlation ($r$) between variables using the $cor()$ function in RStudio Desktop 4.3.2. Variables (numerical) exhibiting strong correlations with each other (threshold > 0.90) were removed to avoid repetition and guarantee variable independence. We considered the threshold of > 0.90 to be an adequate compromise between preserving properties for our analyses and eliminating the repetition of information in our data. Since we have three missing data for $O_2$ concentration (mg/L), we have used the “pairwise.complete.obs method”. This method calculates the Pearson correlation matrix using all accessible pairs of observations, even if some data are missing. Using the above threshold, four variables were discarded: latitude (DD) at the start of sampling (Lat_start_end: $r = 0.9996658$), longitude (DD) at the end of sampling (Long_start_end: $r = 0.9999979$), depth (m) at the end of sampling (Depth_start_end: $r = 0.9998579$) and wind direction at the start of sampling (WindD_start_end: $r = 0.9752331$). The decision to remove these variables was based on their variance ($S^2$) value: $S^2_{\text{Lat\_start}} = 10.03077$ and $S^2_{\text{Lat\_end}} = 10.71335$; $S^2_{\text{Long\_start}} = 30.47940$ and $S^2_{\text{Long\_end}} = 30.47574$; $S^2_{\text{Depth\_start}} = 776437.1$ and $S^2_{\text{Depth\_end}} = 775394.7$; $S^2_{\text{WindD\_start}} = 2.405077$ and $S^2_{\text{WindD\_end}} = 4.482285$. The formula (see Equation Equation 2) and code (Program 2) used to calculate the Pearson correlation coefficient between our final numerical variables are shown below:

$$r = \frac{\sum_{i=1}^{n}(x_i - \mid x)(y_i - \mid y)}{\sqrt{\sum_{i=1}^{n}(x_i - \mid x)^2 \sum_{i=1}^{n}(y_i - \mid y)^2}} \tag{2}$$

where $r$ is the Pearson correlation coefficient between two variables, $x_i$ are the values of the variable $x$, $y_i$ are the values of the variable $y$, $\mid x$ and $\mid y$ are respectively the averages of the two variables, and $n$ is the number of values of the two variables in the dataset.

This selection of variables and data resulted in a table containing 62 rows ($n = 62$) and 16 columns (number of variables).

```r
# Import data
Data <- read.csv(file="Final_Data_Article.csv", header=TRUE, sep=";")

# Select numeric columns only from the dataset
numeric_Data <- Data[sapply(Data, is.numeric)]

# Calculate Pearson correlation matrix
correlation_matrix <- cor(numeric_Data, use="pairwise.complete.obs")

# Display correlation matrix
print(correlation_matrix)
```

**Program 2**. *RStudio script to calculate the Pearson correlation coefficient between all the numerical variables in our final dataset.*

## *4.3. Selected variables in the IceAGE database*

### 4.3.1. *Spatial data:*

- The latitude at the end of sampling (see Figure 3a) and longitude at the start of sampling (see Figure 3b), both in decimal degrees (DD), as they are intimately linked to the environmental gradients and historical mechanisms modeling genetic heterogeneity [23].
- The five oceanic sectors across the seas around Iceland (see Figure 2): the Denmark Strait ($n = 28$), the Iceland Basin ($n = 15$), the Irminger Basin ($n = 12$), the Norwegian Sea ($n = 4$), and the Norwegian Basin ($n = 3$).

### 4.3.2. *Environmental data:*

- Depth (m) at the start of sampling (see Figure 3c), as well as water temperature (°C) (see Figure 3e), and $O_2$ concentration (mg/L) (see Figure 3f), as these are vital elements of the marine ecosystem that have an impact on the distribution and evolutionary acclimatization of marine species [24], [25].
- Since the sedimentary characteristics directly influence the distribution of Cumacea [3], they were included in our data. They are divided into six ecological niche categories: mud ($n = 30$), sandy mud ($n = 15$), sand ($n = 9$), forams ($n = 3$), muddy sand ($n = 3$), and gravel ($n = 2$).

### 4.3.3. *Climatic data:*

Wind speed (m/s) at the start (see Figure 3d) and end of sampling and wind direction at the end of sampling were also included, giving the contribution of wind to benthic ecosystem dynamics and the restructuring of species distribution by wind currents and sediment transport [26], [27], [28]. The wind direction at the end of sampling comprises eight orientations: south (S, $n = 15$), southwest (SW, $n = 15$), northeast (NE, $n = 9$), west-southwest (WSW, $n = 7$), southeast (SE, $n = 6$), north-northwest (NNW, $n = 5$), south-southeast (SSE, $n = 3$), and east (E, $n = 2$).

## *4.4. Selected variables in the BOLDSystem database*

### 4.4.1. *Taxonomic data:*

The family, genus, and scientific name of the Cumacea were integrated into our data to study evolutionary relationships and genetic variation to habitat and acclimatization variables among the specimens. These comprise seven families (see Figure 4): Diastylidae ($n = 21$), Lampropidae ($n = 13$), Leuconidae ($n = 12$), Astacidae ($n = 7$), Bodotriidae ($n = 4$), Ceratocumatidae ($n = 3$), and Pseudocumatidae ($n = 2$). A total of 20 Cumacea species were included in our dataset (see Figure 4). We have also included the sample identity (Sampleid) so that each specimen remains unique. Some specimens were only identified to family ($n = 1$) or genus ($n = 4$).

### 4.5. *Selected variables from article C. Uhlir et al. [3]*

#### 4.5.1. *Other environmental data:*

The habitat and water mass of the sampling points were the only environmental variables taken directly from Table 1 of [3], as they can provide insight into how they may affect Cumacea genetic diversity and the acclimatization of these species in the GIN seas around Iceland. Thus, the water masses definitions, as described in [3], were used as a reference: Arctic Polar Water (APW, $n = 15$), Iceland Sea Overflow Water (ISOW, $n = 15$), North Atlantic Water (NAW, $n = 9$), warm Norwegian Sea Deep Water (NSDWw, $n = 8$), Arctic Polar Water/ Norwegian Sea Arctic Intermediate Water (APW/NSAIW, $n = 7$), Labrador Sea Water (LSW, $n = 3$), cold Norwegian Sea Deep Water (NSDWc, $n = 3$), and Norwegian Sea Arctic Intermediate Water (NSAIW, $n = 2$) (see Figure 5). In terms of habitat, we considered the three categories used in [3]: Deep Sea ($n = 38$), Shelf ($n = 15$), and Slope ($n = 9$) (see Figure 6).

#### 4.5.2. *Genetic data:*

The aligned partial DNA sequence of the 16S rRNA mitochondrial gene was included, as this region is standard in phylogeny and phylogeography studies [29] and sufficiently conserved over time to guarantee exact alignments between different species [30]. We examined 61 of the 306 aligned DNA sequences used for phylogeographic analyses by [3]. As some specimens have their DNA sequence duplicated, or even quadruplicated with a difference of one or two nucleotides, the longest-aligned DNA sequence of each specimen was retained. The "ICE1-Dia004" specimen is the only one whose sequence (not aligned) was taken from the BoldSystem database, as it was absent from the [3] aligned DNA database.

### 4.6. aPhyloGeo *software*

We used the cross-platform Python software *aPhyloGeo*, developed by the Tahiri Lab team, for our phylogeographic analyses. This software is designed to analyze phylogenetic trees using ecological and spatial variables (Program 3) to interpret the evolution of species under different environmental conditions [31], [32], [33].

This software was selected for our analysis as it is the first phylogeographic tool capable of establishing similarity or dissimilarity between the genetics of species and environmental, climatic, and spatial variables [31], [32], [33], which is precisely the objective of our study. The *aPhyloGeo* software offers several key functionalities:

1. **Phylogenetic tree evaluation**: The software identifies the evolutionary relationships among species based on their genetic sequences [31], [32], [33], which is essential for interpreting phylogeographic models that connect species evolution to their spatial distribution and biological and meteorological contexts.
2. **Ecological and regional dissimilarity analysis**: The software highlights divergence and convergence between genetic sequences and habitat variables [31], [32], [33], enabling the assessment of the influence of these variables on genetic fluctuations and the evolutionary history of Cumacea species.
3. **Evaluation of genetic diversity**: The software quantifies genetic heterogeneity, facilitating the identification of potential evolutionary processes (e.g., mutation, speciation, and genetic drift) and local adaptations.

The *aPhyloGeo* Python package is freely and publicly available on GitHub, and is also available on PyPi, to facilitate complex phylogeographic analyses. The software process has three main stages:

```
if __name__ == "__main__":

    # Load parameters
    Params.load_from_file(Params.reference_yaml_filepath)

    # Load the sequence file
    sequence_file = utils.loadSequenceFile(
                    Params.reference_gene_filepath)

    # Create an AlignSequences object
    align_sequence = AlignSequences(sequence_file)

    # Load variable data
    variable_data = pd.read_csv(Params.file_name)

    # Perform the alignment of sequences
    alignments = align_sequence.align()

    # Generate genetic trees based on aligned sequences
    geneticTrees = utils.geneticPipeline(alignments.msa)

    # Create a GeneticTrees object
    trees = GeneticTrees(trees_dict=geneticTrees,
                        format="newick")

    # Generate variable trees
    variableTrees = utils.climaticPipeline(variable_data)

    # Filter the results based on the generated trees
    utils.filterResults(variableTrees,
                       geneticTrees,
                       variable_data)
```

**Program 3**. *Main script for tutorial using the aPhyloGeo package.*

1. **The first step** was to collect DNA sequences from Cumacea of sufficient quality for the needs of our results [31], [32], [33]. In this study, 62 Cumacea specimens were selected to represent 62 partial sequences of the 16S rRNA mitochondrial gene. We then included, from our database, two climatic variables, namely wind speed (m/s) at the start and end of the sampling; three environmental variables, such as depth (m) at the start of sampling, water temperature (°C), and $O_2$ concentration (mg/L); and two geographic variables, latitude (DD) at the end of sampling and longitude (DD) at the start of sampling.
2. **The second step** was to generate trees from genetic, biological, spatial, and meteorological data. For spatial variables, the Neighbor-Joining method[3] was applied between each pair of Cumacea from distinct spatial conditions to produce a symmetrical square matrix and build the spatial tree from this matrix [33]. Each geographic variable generates a distinct phylogenetic tree. If there are $m$ windows from the genetic sequences, there will be $m$ geographic trees. The same approach was applied to biological, meteorological, and genetic data.

For the genetic data, phylogenetic reconstruction was repeated to build genetic trees based on 62 partial sequences of the 16S rRNA mitochondrial gene, considering only data within a window that progresses along the alignment [31], [32], [33]. Each window in the alignment will give a genetic tree. If there are $n$ windows from the sequences, there will be $n$ phylogenetic trees. This displacement can vary according to the steps and the size of the window defined by the user (their length is determined by the number of base pairs (bp)) [31], [32], [33].

In our case, we set up the *aPhyloGeo* software as follows: $pairwiseAligner$ for sequence alignment; $\mathrm{Hamming\ distance}$ to measure simple dissimilarities between sequences; $\mathrm{Wider\ Fit\ by\ elongating\ with\ Gap(starAlignment)}$ algorithm takes alignment gaps into ac-

[3] It is a method used to construct phylogenetic trees using distance matrices.

count, which is often mandatory in the case of major deletions or insertions in the sequences; $\text{windows\_size}$: 10 nucleotide (nt); and finally, $\text{step\_size}$: 1 nt. The last two configurations imply that for each 10 nt window, a phylogenetic tree is produced using the 10 nt sequence of each Cumacea. Next, the window is moved by 1 nt, creating a new tree with the next 10 nt, and so on until the end of the alignment. Genetic trees will be stored in an object called $T_1$, while spatial and ecological trees will be stored in another object called $T_2$.

1. **The third step** is to compare the genetic trees constructed in each sliding window with the ecosystemic, atmospheric, and regional trees using the Robinson-Foulds distance [34], normalized Robinson-Foulds distance and Euclidean distance. These contribute to understanding the correspondence between Cumacea genetic sequences and their habitat variables. The approach also takes bootstrapping into account [31], [32], [33]. The results of these metrics were obtained using the functions $robinson_foulds(tree1, tree2)$ and $euclidean_dist(tree1, tree2)$ from the ***aPhyloGeo*** software and were organized by the main function (Program 3). Those for the normalized Robinson-Foulds distance were obtained with the function $robinson_foulds(tree1, tree2)$ (see the last line of code in Program 4). The result of the metrics indicates which variables show the greatest genetic divergence according to the magnitude of the metric distances (see figures Figure 7 and Figure 8).

A sliding-window approach enables the precise location of subtle sequences with high rates of genetic divergence [31], [32], [33]. This method involves moving a fixed-size window over the alignment of genetic sequences. This allows genetic trees to be built for each part of the aligned sequences, depending on the size of the window and the step size. It therefore makes it possible to recognize changes in evolutionary relationships along the partial sequence region of the 16S rRNA mitochondrial gene of Cumacea species. This method is essential to determine whether this region of the Cumacea genome can be affected by certain ecological or spatial variables in their habitat (see Figure 7 and Figure 8).

## 4.7. *Metrics*

In our phylogeographic analysis, we employed three distance metrics to quantify differences between phylogenetic trees and habitat trees, as well as to evaluate dissimilarities between genetic sequences and the associated environmental variables. This approach allowed for a detailed examination of the evolutionary patterns of Cumacea communities across varying ecological conditions.

The following section provides a detailed description of the three distance functions referenced in the second and third steps of Section 4.6, offering a more rigorous examination of their role in the analysis.

### 4.7.1. *Robinson-Foulds distance:*

The Robinson-Foulds (RF) distance [34] calculates the distance between genetic trees built in each sliding window ($T_1$) and the variable trees ($T_2$) (see the list in the first step of the Section 4.6) [35], [36]. This metric is used to evaluate the topological differences between the two sets of trees by measuring the minimum number of elementary operations (merging and splitting nodes) required to transform one tree (genetic) into another (variable habitat) (see Equation Equation 3 and Program 4). A high distance of a specific window in RF distance analysis may imply that the habitat variable has little to no impact on the evolution of this particular DNA sequence and that the fluctuation of this variable might not explain the genetic divergence observed.

$$\mathrm{RF}(T_1, T_2) = \mid \Sigma(T_1) \Delta \Sigma(T_2) \mid \tag{3}$$

where $\mathrm{RF}(T_1, T_2)$ is the Robinson-Foulds distance between the two sets of trees, $\Sigma(T_1)$ and $\Sigma(T_2)$ are the sets of divisions in trees $T_1$ and $T_2$ and $\Delta$, the difference between these two sets.

```python
def robinson_foulds(tree1, tree2):

    # Initialize the Robinson-Foulds distance
    rf = 0

    # Convert trees from Newick format to ete3.Tree objects
    tree1_newick = ete3.Tree(tree1.format("newick"), format=1)
    tree2_newick = ete3.Tree(tree2.format("newick"), format=1)

    # Calculate the Robinson-Foulds distance
    rf, rf_max, common_leaves = tree1_newick.robinson_foulds(
                                tree2_newick,
                                unrooted_trees=True)

    # If there are no common leaves, set the RF distance to 0
    if len(common_leaves) == 0:
        rf = 0

    # Return the RF distance and its normalized value
    return rf, rf / rf_max
```

**Program 4**. *Python script for calculating the Robinson-Foulds Distance using the ete3 package in the aPhyloGeo package. The Newick format represents the phylogenetic and variable trees in text form.*

### 4.7.2. *Normalized Robinson-Foulds distance:*

The normalized Robinson-Foulds (nRF) distance scales the RF distance to account for the size variations in the trees (number of clades; i.e., a group of species with a common origin), allowing a more equitable comparison. It scales the distance to a range between 0 and 1. In our context, the distance has been normalized by $2n - 6$, where $n$ represents the number of taxa (see Equation Equation 4 and the last line of code in Program 4).

Since the size of environmental trees constructed with $O_2$ concentration data (mg/L) differs from that of other variables due to missing data, this nRF distance allows its dissimilarity with genetic trees to be compared more fairly [33], [36]. It reveals the relative influence of $O_2$ concentration (mg/L) on Cumacea phylogenetic relationships, independent of tree size [33], [36]. A high distance of a specific window in the nRF distance analysis suggests that we cannot conclude that there is a correlation between this DNA sequence and the variable. It may indicate a topological dissimilarity between the habitat variable trees and the genetic trees at that position in the DNA sequence alignments.

$$\mathrm{RF}_{\mathrm{norm}}(T_1, T_2) = \frac{\mid \Sigma(T_1)\Delta\Sigma(T_2) \mid}{\mid \Sigma(T_1) \mid + \mid \Sigma(T_2) \mid} \tag{4}$$

where $\mathrm{RF}_{\mathrm{norm}}(T_1, T_2)$ is the normalized Robinson-Foulds distance between the two sets of trees, $\Sigma(T_1)$ and $\Sigma(T_2)$ are the sets of divisions in trees $T_1$ and $T_2$ and $\Delta$, the difference between these two sets.

### 4.7.3. *Euclidean distance:*

The Euclidean distance calculates the straight-line distance between two sets of points in a multidimensional space, which designates the length divisions of the two sets of trees ($T_1$ and $T_2$). It is used to evaluate the degree of divergence or similarity of topologies between two respective sets of trees (see Equation Equation 5 and Program 5). A high distance of a specific window in the Euclidean distance analysis suggests evolutionary divergences between members of the Cumacea communities at the level of this DNA sequence and the variation of the habitat variable (see Figure 7d and Figure 8d). In other words, the habitat variable may not have a dominant contribution to the evolution of this specific sequence of Cumacea communities.

```python
def euclidean_dist(tree1, tree2):

    # Initialize the Euclidean distance
    ed = 0

    # Create a TaxonNamespace object to handle taxon information
    tns = dendropy.TaxonNamespace()

    # Load the first tree into a dendropy Tree object
    tree1_tc = dendropy.Tree.get(data=tree1.format("newick"),
                                 schema="newick",
                                 taxon_namespace=tns)

    # Load the second tree into a dendropy Tree object
    tree2_tc = dendropy.Tree.get(data=tree2.format("newick"),
                                 schema="newick",
                                 taxon_namespace=tns)

    # Encode the bipartitions of both trees
    tree1_tc.encode_bipartitions()
    tree2_tc.encode_bipartitions()

    # Calculate the Euclidean distance
    ed = dendropy.calculate.treecompare.euclidean_distance(
                                                tree1_tc,
                                                tree2_tc)

    return ed
```

**Program 5**. *Python script for calculating the Euclidean distance using the ete3 and the dendropy packages in the aPhyloGeo package. The Newick format represents the phylogenetic and variable trees in text form.*

$$d_{\text{Euclidean}}(T_1, T_2) = \sqrt{\sum_{i=1}^{n} (T1_i - T2_i)^2} \tag{5}$$

where $d_{\text{Euclidean}}(T_1, T_2)$ is the Euclidean distance between the two sets of trees, and $T1_i$ and $T2_i$ represent the respective divisions of trees $T_1$ and $T_2$ for each $i$-th division.

Interestingly, Euclidean distance is more sensitive to the subtle tree topology, making it suitable for identifying detailed correlations between genetic fluctuations and those of habitat variables [37]. It can therefore be used to study fine divergences between trees, enabling nuanced identification of the effects of habitat variables on the genetic structure of species [37]. As for the Robinson-Foulds distance (normalized or not), although widely applied in evolutionary biology, it is less sensitive to slight topological dissimilarities, making it less accurate for identifying fine correlations between genetics and habitat variables due to its structural nature [38], [39].

### 4.8. *Creating Figures*

Figure 3, Figure 4, Figure 7 and Figure 8 were made with Python 3.11, while Figure 2, Figure 5 and Figure 6 were made with RStudio Desktop 4.3.2.

## 5. Results

The violin diagrams shown in Figure 3 are used to display summary statistics similar to box plots, showing medians (white lines), interquartile ranges (thickened black bars), and the rest of the distributions (thin black lines), except the “extreme” points. Wider areas indicate a greater probability of the variables taking a given value. They summarize the distribution of spatial (latitude at the end of sampling and longitude at the start of sampling, both in DD), atmospheric (wind speed (m/s) at the start of sampling), and ecosystemic (depth (m) at the start of sampling, water temperature (°C), and $O_2$ concentration (mg/L)) data. These

**Table 1**. *Table summarizing key statistics such as mean, median, standard deviation (Std Dev), 1st quartile (Q1) and 3rd quartile (Q3) of biological (depth (m) at the start of sampling, water temperature (°C), and $O_2$ concentration (mg/L)), spatial (latitude (DD) at the end of sampling and longitude (DD) at the start of sampling) and atmospheric (wind speed (m/s) at the start of sampling) variables for our phylogeographic analyses.*

| Attributes | Statistics | Mean | Std Dev | Q1 | Median | Q3 | Min | Max |
|---|---|---|---|---|---|---|---|---|
| Lat end (DD) | | 64.75 | 3.27 | 61.65 | 67.15 | 67.63 | 60.05 | 67.86 |
| Long start (DD) | | -23.12 | 5.52 | -26.77 | -26.21 | -18.14 | -31.35 | -12.16 |
| Depth start (m) | | 1412.57 | 881.16 | 579.10 | 1574.70 | 2504.70 | 315.90 | 2567.70 |
| Watertemp ground (°C) | | 1.45 | 1.73 | 0.07 | 0.71 | 2.65 | 0.85 | 4.28 |
| O2 saturation ground (mg/L) | | 271.88 | 18.11 | 258.39 | 278.77 | 290.90 | 245.53 | 292.97 |
| Windspeed start (m/s) | | 6.26 | 2.16 | 5.25 | 6.00 | 7.00 | 2.00 | 11.00 |

diagrams are essential for understanding habitat conditions and highlighting the variables that can potentially influence genetic fluctuation and adaptability in Cumacea. In Table 1 and Figure 3, the variables are designated by their names from the IceAGE database, except for latitude (DD) at the end of sampling and longitude (DD) at the start of sampling, for which the term "dec" has been removed at the end to avoid confusion.

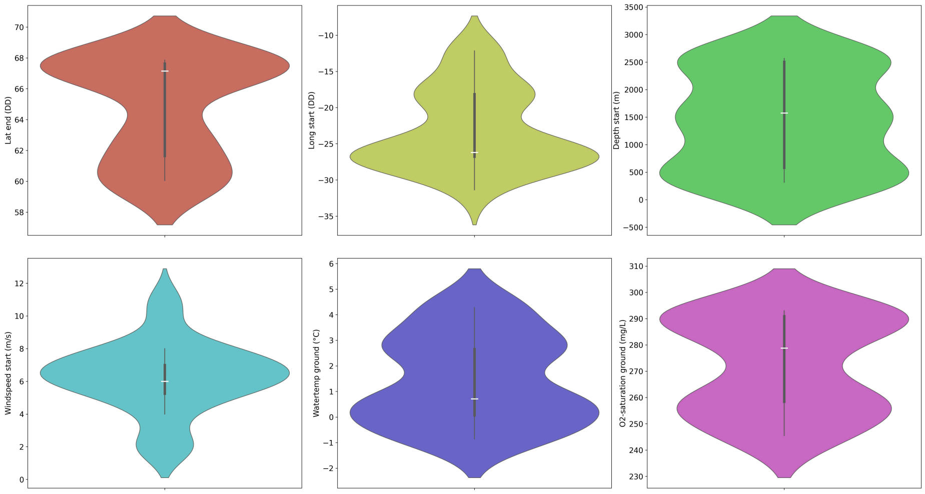


**Figure 3**. *Violin diagrams of two regional, one atmospheric, and three ecosystemic variables that provide essential information about the ecological and meteorological conditions of Cumacea habitats. a) Latitude (DD) at the end of sampling (red) suggest that the specimens come from two dominant latitudinal (DD) regions (around 61.65 DD and 67.63 DD); b) Longitude (DD) at the start of sampling (yellow) implies that the specimens come from two dominant longitudinal (DD) regions (around -26.77 DD and -18.14 DD); c) Depth (m) at the start of sampling (green) suggest that the specimens were mainly collected and concentrated at three different depths (m) (around 500 m, 1500 m and 2500 m); d) Wind speed (m/s) at start of sampling (light blue) indicate stable the wind conditions (m/s) at the start of sampling (around 6.00 m/s); e) Water temperature (°C) (dark blue) suggest that the specimens were mostly collected and concentrated at two different water temperatures (°C) (around 0.07 °C and 2.66 °C); f) $O_2$ concentration (mg/L) (pink) implies that the specimens were primarily collected and concentrated at two different $O_2$ concentrations (mg/L) (around 258.39 mg/L and 290.90 mg/L).*

Our results revealed variability in most habitat variables, as shown in Figure 3. For instance, the median of the latitude at the end of sampling (67.15 DD; Table 1) is higher than the mean (64.75 DD; Table 1), showing an asymmetric distribution skewed towards lower values. This trend is also observed for depth (m) at the start of sampling (Median: 1574.70 m; Mean: 1412.57 m; see Figure 3c and Table 1) and $O_2$ concentration (mg/L) (Median: 278.77 mg/L; Mean: 271.88 mg/L; see Figure 3f and Table 1). The bimodal shape of the latitude distribution curve suggests that the specimens came from two dominant latitudinal regions at the end of sampling (around 61.65 DD and 67.63 DD; see Figure 3a and Table 1). This bimodality is also observed in longitude (DD) at the start of sampling (around -26.77 DD and -18.14 DD; see Figure 3b and Table 1), as well as for water temperature (°C) (around 0.07 °C and 2.66 °C; see Figure 3e and Table 1), and $O_2$ concentration (mg/L) (around 258.39 mg/L and 290.90 mg/L; see Figure 3f and Table 1).

The median of the longitude (DD) at the start of sampling (-26.21 DD; Table 1) is lower than the mean (-23.12 DD; Table 1), indicating asymmetry on the higher sides (see Figure 3b), as does the water temperature (°C) (Mean: 1.45 °C; Median: 0.71 °C; see Figure 3e and Table 1). Unlike all the other diagrams in Figure 3, the curve of the depth (m) at the start of sampling (see Figure 3c) has a multimodal shape with three prominent peaks, suggesting that the specimens were mainly collected and concentrated at three different depths (around 500 m, 1500 m and 2500 m; see Figure 3c).

The mean (6.26 m/s; Table 1) and median of wind speed (m/s) at the start of sampling are fairly similar, with a high density of data around the median (6.00 m/s; see Figure 3d and Table 1). This suggests stable wind conditions (m/s) at the start of sampling. The key statistics and the figure for the wind speed (m/s) at the end of sampling are available in the $img$ file on GitHub. The standard deviation of water temperature (°C) is relatively high (1.73 °C; Table 1) compared to the mean (1.45 °C; Table 1), suggesting acclimatization of Cumacea to a variety of habitat temperatures (-0.85 °C – 4.28 °C; see Figure 3e and Table 1). The range of data for $O_2$ concentration (mg/L) shows some variability (245.53 mg/L – 292.97 mg/L; see Figure 3f and Table 1) in the environmental conditions. This reflects a diversity of requirements in terms of $O_2$ concentration (mg/L), with Cumacea potentially affected by the heterogeneity of biogeochemical cycles, such as photosynthesis, respiration, and organic decomposition, which affect depth-dependent dissolved $O_2$ concentration (mg/L).

The distribution and diversity of the various Cumacea species and family found are shown in Figure 4. It shows that the most represented species are *Leptostylis ampullacea* (14.1%) and *Leucon pallidus* (12.5%). In contrast, species like *Bathycuma brevirostre* and *Styloptocuma gracillimum* are less represented (1.6%), implying that some species may have restricted ecological niches or face ecological forces that limit their distribution. The dominance of certain species (such as *Leptostylis ampullacea* and *Leucon pallidus*) suggests that they may have adaptive traits that enable them to make the most of the accessible resources, resist interspecific competition, or survive in fluctuating ecosystemic conditions, aligns with our study's aim of relating genetic adaptation to habitat characteristics.

The figure above supports the objective of our study by showing the distribution of the different Cumacea families in the various water masses (see Figure 5). The Diastylidae family, for example, is the most common in all water masses (turquoise color in Figure 5), testifying to its resilience and ecological adaptability to a wide variety of habitat conditions, reminiscent of the dominance of *Leptostylis ampullacea* which belongs to the Diastylidae family (see Figure 4, 14.1%).

The distribution of the different Cumacea families according to the type of habitat where they were collected during sampling is shown in Figure 6. The deep-sea habitats show the greatest diversity of families, mainly Diastylidae and Lampropidae, suggesting they are well acclimatized to deep-sea conditions. In contrast, the slope has the lowest diversity,

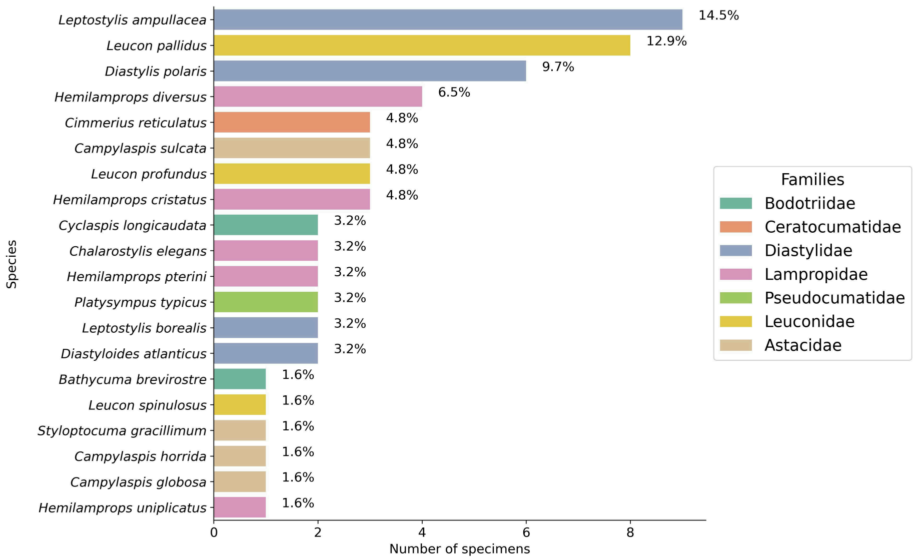


**Figure 4**. *Cumacea frequency distribution by species and family. The percentages (%) displayed above the bars indicate the relative abundance of each species in our dataset. Unlike less common species, those that are abundant (such as* Leptostylis ampullacea *and* Leucon pallidus*) may have adaptive characteristics that enable them to exploit resources more easily, resist interspecific competition or withstand changing biological conditions.*

with Diastylidae again the most dominant, implying that some Cumacea species have fewer ecological niches or are less adapted to this habitat. Although less diverse than the deep sea, the shelf is dominated by Leuconidae, indicating that this family may be specifically well-acclimated to this habitat. These patterns imply that certain Cumacea families, such as the Diastylidae, Lampropidae, Leuconidae, Pseudocumatidae, and Astacidae, have developed distinct adaptations (physiological, behavioral, or morphological) to remain in particular ecological niches, reflecting the impact of habitat conditions on the genetic distribution of Cumacea.

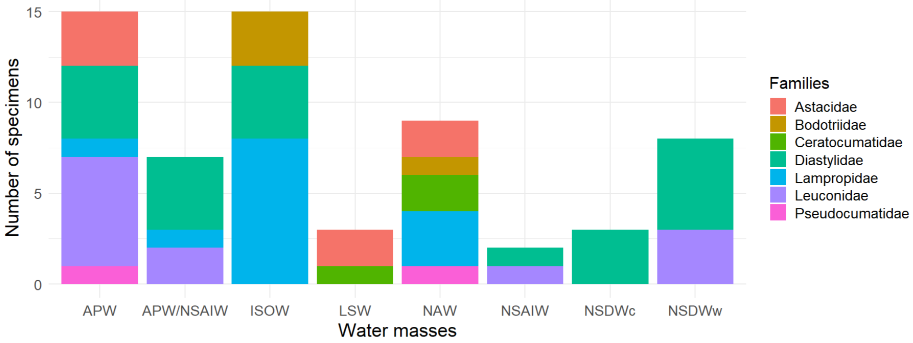


**Figure 5**. *Distribution of Cumacea families by water mass. This histogram represents the frequency of occurrence of the different Cumacea families, classified according to the water mass in which they were collected. Eight water mass categories are represented: Arctic Polar Water (APW), Arctic Polar Water/ North Sub-Arctic Intermediate Water (APW/NSAIW), Iceland Scotland Overflow Water (ISOW), Labrador Sea Water (LSW), North Atlantic Water (NAW), North Sub-Arctic Intermediate Water (NSAIW), cold North Sub-Atlantic Deep Water (NSDWc), and warm North Sub-Atlantic Deep Water (NSDWw). The presence of the Diastylidae (turquoise) family in the majority of water bodies (APW, APW/NSAIW, ISOW, NSAIW, NSDWc, and NSDWw) accentuates the resilience and ecological acclimatization of this family to various ecological niches and conditions.*

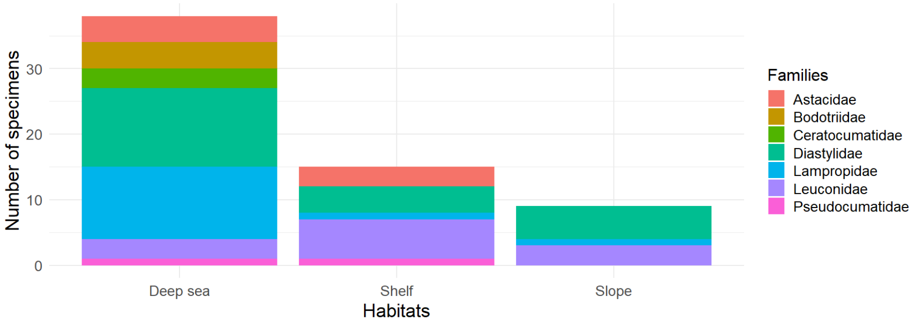


**Figure 6**. *Distribution of Cumacea families by habitat. This histogram represents the frequency of occurrence of the different Cumacea families, classified according to the habitat in which they were collected. Three habitat categories are represented: Deep Sea, Shelf, and Slope. The presence of Cumacea families in more than one habitat, such as Diastylidae (turquoise), Lampropidae (blue), Leuconidae (purple), Pseudocumatidae (pink), and Astacidae (red), may indicate the development of adaptations, whether morphological, physiological or behavioral, that could favor their persistence in these habitats.*

The divergence between specific genetic sequences and two variables, one climatic (wind speed (m/s) at the start of sampling) and the other environmental ($O_2$ concentration (mg/L)) is presented in Figure 7 and Figure 8. All the variables given in the first step of the Section 4.6 were analyzed and the configuration parameters are available in the $scripts$ Python file on GitHub. However, only these two show the most interesting rate of divergence. Using the three metrics mentioned in the Section 4.7, we noticed that the Euclidean distance is particularly sensitive to our data, manifesting considerable sequence variation at the position in MSA 560-569 amino acids (aa) (Euclidean distance: 0.85; see Figure 7d) and 1210-1219 aa (Euclidean distance: 1.23; see Figure 8d). The fluctuations in wind speed (m/s) at the start of sampling and in $O_2$ concentration (mg/L) do not appear to explain the variations in these two specific windows. This could indicate the absence of directional selection in these sequences due to these habitat variables, local selective pressures not considered in our analysis, or other evolutionary factors (e.g., genetic drift or biotic interactions) predominate over these two variables concerning these two sequences. On the other hand, this may suggest that these two variables could potentially influence the divergent (i.e., genetic diversification) rather than a convergent adaptation of these Cumacea, reflecting unique evolutionary responses to these specific ecological pressures. These results are consistent with the aim of our study, which is to identify the Cumacea genetic region that diverges most as a function of habitat variables, to determine whether this is due to divergent local adaptation or other evolutionary processes.

These results provide important insight into the genetic adaptation of Cumacea to their environment. These results need to be analyzed in greater depth to certify their involvement, especially in contrast with [3], which investigated similar topics of environmental and climatic effects on Cumacea distribution and genetics. The *aPhyloGeo* package is still in the process of being updated.

## 6. Conclusion

This study examines the effects of meteorological, regional, and ecosystemic variables on the genetics of Cumacea in the waters surrounding Iceland. Our main objective is to determine whether there is a discrepancy between the genetic informations of the partial 16S rRNA mitochondrial gene sequence (i.e. a window) of Cumacea species and their habitat variables. In addition to data distribution representations (see Figure 3, Figure 4, Figure 5 and Figure 6), DNA sequence analyses, using the *aPhyloGeo* software, have identified specific genetic windows that diverge from atmospheric and biological variables such as

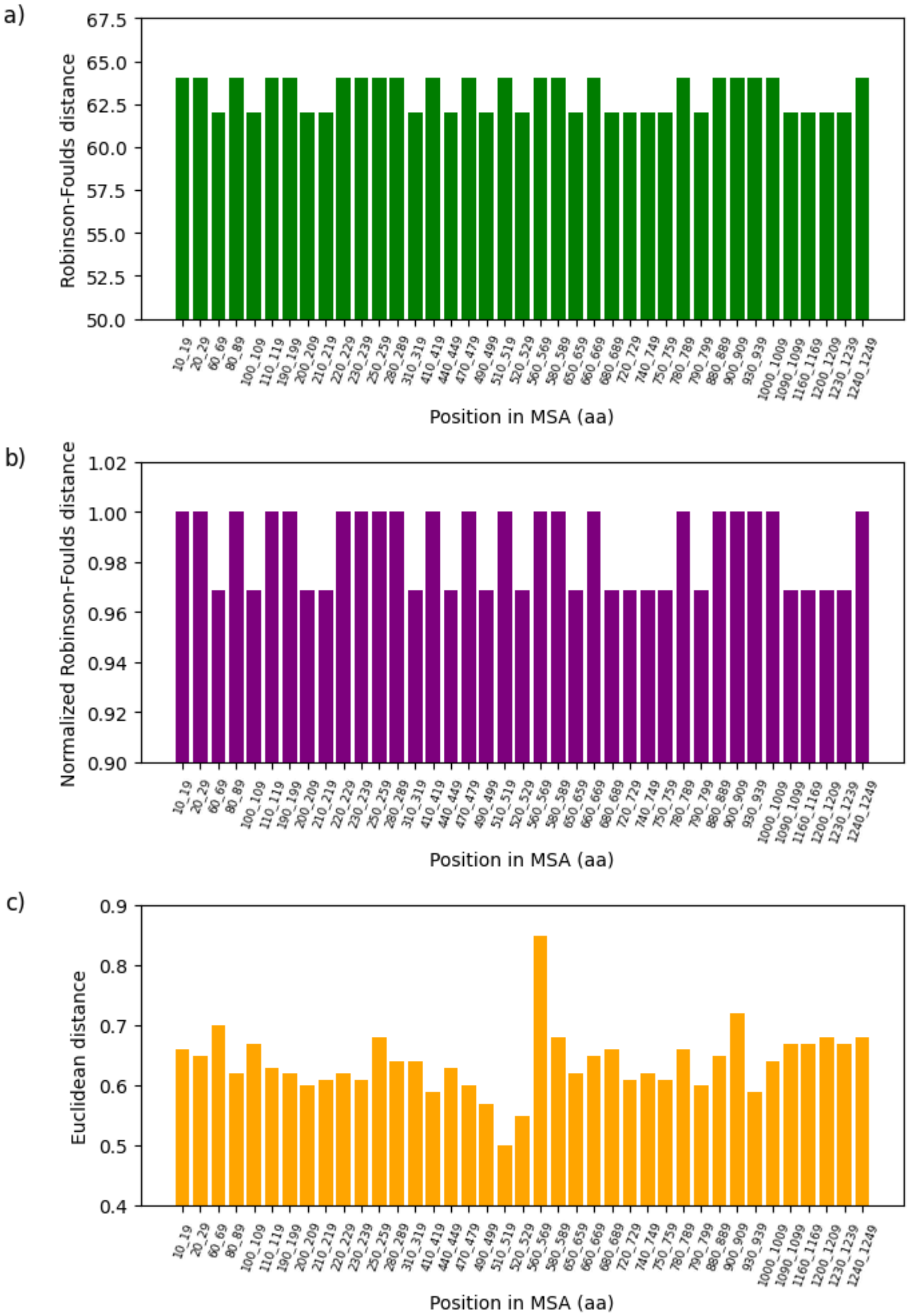


**Figure 7**. *Analysis of fluctuations in three distance metrics using multiple sequence alignment (MSA): a) Robinson-Foulds distance, b) normalized Robinson-Foulds distance, and c) Euclidean distance. Distance variations are studied to establish the potential dissimilarity between the partial sequence of the 16S rRNA mitochondrial gene of 62 Cumacea specimens and the variability of wind speed (m/s) at the start of sampling.*

wind speed (m/s) at the start of sampling (Position in MSA: 560-569 aa; Euclidean distance: 0.85; see Figure 7d) and $O_2$ concentration (mg/L) (Position in MSA: 1210-1219 aa; Euclidean distance: 1.23; see Figure 8d). These results could mean that these specimens have been shaped by these unique local environments, resulting in genetic sequences adapted to their particular conditions.

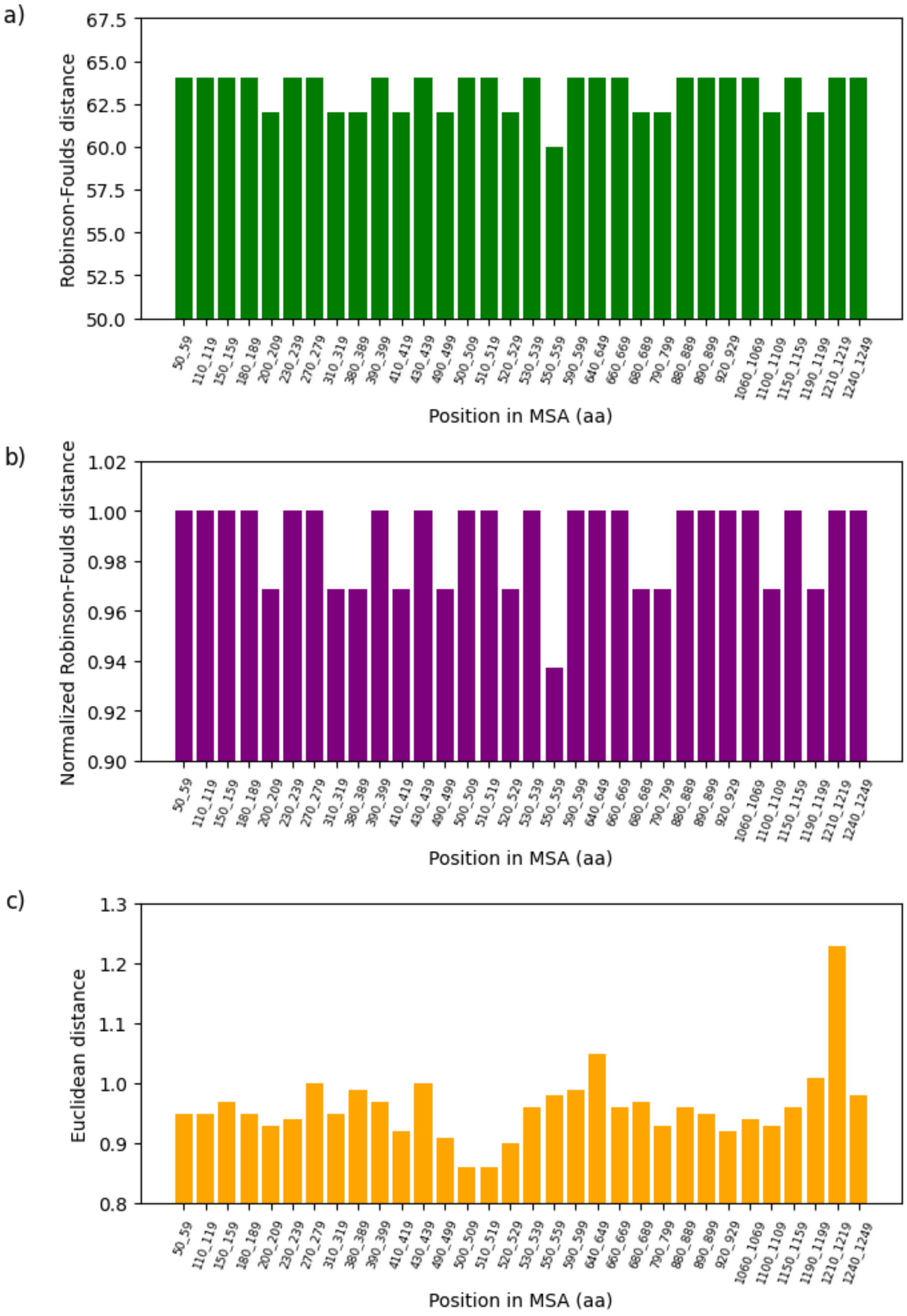


**Figure 8**. *Analysis of fluctuations in three distance metrics using multiple sequence alignment (MSA): a) Robinson-Foulds distance, b) normalized Robinson-Foulds distance, and c) Euclidean distance. These distances aim to determine the degree of dissimilarity between the partial sequence of the 16S rRNA mitochondrial gene of 62 Cumacea specimens and the variation in $O_2$ concentration (mg/L) at the sampling sites.*

The novelty in our research lies in the exhaustive divergence between habitat variables and genetic divegence in Cumacea, particularly in identifying genetic windows that diverge from habitat fluctuations, which has not been widely investigated in previous studies [14], [21]. Our integrated method identifies specific genetic regions sensitive to ecosystemic and atmospheric variations. Thus, the eventual identification of proteins linked to one of these variable DNA sequences will make it possible to represent its functional effects in responses to habitat changes. Our future research will focus on verifying the prediction of this protein and assessing its role in the physiological adaptation of Cumacea to fluctuating conditions, adding a link between genetic data and ecological function.

Interpreting how marine invertebrates genetically adapt to variations in their habitat can help predict their response to climate change and advance conservation plans to protect them. Identifying the variables that influence genetic variability in Cumacea can contribute to the designation and supervision of marine protected areas, assuring they include habitats crucial to the survival and acclimatization of these species. Thus, our results can inform the management of fishing and seabed mining companies by revealing ecologically vulnerable areas where these disturbances can seriously affect benthic biodiversity.

Furthermore, our results provide essential knowledge to guide future studies on the genetic adaptation of Cumacea and other invertebrates to ecological and regional variability. Based on these findings, future research should focus on additional ecosystemic and meteorological variables, such as nutrient accessibility, water pH, ocean currents, and the degree of human disturbance, to further improve the interpretation of the complex interactions between genetics and the environment. Extending the scope of application to other marine species, not just marine invertebrates, and various spatial regions would provide a better means of generalizing the results. With this in mind, longitudinal study models on these different species could reflect long-term climatic and biological fluctuations, and improve knowledge of the dynamics of genetic acclimatization.

However, it is important to recognize the limitations of our study. In particular, the three missing data points on $O_2$ concentration (mg/L) and the relatively small sample size ($n = 62$) may have induced a bias, which could impact the validity of our interpretations and restrict the generalizability of our results. Moreover, these missing data could provide partial insight into the relationship between $O_2$ concentration (mg/L) and genetic fluctuation in Cumacea, and our sample size may reduce the statistical power of our results. Future studies should address these gaps by incorporating larger sample sizes and more complete datasets to confirm and expand our conclusions. Additionally, as our research focuses solely on the partial sequence of the mitochondrial 16S rRNA gene, utilizing more elaborate genomic methods, such as whole-gene or even whole-genome sequencing, could help better understand marine species' genetic variety and global acclimatization mechanisms. This would provide more comprehensive genetic databases to improve accuracy and knowledge in identifying existing (and new) marine invertebrate species using DNA barcoding (e.g., mitochondrial DNA cytochrome c oxidase I (COX1)). Finally, multidisciplinary collaborations between ecology, genetics, and oceanography would be essential to enhance knowledge sharing and its application in future research.

## Acknowledgments

The authors thank the SciPy conference and reviewers for their valuable comments on this paper as well as Mansour Kebe for his technical support and Carolin Uhlir for her clarifications and advice on her study [3]. This work was supported by the Natural Sciences and Engineering Research Council of Canada (NSERC), the Fonds de recherche du Québec - Nature et technologies (FRQNT), the Université de Sherbrooke grant, and the Centre de recherche en écologie de l'Université de Sherbrooke (CREUS).

## References

[1] S. Schnurr, A. Brandt, S. Brix, D. Fiorentino, M. Malyutina, and J. Svavarsson, "Composition and distribution of selected munnopsid genera (Crustacea, Isopoda, Asellota) in Icelandic waters," *Deep Sea Research Part I: Oceanographic Research Papers*, vol. 84, pp. 142–155, 2014, doi: 10.1016/j.dsr.2013.11.004.

[2] K. Meißner, N. Brenke, and J. Svavarsson, “Benthic habitats around Iceland investigated during the IceAGE expeditions,” *Polish Polar Research*, vol. 35, no. 2, pp. 177–202, 2014, doi: 10.2478/popore-2014-0016.

[3] C. Uhlir *et al.*, “Adding pieces to the puzzle: insights into diversity and distribution patterns of Cumacea (Crustacea: Peracarida) from the deep North Atlantic to the Arctic Ocean,” *PeerJ*, vol. 9, p. e12379, 2021, doi: 10.7717/peerj.12379.

[4] L. A. Levin and P. K. Dayton, “Ecological theory and continental margins: where shallow meets deep,” *Trends in ecology & evolution*, vol. 24, no. 11, pp. 606–617, 2009, doi: 10.1016/j.tree.2009.04.012.

[5] A. D. Rogers, A. Baco, H. Griffiths, T. Hart, and J. M. Hall-Spencer, “Corals on seamounts,” *Seamounts: ecology, fisheries & conservation*, pp. 141–169, 2007, doi: 10.1002/9780470691953.ch8.

[6] R. Danovaro *et al.*, “Exponential decline of deep-sea ecosystem functioning linked to benthic biodiversity loss,” *Current Biology*, vol. 18, no. 1, pp. 1–8, 2008, doi: 10.1016/j.cub.2007.11.056.

[7] K. Meißner, S. Brix, K. M. Halanych, and A. M. Jażdżewska, “Preface—biodiversity of Icelandic waters,” *Marine Biodiversity*, vol. 48, no. 2, pp. 715–718, 2018, doi: 10.1007/s12526-018-0884-7.

[8] B. Stransky and J. Svavarsson, “Diversity and species composition of peracarids (Crustacea: Malacostraca) on the South Greenland shelf: spatial and temporal variation,” *Polar Biology*, vol. 33, no. 2, pp. 125–139, 2010, doi: 10.1007/s00300-009-0691-5.

[9] P. Rehm, “Cumacea (Crustacea; Peracarida) of the Antarctic shelf-diversity, biogeography, and phylogeny= Cumacea (Crustacea; Peracarida) des antarktischen Schelfs-Diversität, Biogeographie und Phylogenie,” *Berichte zur Polar-und Meeresforschung (Reports on Polar and Marine Research)*, vol. 602, 2009, doi: 10.2312/BzPM_0602_2009.

[10] R. M. Jennings and R. J. Etter, “Phylogeographic Estimates of Colonization of The Deep Atlantic by The Protobranch Bivalve Nucula Atacellana,” *Polish Polar Research*, no. 2, pp. 261–278, 2014, doi: 10.2478/popore-2014-0017.

[11] J. F. Grassle and N. J. Maciolek, “Deep-sea species richness: regional and local diversity estimates from quantitative bottom samples,” *The American Naturalist*, vol. 139, no. 2, pp. 313–341, 1992, doi: 10.1086/285329.

[12] M. A. Rex, R. J. Etter, and C. T. Stuart, “Large-scale patterns of species diversity in the deep-sea benthos,” *Marine biodiversity: patterns and processes*, pp. 94–121, 1997, doi: 10.1017/CBO9780511752360.006.

[13] C. J. Brown, S. J. Smith, P. Lawton, and J. T. Anderson, “Benthic habitat mapping: A review of progress towards improved understanding of the spatial ecology of the seafloor using acoustic techniques,” *Estuarine, Coastal and Shelf Science*, vol. 92, no. 3, pp. 502–520, 2011, doi: 10.1016/j.ecss.2011.02.007.

[14] R. C. Vrijenhoek, “Cryptic species, phenotypic plasticity, and complex life histories: assessing deep-sea faunal diversity with molecular markers,” *Deep Sea Research Part II: Topical Studies in Oceanography*, vol. 56, no. 19–20, pp. 1713–1723, 2009, doi: 10.1016/j.dsr2.2009.05.016.

[15] N. Balkenhol *et al.*, “Identifying future research needs in landscape genetics: where to from here?,” *Landscape Ecology*, vol. 24, pp. 455–463, 2009, doi: 10.1007/s10980-009-9334-z.

[16] S. Manel *et al.*, “Perspectives on the use of landscape genetics to detect genetic adaptive variation in the field,” *Molecular Ecology*, vol. 19, no. 17, pp. 3760–3772, 2010, doi: 10.1111/j.1365-294X.2010.04717.x.

[17] N. Balkenhol *et al.*, “Landscape genomics: understanding relationships between environmental heterogeneity and genomic characteristics of populations,” *Population genomics: Concepts, approaches and applications*, pp. 261–322, 2019, doi: 10.1007/13836_2017_2.

[18] A. B. Shafer and J. B. Wolf, “Widespread evidence for incipient ecological speciation: a metaanalysis of isolation-byecology,” *Ecology letters*, vol. 16, no. 7, pp. 940–950, 2013, doi: 10.1111/ele.12120.

[19] M. A. Rex, C. T. Stuart, and G. Coyne, “Latitudinal gradients of species richness in the deep-sea benthos of the North Atlantic,” *Proceedings of the National Academy of Sciences*, vol. 97, no. 8, pp. 4082–4085, 2000, doi: 10.1073/pnas.050589497.

[20] R. J. Etter and M. A. Rex, “Population differentiation decreases with depth in deep-sea gastropods,” *Deep Sea Research Part A. Oceanographic Research Papers*, vol. 37, no. 8, pp. 1251–1261, 1990, doi: 10.1016/0198-0149(90)90041-S.

[21] S. Manel, M. K. Schwartz, G. Luikart, and P. Taberlet, “Landscape genetics: combining landscape ecology and population genetics,” *Trends in ecology & evolution*, vol. 18, no. 4, pp. 189–197, 2003, doi: 10.1016/S0169-5347(03)00008-9.

[22] N. Balkenhol, L. P. Waits, and R. J. Dezzani, “Statistical approaches in landscape genetics: an evaluation of methods for linking landscape and genetic data,” *Ecography*, vol. 32, no. 5, pp. 818–830, 2009, doi: 10.1111/j.1600-0587.2009.05807.x.

[23] M. R. Gaither and L. A. Rocha, “Origins of species richness in the Indo-Malay-Philippine biodiversity hotspot: Evidence for the centre of overlap hypothesis,” *Journal of biogeography*, vol. 40, no. 9, pp. 1638–1648, 2013, doi: 10.1111/jbi.12126.

[24] M. A. Rex *et al.*, “Global bathymetric patterns of standing stock and body size in the deep-sea benthos,” *Marine Ecology Progress Series*, vol. 317, pp. 1–8, 2006, doi: 10.3354/meps317001.

[25] R. Danovaro, A. Dell'Anno, A. Pusceddu, C. Gambi, I. Heiner, and R. M̸bjerg Kristensen, “The first metazoa living in permanently anoxic conditions,” *BMC biology*, vol. 8, pp. 1–10, 2010, doi: 10.1186/1741-7007-8-30.

[26] S. A. Siedlecki *et al.*, “Experiments with seasonal forecasts of ocean conditions for the northern region of the California Current upwelling system,” *Scientific Reports*, vol. 6, no. 1, p. 27203, 2016, doi: 10.1038/srep27203.

[27] H. Waga, T. Hirawake, and J. M. Grebmeier, “Recent change in benthic macrofaunal community composition in relation to physical forcing in the Pacific Arctic,” *Polar Biology*, vol. 43, no. 4, pp. 285–294, 2020, doi: 10.1007/s00300-020-02632-3.

[28] H. Saeedi, D. Warren, and A. Brandt, “The Environmental Drivers of Benthic Fauna Diversity and Community Composition,” *Frontiers in Marine Science*, vol. 9, 2022, doi: 10.3389/fmars.2022.804019.

[29] P. Hugenholtz, B. M. Goebel, and N. R. Pace, “Impact of culture-independent studies on the emerging phylogenetic view of bacterial diversity,” *Journal of bacteriology*, vol. 180, no. 18, pp. 4765–4774, 1998, doi: 10.1128/jb.180.18.4765-4774.1998.

[30] C. Saccone, C. De Giorgi, C. Gissi, G. Pesole, and A. Reyes, “Evolutionary genomics in Metazoa: the mitochondrial DNA as a model system,” *Gene*, vol. 238, no. 1, pp. 195–209, 1999, doi: 10.1016/s0378-1119(99)00270-x.

[31] W. Li and N. Tahiri, “Host–Virus Cophylogenetic Trajectories: Investigating Molecular Relationships between Coronaviruses and Bat Hosts,” *Viruses*, vol. 16, no. 7, p. 1133, 2024, doi: 10.3390/v16071133.

[32] W. Li and N. Tahiri, “aPhyloGeo-Covid: A web interface for reproducible phylogeographic analysis of SARS-CoV-2 variation using Neo4j and Snakemake,” p. 44, 2023, doi: 10.25080/gerudo-f2bc6f59-00f.

[33] A. Koshkarov, W. Li, M.-L. Luu, and N. Tahiri, *Phylogeography: Analysis of genetic and climatic data of SARS-CoV-2*. 2022. doi: 10.25080/majora-212e5952-018.

[34] D. F. Robinson and L. R. Foulds, “Comparison of phylogenetic trees,” *Mathematical Biosciences*, vol. 53, no. 1, pp. 131–147, 1981, doi: 10.1016/0025-5564(81)90043-2.

[35] W. Li, A. Koshkarov, and N. Tahiri, “Comparison of phylogenetic trees defined on different but mutually overlapping sets of taxa: A review,” *Ecology and Evolution*, vol. 14, no. 8, p. e70054, 2024, doi: 10.1002/ece3.70054.

[36] N. Tahiri, M. Willems, and V. Makarenkov, “A new fast method for inferring multiple consensus trees using k-medoids,” *BMC evolutionary biology*, vol. 18, pp. 1–12, 2018, doi: 10.1186/s12862-018-1163-8.

[37] A. Czarna, R. Sanjuán, F. González-Candelas, and B. Wróbel, “Topology testing of phylogenies using least squares methods,” *BMC Evolutionary Biology*, vol. 6, pp. 1–13, 2006, doi: 10.1186/1471-2148-6-105.

[38] M. R. Smith, “Bayesian and parsimony approaches reconstruct informative trees from simulated morphological datasets,” *Biology letters*, vol. 15, no. 2, p. 20180632, 2019, doi: 10.1098/rsbl.2018.0632.

[39] M. R. Smith, “Information theoretic generalized Robinson–Foulds metrics for comparing phylogenetic trees,” *Bioinformatics*, vol. 36, no. 20, pp. 5007–5013, 2020, doi: 10.1093/bioinformatics/btaa614.

Open Access | https://doi.org/10.25080/MDCE8355

SciPy 2024
July 8 - July 14, 2024

Proceedings of the 23rd
Python in Science Conference
ISSN: 2575-9752

# Model Share AI: An Integrated Toolkit for Collaborative Machine Learning Model Development, Provenance Tracking, and Deployment in Python

**Heinrich Peters**[1], and **Michael Parrott**[1]

[1] Columbia University

## Abstract

Machine learning (ML) is revolutionizing a wide range of research areas and industries, but many ML projects never progress past the proof-of-concept stage. To address this problem, we introduce Model Share AI (AIMS), a platform designed to streamline collaborative model development, model provenance tracking, and model deployment, as well as a host of other functions aiming to maximize the real-world impact of ML research. AIMS features collaborative project spaces and a standardized model evaluation process that ranks model submissions based on their performance on holdout evaluation data, enabling users to run experiments and competitions. In addition, various model metadata are automatically captured to facilitate provenance tracking and allow users to learn from and build on previous submissions. Furthermore, AIMS allows users to deploy ML models built in Scikit-Learn, TensorFlow Keras, or PyTorch into live REST APIs and automatically generated web apps with minimal code. The ability to collaboratively develop and rapidly deploy models, making them accessible to non-technical end-users through automatically generated web apps, ensures that ML projects can transition smoothly from concept to real-world application.

**Keywords** Machine Learning, MLOps, Model Deployment, Provenance Tracking, Crowdsourcing

**Published** Jul 10, 2024

**Correspondence to**
Heinrich Peters
hp2500@columbia.edu

## 1. Introduction

Machine learning (ML) is revolutionizing a wide range of research areas and industries, providing data-driven solutions to important societal problems. The success of many ML projects depends on effective collaboration, rigorous evaluation, and the ability to deploy models. Traditionally, researchers and practitioners have been using version-control systems like GitHub in combination with custom model evaluation and benchmarking experiments to ensure reproducibility and to compare models. However, these systems tend to lack easy-to-use, structured pathways specifically designed to collaboratively develop and rapidly deploy ML models. Furthermore, the creation of custom resources for model evaluation, benchmarking, and deployment, can require substantial upfront effort. As a result, most models do not progress past the proof-of-concept stage and are never deployed [1], [2], preventing a wider audience from participating in the promise of applied ML research.

While the recent rise of platforms like Hugging Face Hub [3], TensorFlow Hub [4], and MLflow [5], [6], [7], illustrates the demand for open-source model repositories and MLOps solutions, barriers of entry are still high for researchers, educators, and practitioners from non-technical disciplines. Model Share AI (AIMS) addresses this problem by providing a lightweight, easy-to-use alternative. In a few lines of code, users can create Model Playgrounds - standardized ML project spaces that offer an all-in-one toolkit for collaborative model improvement, experiment tracking, model metadata analytics, and instant model deployment, allowing researchers to rapidly iterate on ML models in one streamlined workflow. The present paper provides an introduction to AIMS by situating it in relation to other current solutions and outlining its key functions, technical background, and workflow.

## 2. Related Work

The primary objective of AIMS is to offer an easy pathway to organize ML projects by reducing common and complex tasks down to minimal code. Its approach builds on the work of various projects focused on providing value for common ML tasks (e.g., model improvement, version tracking, deployment, etc.). Currently, there are several open-source tools and platforms, such as Hugging Face Hub [3], TensorFlow Hub [4], MLflow [5], [6], [7], and OpenML [8], [9], [10], providing ML model repositories where researchers can find and download model objects or deploy models. Hugging Face Hub [3] is a platform allowing users to share pre-trained models, datasets, and demos of ML projects. It has GitHub-inspired features for code-sharing and collaboration, such as discussions and pull requests, and it includes a pathway for model deployment into API endpoints. Similarly, TensorFlow Hub [4] is a repository and library for reusable ML in TensorFlow, enabling users to fine-tune and deploy deep learning models. Deployment is facilitated by TensorFlow Serving [11], which allows users to make their models accessible on a server through API endpoints. MLflow [5], [6], [7] is an open-source platform that manages the end-to-end ML lifecycle. It provides experiment tracking, code packaging, a model registry, model serving, and integration with all popular ML frameworks. While Hugging Face Hub, TensorFlow Hub, and MLflow are well suited for large-scale deep learning tasks, OpenML [8], [9], [10], focuses more on classic ML and model reproducibility. Here, researchers can share, explore, and experiment with ML models, datasets, and workflows, but there is currently no option for model deployment. The OpenML API provides access through various programming languages, but users can also employ an OpenML web interface to browse and visualize data, models, and experiments.

Hugging Face Hub and TensorFlow Hub are primarily *model-focused* platforms, providing repositories of pre-trained models that users can fine-tune for their specific purposes. OpenML and MLflow, on the other hand, are more *task-focused*, emphasizing model evaluation and benchmarking on standardized tasks. While all of these platforms are extensively used by ML researchers and practitioners, including for industry-scale projects, their wide range of features and customizations can be overwhelming for researchers from non-technical disciplines, students, and educators. In comparison, AIMS stands out by prioritizing ease of use and a hyper-collaborative approach. Unlike Hugging Face Hub and TensorFlow Hub, AIMS emphasizes model metadata analytics and task-focused collaboration. Compared to MLflow, it offers more community-based features like competitions that promote collective problem-solving and crowd-sourcing. Moreover, while OpenML excels in model evaluation and benchmarking, AIMS provides additional capabilities for model deployment, allowing users to share models directly from their local environment into live REST APIs and auto-generated web applications. Taken together, the key distinctions of AIMS are its beginner-friendly design and its strong focus on collaborative model development, as well as its goal of providing value not just for ML researchers but also for researchers, practitioners, and educators from non-technical disciplines.

## 3. Model Share AI

AIMS provides standardized ML project spaces (Model Playgrounds; see Figure 1) with accessible MLOps features designed to support collaborative model development, model

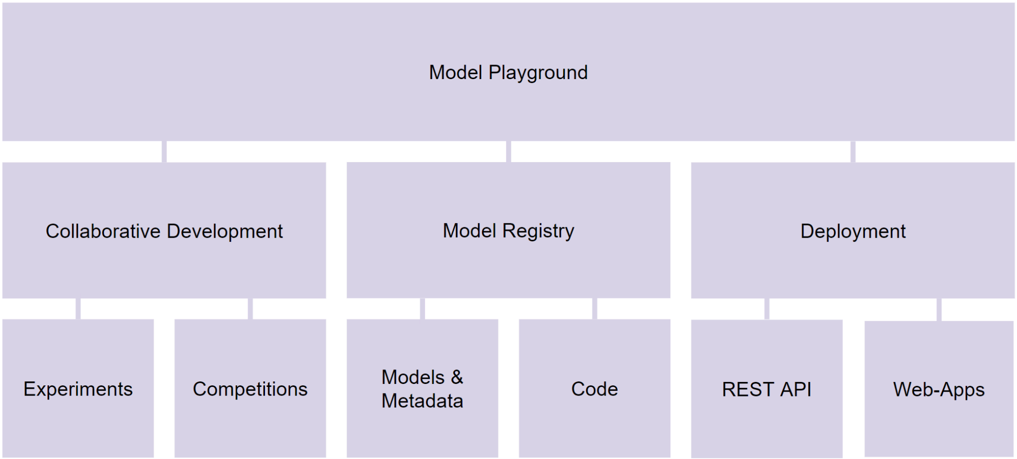


**Figure 1**. *A Model Playground is a standardized ML project space, representing a ML task that is associated with a specific dataset and a task type (classification, regression). A Model Playground includes Experiments and Competitions for collaborative model development, a model registry with model metadata and user-generated code, as well as deployment functionalities including a live REST API and pre-built web-apps.*

metadata analytics, and model deployment, as well as a host of other functions aiming to maximize the real-world impact of ML research. As such, it simplifies, combines, and extends the capabilities of current solutions in 4 important ways:

1. Collaborative model development: AIMS facilitates collaborative model development and crowd-sourcing through standardized model evaluation procedures, experiment tracking, and competitions.
2. Model registry: AIMS hosts model objects, as well as model metadata, automatically extracted from each model submission, enabling users to analyze which modeling decisions lead to high performance on specific tasks.
3. Model deployment: Model deployment is simplified considerably, allowing users to share their models into live REST APIs and pre-built web apps directly from their Python training environments with just a few lines of code.
4. AIMS provides a wide range of supporting functionalities, including workflows for reproducibility, data sharing, code sharing, creating ML project portfolios, and more.

## *3.1. Key Functions*

### 3.1.1. *Collaborative Model Development:*

A key feature of AIMS is its focus on collaborative model development and crowd-sourced model improvement, enabling teams to iterate quickly by allowing collaborators to build on each other's progress, even across libraries. For supervised learning tasks, users can collaboratively submit models into Experiments or Competitions associated with a Model Playground project in order to track model performance and rank submissions in standardized leaderboards according to their evaluation metric of choice. Experiments and Competitions are set up by providing holdout evaluation data against which the predictions of submitted models are evaluated. Standardized model evaluations allow collaborators to track the performance of their models along with a wide range of model metadata that are automatically extracted from submitted models and added to the model registry (see section below). Out of the box, AIMS calculates accuracy, f1-score, precision, and recall for classification tasks, and mean squared error, root mean squared error, mean absolute error, and $R^2$-scores for regression tasks. The main difference between Experiments and Competitions is that a proportion of the evaluation data is kept secret for Competitions, preventing participants from deliberately overfitting on evaluation data. Being able to submit models into shared Experiments enables ML teams to standardize tasks, rigorously track their progress, and build on each other's success, while Competitions facilitate crowd-sourced solutions. Both Experiments and Competitions can be either public (any AIMS user can submit) or private (only designated team members can submit). Users can deploy any model from an Experiment or Competition into the REST API associated with their Model Playground with a single line of code.

### 3.1.2. *Model Registry:*

Model versions are made available for each Model Playground and comprehensive model metadata are automatically extracted for each submitted model. In addition to evaluation metrics, this includes hyperparameter settings for Scikit-Learn models and model architecture data (such as layer types and dimensions, number of parameters, optimizers, loss function, memory size) for Keras and Pytorch models. Users can also submit any additional metadata they choose to capture. Model metadata are integrated into Competition and Experiment leaderboards, enabling users to analyze which types of models tend to perform well for a specific ML task. Users can either visually explore leaderboards on their Model Playground page or they can download leaderboards into Pandas data frames to run their own analyses. There is also a set of AIMS methods designed to visualize model metadata. For example, models can be compared in a color-coded layout showing differences in model architectures and hyperparameter settings. Furthermore, users can instantiate models from the AIMS model registry into reproducible environments. Taken together, these functions are designed to streamline the management, collaboration, and deployment of ML models, enhancing their discoverability, reproducibility, and traceability throughout their lifecycle.

### 3.1.3. *Instant Model Deployment:*

AIMS currently allows users to deploy ML models built in Scikit-Learn [12], Tensorflow Keras [13], [14], and Pytorch [15] into live REST APIs rapidly with minimal code. Additionally, users can deploy models from other ML frameworks by transforming them into ONNX (Open Neural Network Exchange) format - an open-source format for standardized representations of ML models with the goal of making them interoperable across platforms. Each deployed model is associated with a Model Playground page on the AIMS website and a REST API endpoint hosted in a serverless AWS backend. End-users can either manually upload data to make predictions using an automatically generated web app on the Model Playground Page, or they can programmatically query the REST API associated with the model. In addition to auto-generated web apps, AIMS enables users to submit their own Streamlit apps. Out of the box, AIMS supports models built on tabular, text, image, audio, and video data. Allowing users to deploy models with minimal effort and making those models accessible to even non-technical end-users through web apps holds the promise of making ML research applicable to real-world challenges.

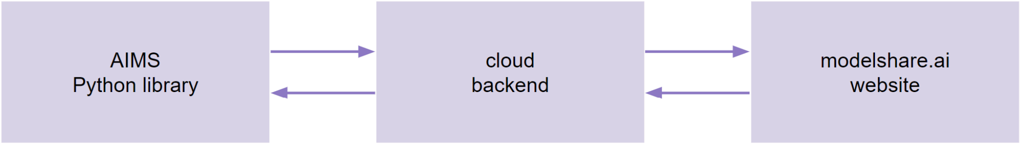


**Figure 2.** *Overview of the AIMS architecture. The AIMS Python library allows users to create Model Playground pages, submit and deploy models, and analyze model metadata. The modelshare.ai website provides a graphical user interface to explore model metadata and generate predictions via auto-generated web apps. All required resources are automatically generated in scalable serverless cloud infrastructure.*

### *3.2. Architecture*

AIMS consists of three main components: an open-source Python library, user-owned cloud backend resources, and the AIMS website (see Figure 2). The AIMS Python library is the main interface allowing users to set up Model Playground pages (including Experiments and Competitions), submit and deploy models, analyze model metadata, and reproduce model artifacts. It provides an accessible layer that facilitates the creation of the cloud backend resources that power REST APIs, as well as model evaluations and model metadata extraction. The *ModelPlayground()* class acts as a local representation of a Model Playground page and its associated REST API. It provides a range of methods to configure, change, and query Model Playground resources. A detailed overview of the Python library is provided below (AIMS Workflow).

The cloud backend hosts model objects and associated artifacts in S3 storage, while REST APIs are deployed into serverless lambda functions. Lambda functions are programs or scripts that run on high-availability AWS compute infrastructure. They can be invoked by various event sources (e.g., API calls) and scale automatically based on the volume of incoming requests. This means that users do not have to maintain servers, and they only pay for the time and resources actually consumed, but not for idle time. The AIMS Python library allows users to automatically generate and deploy lambda functions based on specific properties of their ML tasks, without the need to explicitly manage any AWS resources. The most important lambda functions in the context of AIMS are the Evaluation Lambda, which computes evaluation metrics and extracts model metadata from submitted models, and the Main Lambda, which computes predictions on data submitted through the REST API. Runtime models are automatically packaged into Docker containers that run on lambda. Additionally, certain metadata are stored in a centralized Redis database that powers the modelshare.ai website.

The AIMS website hosts user profile pages, model pages, web apps, example code, and a documentation page, as well as user-generated code and documentation for their specific ML projects (see Supplementary Information A-E).

### *3.3. AIMS Workflow*

The AIMS workflow is designed to help teams collaboratively and continuously train, evaluate, improve, select, and deploy models using standardized ML project spaces or Model Playgrounds. After training a model, users submit their model to a Competition or Experiment associated with a Model Playground. The model is then automatically evaluated, and model metadata are extracted. Evaluations and metadata are made available via a graphical user interface on the Model Playground page or can be queried using the AIMS Python library, enabling users to analyze which types of models perform well on a given learning task. This information can then be used to either improve the training process and submit more models or to select a model and instantly deploy it with a single line of code. Deployed models can easily be swapped out if they degrade over time or if they are outperformed by new submissions. An overview of the process can be found in Figure 3.

In order to use AIMS, users first need to create an AIMS user profile and generate credentials through the AIMS website. Additionally, users are required to generate AWS credentials if they wish to deploy models into their own AWS resources.

The AIMS workflow is centered around the concept of Model Playgrounds. A Model Playground is a standardized ML project space, representing an ML project that is associated with a specific dataset and a task type, such as classification or regression. A Model Playground is first instantiated locally using the *ModelPlayground()* class of the AIMS Python library. Here, users need to specify the *input_type* (tabular, text, image, audio, etc.) and

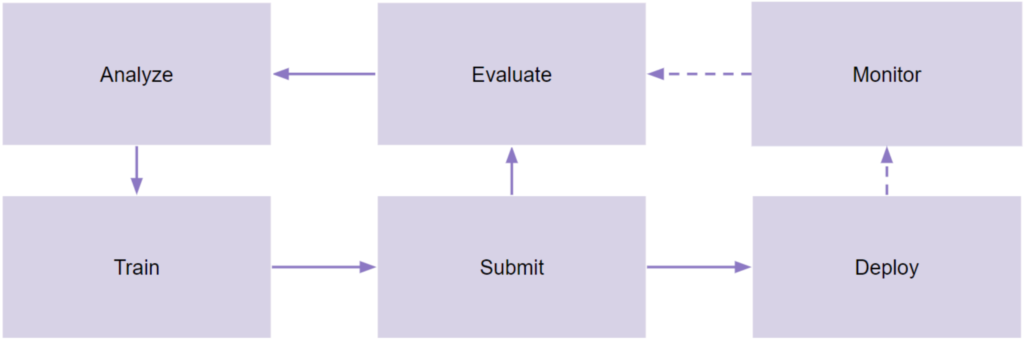


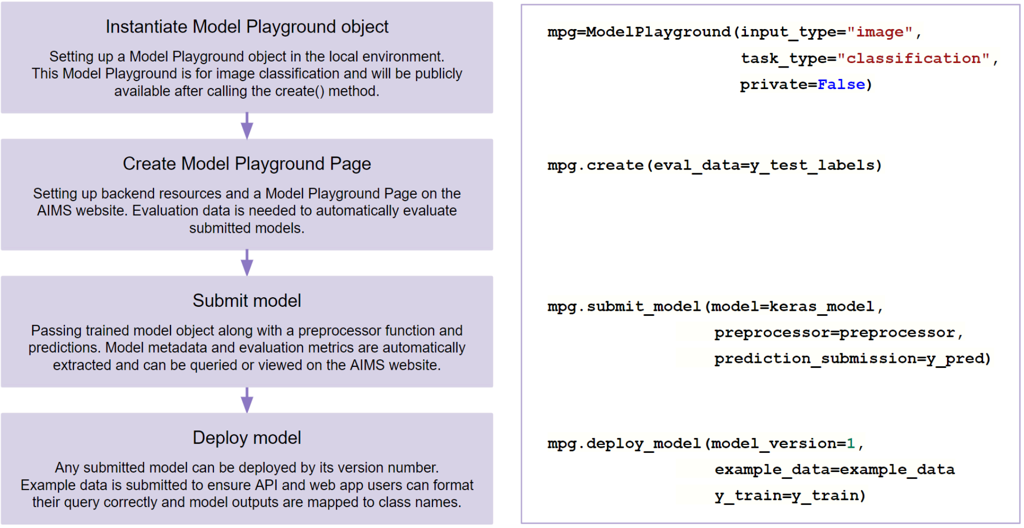
**Figure 3.** *MLOps workflow with AIMS. Users iteratively train, submit, evaluate, and analyze their models. As contributors have access to model evaluations, model architecture metadata, and reproducible model objects of previous submissions, they can rapidly develop high-performing models. Submitted models can easily be deployed into live REST APIs. Deployed runtime models can be monitored and seamlessly be swapped out against newly submitted models.*

the *task_type* (classification, regression), as well as select whether their Model Playground should be public or private. Private Model Playgrounds can only be accessed by invited collaborators, whereas public Model Playgrounds are open to all AIMS users. After instantiating their Model Playground object, users can create an online Model Playground page by calling the *create()* method and submitting their evaluation data. This generates a fully functioning Model Playground Page on the AIMS website, including a placeholder REST API, and enables users to submit models into associated Experiments and Competitions.

Once a Model Playground Page is created, users can start submitting models to Experiments or Competitions and deploy models into the Model Playground's REST API. A model submission includes a model object (Scikit-Learn, Keras, PyTorch, ONNX), a preprocessor function, and a set of predicted values corresponding to the previously submitted evaluation data. The predictions are then evaluated against the evaluation data, and model metadata are automatically extracted from model objects. Users can submit additional metadata in dictionary format using the *custom_metadata* argument of the *submit_model()* method. After submitting one or more models, users can explore evaluations and model metadata on the associated Model Playground page or query this information for further analysis using the *get_leaderboard()* and *compare_models()* methods. To inspect and improve models, users can instantiate models from the leaderboard using the *instantiate_model()* method. Alternatively, users can instantly deploy a model using the *deploy_model()* method by referring to the model's leaderboard version number. Additionally, users should submit example data, which will help end-users format their input data correctly, and y training data to help the web app and prediction API transform raw model outputs into the correct labels. As mentioned above, the process does not stop when a model is deployed. Users can submit and analyze more models, informed by previous submissions, and easily monitor and swap out the runtime model using the *update_runtime_model()* method. An overview of the model deployment process, including code, is available in Figure 4. Detailed tutorials and documentation can be found in Supplementary Information A.

## 4. Impact

Collaborative model improvement and crowd-sourcing are important, not only because they make teams more efficient but also because they foster diverse perspectives. Such collective efforts can contribute to the democratization of ML, provide access to resources for a wider audience, and enable community-driven innovation. For instance, the AIMS Competition feature can facilitate crowd-sourced research projects in various disciplines utilizing the common task framework. Relatedly, the AIMS model registry promotes discov-



Instantiate Model Playground object

Setting up a Model Playground object in the local environment. This Model Playground is for image classification and will be publicly available after calling the create() method.

Create Model Playground Page

Setting up backend resources and a Model Playground Page on the AIMS website. Evaluation data is needed to automatically evaluate submitted models.

Submit model

Passing trained model object along with a preprocessor function and predictions. Model metadata and evaluation metrics are automatically extracted and can be queried or viewed on the AIMS website.

Deploy model

Any submitted model can be deployed by its version number. Example data is submitted to ensure API and web app users can format their query correctly and model outputs are mapped to class names.

```
mpg=ModelPlayground(input_type="image",
                    task_type="classification",
                    private=False)

mpg.create(eval_data=y_test_labels)

mpg.submit_model(model=keras_model,
                 preprocessor=preprocessor,
                 prediction_submission=y_pred)

mpg.deploy_model(model_version=1,
                 example_data=example_data
                 y_train=y_train)
```

**Figure 4**. *Model deployment process: Models can be deployed with just a few lines of code. After instantiating a local Model Playground object, users can create a Model Playground page that is ready for model submissions. Submitted models are automatically evaluated and can be deployed into live REST APIs.*

erability, transparency, and reproducibility by providing a centralized platform for users to find models and their associated metadata. Sharing model metadata, such as model architectures, hyperparameters, training data, and evaluation metrics, allows researchers to verify previous results and build upon each other's work. The repository also acts as an educational resource, offering students, educators, and self-learners the opportunity to study various ML models, techniques, and best practices. For example, AIMS has thus far been extensively used in classrooms at Columbia University and beyond to organize challenges, collaboratively work on ML projects, and make models accessible through API endpoints. In addition, AIMS is positioned to realize several important missions with regard to model deployment. Firstly, simplifying model deployment is important because it lowers barriers to entry, allowing more developers, researchers, and organizations to incorporate ML in their projects. Secondly, easier deployment saves resources and time, so that developers can dedicate more effort to model training, tuning, and evaluation. Ultimately, a streamlined deployment process allows users to iterate faster and explore novel ideas in real-world contexts.

## 5. Future Work

While AIMS provides a simple approach to collaborative model development and deployment, there are opportunities for further improvements. Firstly, it is important to strike the right balance between flexibility and ease of use. A highly flexible platform may allow researchers to use various ML frameworks and libraries, accommodate a wide range of data sources and formats, and support custom deployment strategies. However, this flexibility may come at the cost of increased complexity and a harder learning curve for users. On the other hand, a more user-friendly solution might provide a simpler, more streamlined interface but may lack the flexibility to accommodate unique or complex requirements. By default, AIMS prioritizes standardization and ease of use, making it attractive for researchers, educators, and data scientists who are interested in quick and lightweight solutions. For users who want more flexibility, AIMS provides customizations, such as custom AWS lambda functions, custom containers, and custom metadata submissions. We will continue to extend the functionality of AIMS to make it useful for a wide range of users and applications. This includes making the platform compatible with more advanced model types, task types, and ML frameworks (e.g., PySpark). Relatedly, future work could include additional MLOps functionality, including improved pathways for monitoring, model explainability, continuous retraining, and automated ML (AutoML) [16], [17]. The latter point is of special interest, as each new model submission contributes to a growing repository of evaluated, reusable ML models, including rich model metadata. These resources can be utilized to suggest pre-trained models that are expected to work well for a given task or enable users to run their own analyses to choose pre-trained models. We hope that AIMS will become a resource for researchers interested in meta-learning [18], [19] and related problems. Additionally, we expect AIMS to be used increasingly by researchers from various disciplines, including social sciences and natural sciences, trying to solve substantive questions through ML. We are planning to further accommodate their diverse needs in order to promote cross-fertilization and widespread participation in the promise of applied ML research.

## 6. Conclusion

AIMS provides a versatile yet easy-to-use approach to collaborative model development, model metadata analytics, and model deployment. It enables users to quickly find, analyze, and collaboratively improve trained ML models, and to share models from their local environment directly into live REST APIs and pre-built web applications in a single tightly integrated workflow. Compared to existing solutions, AIMS is intended to lower the barriers of entry to ML research, making it attractive to a large group of researchers, educators, and data scientists. We are confident that AIMS will become a widely used tool and maximize the real-world impact of ML research.

### Acknowledgments

M.P. developed the original idea for the Model Share AI platform in 2019, assembled the team, and led the way to the platform's completion as the chief engineer and team lead. H.P. has contributed to the development of the AIMS Python library and has had pivotal impact on ideation and development since 2020. We are profoundly grateful to our early and repeated organizational backers. Columbia University's QMSS program and Director Greg Eirich were early supporters. Professor David Park, Columbia University's former GSAS Dean of Strategic Initiatives, helped to carve out a strategy that led to the project's eventual fundraising success. A special acknowledgment goes to Columbia University's ISERP Startup Center. Their funding enabled Dr. Michael Parrott, our founding Director, to assemble an early team and develop a prototype of the platform. This early funding and technical prototype laid the groundwork for further large-scale funding and support from the Alfred P. Sloan Foundation. Director Josh Greenburg and the Tech program team at Sloan gave us the momentum and repeated support to innovate and build what has become the first research and education-facing MLOps platform. Without their grant resources and their belief in our success, this project would have been impossible. Of course, resources are only a starting point. We must highlight the exceptional contributions of Sam Alsmadi, who developed the AIMS website, and Gretchen Street, who led the way on library user security, ML datasets, and documentation. Finally, we want to thank the talented alumni who have contributed code over the years. This platform would not have been possible without all your hard work and innovation!

### References

[1] T. H. Davenport and D. J. Patil, "Is Data Scientist Still the Sexiest Job of the 21st Century?," *Harvard Business Review*, Jul. 2022, [Online]. Available: https://hbr.org/2022/07/is-data-scientist-still-the-sexiest-job-of-the-21st-century

[2] E. Siegel, “Models Are Rarely Deployed: An Industry-wide Failure in Machine Learning Leadership.” [Online]. Available: https://www.kdnuggets.com/models-are-rarely-deployed-an-industry-wide-failure-in-machine-learning-leadership.html

[3] “Hugging Face Hub.” [Online]. Available: https://huggingface.co/docs/hub/index

[4] “TensorFlow Hub.” [Online]. Available: https://www.tensorflow.org/hub

[5] “MLflow.” [Online]. Available: https://mlflow.org/

[6] A. Chen *et al.*, “Developments in MLflow: A System to Accelerate the Machine Learning Lifecycle,” in *Proceedings of the Fourth International Workshop on Data Management for End-to-End Machine Learning*, in DEEM'20. New York, NY, USA, Jun. 2020, pp. 1–4. doi: 10.1145/3399579.3399867.

[7] M. Zaharia *et al.*, “Accelerating the Machine Learning Lifecycle with MLflow,” *IEEE Data Eng. Bull.*, 2018, [Online]. Available: https://www.semanticscholar.org/paper/Accelerating-the-Machine-Learning-Lifecycle-with-Zaharia-Chen/b2e0b79e6f180af2e0e559f2b1faba66b2bd578a

[8] M. Feurer *et al.*, “Openml-python: an extensible python api for openml,” *The Journal of Machine Learning Research*, vol. 22, no. 1, pp. 4573–4577, 2021.

[9] J. Vanschoren, J. N. van Rijn, B. Bischl, and L. Torgo, “OpenML: networked science in machine learning,” *ACM SIGKDD Explorations Newsletter*, vol. 15, no. 2, pp. 49–60, Jun. 2014, doi: 10.1145/2641190.2641198.

[10] J. N. van Rijn *et al.*, “OpenML: A Collaborative Science Platform,” in *Machine Learning and Knowledge Discovery in Databases*, H. Blockeel, K. Kersting, S. Nijssen, and F.̌ Zelezný, Eds., in Lecture Notes in Computer Science. Berlin, Heidelberg, 2013, pp. 645–649. doi: 10.1007/978-3-642-40994-3_46.

[11] C. Olston *et al.*, “TensorFlow-Serving: Flexible, High-Performance ML Serving.” [Online]. Available: http://arxiv.org/abs/1712.06139

[12] F. Pedregosa *et al.*, “Scikit-learn: Machine Learning in Python,” *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011, [Online]. Available: http://jmlr.csail.mit.edu/papers/v12/pedregosa11a.html

[13] M. Abadi *et al.*, “TensorFlow: A system for large-scale machine learning,” 2016, doi: https://doi.org/10.48550/arXiv.1605.08695.

[14] F. Chollet, “Keras: the Python deep learning API.” [Online]. Available: https://keras.io/

[15] A. Paszke *et al.*, “PyTorch: An Imperative Style, High-Performance Deep Learning Library.” [Online]. Available: http://arxiv.org/abs/1912.01703

[16] F. Hutter, L. Kotthoff, and J. Vanschoren, Eds., *Automated Machine Learning: Methods, Systems, Challenges*. in The Springer Series on Challenges in Machine Learning. Springer International Publishing, 2019. doi: https://doi.org/10.1007/978-3-030-05318-5.

[17] X. He, K. Zhao, and X. Chu, “AutoML: A Survey of the State-of-the-Art,” *Knowledge-Based Systems*, vol. 212, p. 106622, 2021, doi: 10.1016/j.knosys.2020.106622.

[18] T. Hospedales, A. Antoniou, P. Micaelli, and A. Storkey, “Meta-Learning in Neural Networks: A Survey.” [Online]. Available: http://arxiv.org/abs/2004.05439

[19] C. Finn, P. Abbeel, and S. Levine, “Model-Agnostic Meta-Learning for Fast Adaptation of Deep Networks.” [Online]. Available: http://arxiv.org/abs/1703.03400

OPEN ACCESS | https://doi.org/10.25080/FMKR8387

# Scikit-build-core

## A modern build-backend for CPython C/C++/Fortran/Cython extensions

**Henry Schreiner**[1], **Jean-Christophe Fillion-Robin**[2], and **Matt McCormick**[2]

[1]Princeton University, [2]Kitware Inc.

**SciPy 2024**

*July 8 - July 14, 2024*

Proceedings of the 23rd Python in Science Conference
ISSN: 2575-9752

### Abstract

Discover how scikit-build-core revolutionizes Python extension building with its seamless integration of CMake and Python packaging standards. Learn about its enhanced features for cross-compilation, multi-platform support, and simplified configuration, which enable writing binary extensions with pybind11, Nanobind, Fortran, Cython, C++, and more. Dive into the transition from the classic scikit-build to the robust scikit-build-core and explore its potential to streamline package distribution across various environments.

**Keywords** Build system, CMake

## 1. Introduction

Python packaging has evolved significantly in the last few years. Standards have been written to allow the development of new build backends not constrained by setuptool's complex legacy. Build time dependencies, once nearly impossible to rely on, are now the standard, and required for any build system, including the original setuptools. And this new system is controlled by one central location, the `pyproject.toml` file.

We present scikit-build-core: a new build system based on these standards that brings together Python packaging and the CMake build system, the popular general purpose build system for languages like C, C++, Fortran, Cython, and CUDA. Together, this new system provides a simple entry point to building extensions that still scales to major projects. This allows everyone to take advantage of existing libraries and harness the performance available in these languages.

We will look at creating a package with scikit-build-core. Then we will cover some of it's most interesting and innovative features. Then we will peer into the design and internal workings. Finally, we will look at some of the packages that adopted scikit-build-core. A lot of the ecosystem was improved, even beyond scikit-build-core, as part of this project, so we will also highlight some of that work.

## 2. History of packaging

Python has a long history compared to modern languages with first-party packaging solutions; packaging wasn't something that was considered important for Python for quite a while. Python gained a standard library module to help with packaging called `distutils` in Python 1.6 and 2.0, in the year 2000, nearly ten years after the initial release. Distribution was difficult, leading to packages containing large numbers of distinct modules (such as SciPy [1]) to reduce the number of packages one had to figure out how to install, and "distributions" of Python to be created, such as the Enthought distribution. This eventually

**Published** Jul 10, 2024

**Correspondence to**
Henry Schreiner
henryfs@princeton.edu

led to the creation of Conda, a modular binary package manager that was particularly good at (and written in) Python.

Several developments greatly improved Python's packaging story, such the addition of a new binary format, the "wheel" made distributing binaries easier. This, along with new hardware and a changing installation landscape highlighted an issue with putting a packaging tool in the standard library: you can't freely update the standard library. Third party "extensions" to distutils appeared; the one that became the main tool for many years is setuptools. It wasn't long before setuptools became required; building directly with distutils was too buggy, and setuptools was injecting patches into distutils when it loaded.

Package installers, originally `easy_install` and then the more full-featured `pip` came along. `pip` was tightly coupled to setuptools, and even helped it out by making sure it's injections to distutils were done, even if packages didn't import setuptools first or at all. Pip was directly and deeply tied to setuptools; if you wanted to build a package, setuptools was the only answer. Even if you make the SDist (Python's source distribution format) and wheel(s) (Python's binary distribution format) yourself, which was actually pretty easy, as those were standardized, you couldn't make sure that `pip` wouldn't try to use setuptools to build a wheel from an SDist.

Faults in the organically grown distutils/setuptools quickly became apparent. You couldn't tell the installer to update setuptools from `setup.py`, because it was just a Python file that was being run. It was hard to declare built-time dependencies (though they really did try by running the Python file twice, the first time with stubs). You couldn't parse metadata without running the setup.py. It was hard to extend the setuptools commands, and the entire API was public, which meant it was really hard to fix something without breaking everyone else.

Third party tools for building packages started showing up, like Flit and Poetry. These worked by making a compatibility `setup.py` and injecting it into the SDist, just in case pip needed to build the wheel. This is when standardization efforts, in the form of PEPs, began to change the packaging landscape forever.

PEP 517 [2] defined an API for build-frontends (like pip) to talk to build-backends (like setuptools). PEP 518 [3] defined an isolated build environment, which would allow builders to download build dependencies, like a specific version of setuptools or a plugin. Later, PEP 621 [4] would add a standard way to define basic package metadata, and PEP 660 [5] would add an API for editable installs.

Python build backends started to appear. Most of the initial build backends were were designed for Python-only packages. `flit-core`, `poetry-core`, `pdm-backend`, and `hatchling` are some popular build backends that support some or all of these PEPs. Setuptools also ended up gaining support for all these PEPs, as well.

Compiled backends were a bit slower, but today we have several great choices. `scikit-build-core` for CMake, `meson-python` for Meson, and `maturin` for Cargo (Rust) are the most popular. `enscons` for SCons should get a special mention as the first binary build backend, even though it is mostly a historical curiosity at this point.

A side note on Conda: like most general package managers, it runs arbitrary commands from a recipe to build and install the package, then it captures the installed files. Many Python packages use the `pip install .` command, but any commands can be placed here, including native CMake calls; as long files are placed in the right place, the result can be used from conda installed Python. Though packages doing this may forget to generate a `<package>.dist-info` directory, which is used by things like `importlib.metadata`.

## 3. Scikit-build (classic)

The original scikit-build [6] was released as PyCMake at SciPy 2014, and renamed two years later, at SciPy 2016, following the “scikit” convention introduced by SciPy. Being developed well before the packaging PEPs, it was designed as a wrapper around distutils and/or setuptools. This design had flaws, but was the only way this could be done at the time.

Because it was deeply tied to setuptools internals, updates to setuptools or wheel often would break scikit-build. There were a lot of limitations of setuptools that scikit-build couldn’t alleviate properly.

However, it did allow users to use a real build system (CMake) with their Python projects. A notable example are two packages produced by the scikit-build team: ninja and cmake redistributions on PyPI. Users could pip install `cmake` and `ninja` anywhere that wheels worked.

In 2021, a proposal was written and accepted by the NSF to fund development on a new package, scikit-build-core, built on top of the packaging standards and free from setuptools and other historical cruft. Work started in 2022, and the first major package to switch was Awkward Array, at the end of that year.

## 4. Using scikit-build-core

### *4.1. Binary Python package generation overview*

Figure 1 provides an overview of modern binary Python package generation. The package developer begins by defining the package structure, incorporating package source files, a CMakeLists.txt CMake configuration, and a pyproject.toml file adhering to PEP 517 specifications. To initiate the package build process, the developer employs a Build Frontend tool, such as `pip` or `build`, and may supply optional configuration parameters. The chosen Build Frontend then triggers the PEP 517 Build Backend, exemplified by scikit-build-core, which in turn employs CMake, managing configuration options appropriately. Utilizing a blend of build system introspection and configured options alongside the *CMakeLists.txt* logic, CMake generates a native build system configuration. This native build system, whether Ninja configuration files, Unix Makefiles, Visual Studio Project Files, etc., defines dependencies for the artifacts produced by the native toolchain, which is responsible for compiling C/C++ source code into native binaries.

Once configured, scikit-build-core initiates the native build system, managing the bundling of resulting build artifacts and metadata into a wheel. These wheels may undergo additional processing with post-processing tools to enhance platform compatibility. End-users of the package have the option to directly utilize the wheel via a package manager or generate their own package by invoking the build system from a source distribution (SDist). Typically, both the binary wheels and the source distributions are uploaded to the Python Package Index (PyPI) for broader accessibility.

### *4.2. The simplest example*

Scikit-build-core was designed to be easy to use as possible, with reasonable defaults and simple configuration. A minimal, working example is possible with just three files. You need a file to build, for example, this is a simple pybind11 module in a `main.cpp`:

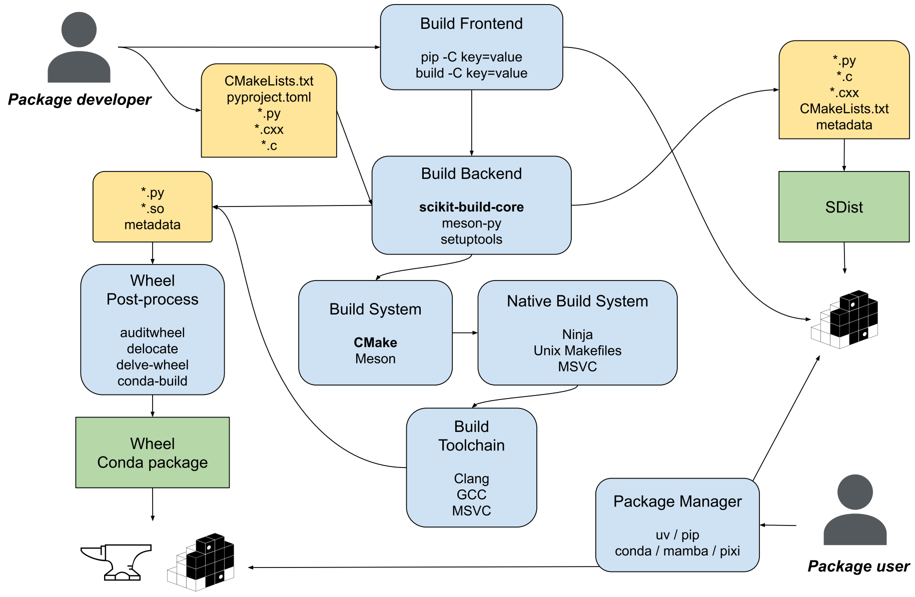


**Figure 1.** *Binary Python packaging dataflow.*

```cpp
#include <pybind11/pybind11.h>

PYBIND11_MODULE(example, m) {
    m.def("square", [](double x) { return x*x; });
}
```

You then need a `pyproject.toml`, and it can be as little as six lines long, just like for pure Python:

```toml
[build-system]
requires = ["scikit-build-core", "pybind11"]
build-backend = "scikit_build_core.build"

[project]
name = "example"
version = "0.0.1"
```

And finally, you need a `CMakeLists.txt` for CMake, which is also as little as six lines:

```cmake
cmake_minimum_required(VERSION 3.15...3.26)
project(example LANGUAGES CXX)

set(PYBIND11_NEWPYTHON ON)
find_package(pybind11 CONFIG REQUIRED)

pybind11_add_module(example example.cpp)
install(TARGETS example LIBRARY DESTINATION .)
```

And that all that is needed to get started.

### *4.3. Common needs as simple configuration*

Customizing scikit-build-core is done through the `[tool.scikit-build]` table in the `pyproject.toml`. An example configuration is shown below:

```
[tool.scikit-build]
minimum-version = "0.10"

build.verbose = true
logging.level = "INFO"

wheel.expand-macos-universal-tags = true
```

On line 2, you can see the `minimum-version` setting. This is a special setting that allows scikit-build-core to make changes to default values in the future while ensuring your configuration continues to build with the defaults present the version you specify. This is very similar to CMake's `cmake_minimum_requires` feature.

On line 4-5, you can see an example of increasing the verbosity, both to print out build commands as well as see scikit-build-core's logs.

On line 7, you can see that scikit-build-core can expand the macOS universal tags to include both ARM and Intel variants for maximum compatibility with older Pip versions; this is impossible to do in setuptools unless you use API declared private and unstable in wheel.

## 5. Innovative features of scikit-build-core

Scikit-build-core has some interesting and powerful innovations. This section is by no means exhaustive; it doesn't cover scikit-build-core's over forty different configuration options, for example. It is instead just meant to highlight some of the most interesting features of scikit-build-core.

### *5.1. Dynamic requirement on CMake/Ninja*

Scikit-build-core needs CMake to build, and often ninja as well. Both are available as Python wheels, but simply adding them to your `build-system.requires` list is not a good idea, because some platforms aren't supported by wheels on PyPI, such as WebAssembly, the various BSD's, ClearLinux, or Android. But these systems can get CMake other ways, such as from a package manager, so users shouldn't have a build fail because they can't get a cmake wheel if they have a sufficient version of CMake already. As it turns out, PEP 517 is flexible enough to handle this very elegantly.

PEP 517 has an API for a build tool to declare it's dependencies. This was initially used for setuptools to only depend on wheel when making wheels, but it works perfectly for optional binary dependencies as well. Scikit-build-core will check to see if `cmake` is present on the system. If it is, and it reports a version sufficient for the package, it will not be added to the requested dependencies. This is also done for ninja. This same system was added to `meson-python`, as well, since Meson also requires ninja.

### *5.2. Integration with external packages*

Scikit-build-core has three mechanisms to allow packages on PyPI to provide CMake configuration and tools:

- The site-packages directory is added to the CMake search path by scikit-build-core. Due to the way CMake config file discovery works, this allows a package to provide a `<PkgName>Config.cmake` file in any GNU-like standard location in the package (like `pybind11/share/cmake/pybind11/pybind11Config.cmake`, for example).
- An entry point `cmake.prefix`, which adds prefix dirs, useful for adding module files that don't have to match the installed package name or are in arbitrary locations.
- An entry point `cmake.module`, which can be used to add any CMake helper files.

### 5.3. *Dual editable modes with automatic recompile*

Editable installs are supported in two modes. The default mode installs a custom finder that combines the Python files in your development directory (as specified by packages) with CMake installed files in site-packages. It is important to rely on importlib.resources instead of file manipulation in this mode. This mode supports installs into multiple environments.

This mode also supports automatic rebuilds. If you enable it, then importing your package will rebuild on any changes to the source files, something that could not be done with setuptools.

There is also an opt-in "inplace" mode that uses CMake's inplace build to build your files in the source directory. This is very similar to setuptools `build_ext --inplace`, and works with tools that can't run Python finders, like type checkers. This mode works with a single environment, because the build dir *is* the source dir. There is no dynamic finder in this mode, so automatic rebuilds are not supported.

### 5.4. *Dynamic metadata*

A system for allowing custom metadata plugins was contributed and has been one of the most successful parts of scikit-build-core. Any metadata field other than the `name` can be dynamically specified in Python packaging; the dynamic-metadata mechanism provides a way for plugins to be written by anyone, including "in-tree" plugins written inside a specific package. This gives authors quite a bit of freedom to do things like customize the package description or read the version from a file.

This system is being worked on as a standard for other build backends to use and a stand-alone namespace. Changes are expected, but the current implementation in scikit-build-core uses the following API:

```python
def dynamic_metadata(
    field: str,
    settings: Mapping[str, Any],
) -> str:
   ...
```

Plugins provide this function. Three built-in plugins are provided in scikit-build-core: one for regex, one that wraps `setuptools_scm`, and one that wraps `hatch-fancy-pypi-readme`. Using one of these looks like this:

```toml
name = "mypackage"
dynamic = ["version"]

[tool.scikit-build.metadata.version]
provider = "scikit_build_core.metadata.regex"
input = "src/mypackage/__init__.py"
```

The `provider` key tells scikit-build-core where to look for the plugin. There is also a provider-path for local plugins. All other keys are implemented by the plugin; the regex plugin implements `input` and `regex`, for example.

An extra feature that is not being proposed for broader adoption is a generation mechanism. Scikit-build-core can generate a file for you with metadata provided via templating. It looks like this:

```
[[tool.scikit-build.generate]]
path = "package/_version.py"
template = '''
version = "${version}"
'''
```

This is extremely useful due to the `location` key, which defaults to `install`, which will put the file only in the built wheel, `build`, which puts the file in the build directory, and `source`, which writes it out to the source directory and SDist, much like `setuptools_scm` normally does. In the default mode, though, no generated files are placed in your source tree.

### *5.5. Overrides*

Static configuration has many benefits, but it has a significant drawback; you often want to configure different situations differently. For example, you might want a higher minimum CMake version on Windows, or a pure Python version for unreleased Python versions. These sorts of things can be expressed with overrides, which was designed after overrides in cibuildwheel, which in turn were based on mypy's overrides. Scikit-build-core has the most powerful version of the system, however, with an `.if` table that can do version comparisons and regexs, and supports any/all merging of conditions, and inheritance from a previous override (also added to cibuildwheel). It includes conditions for the state of the build (wheel, SDist, editable, or metadata builds) and environment variables. An example is shown below:

```
[[tool.scikit-build.overrides]]
if.platform-system = "darwin"
cmake.version = ">=3.18"
```

This will change the minimum CMake version to `>=3.18` on macOS.

## 6. Scikit-build-core's design

This section is devoted to the internals of scikit-build-core.

### *6.1. The configuration system*

Scikit-build-core has over forty options, and each of these options can be specified in the `pyproject.toml`, or via config-settings (`-C` in build or pip), or via environment variables starting with `SKBUILD`. We also provide a JSON Schema for the TOML configuration, and a list of all options in the README. All of these are powered by a single dataclass-based configuration system developed for scikit-build-core.

All of the configuration lives in dataclasses that look like this:

```python
@dataclasses.dataclass
class LoggingSettings:
    level: Literal["NOTSET", "DEBUG", "INFO", "WARNING", "ERROR", "CRITICAL"] = (
        "WARNING"
    )
    """
    The logging level to display, "DEBUG", "INFO", "WARNING", and "ERROR" are
    possible options.
    """

@dataclasses.dataclass
class ScikitBuildSettings:
    cmake: CMakeSettings = dataclasses.field(default_factory=CMakeSettings)
    ninja: NinjaSettings = dataclasses.field(default_factory=NinjaSettings)
    logging: LoggingSettings = dataclasses.field(default_factory=LoggingSettings)
    ...
```

The system is a conversion system, originally based on `cattrs`, but reimplemented in pure Python to reduce dependencies and the potential for uncontrollable breakage. The system recognised a wide variety of types, including nested types and as seen above, literals. A comment in a string following the option is also recognised for the JSONSchema/README processors, because nothing has been standardized for variable docstrings, and Sphinx understands this convention.

Converters are implemented for TOML, config-settings (dict), and environment variables. The converters fill the `ScikitBuildSettings` instance with the values from the three[1] sources. Once read in, these are just a normal fully typed dataclass, so access and usage is natural and protected by mypy. Overrides are currently implemented via pre-processing the TOML. A post processing step is also added to validate the settings, and produce errors/warnings based on invalid combinations.

### *6.2. The File API*

CMake provides a File API, which allows third party tools to read a variety of information about the build process. Scikit-build-core has a full implementation of a reader for the File API for use in plugins (currently not required for scikit-build-core itself).

The File API was implemented via dataclasses following the official schema. A recursive conversion system, similar to the settings system, was implemented to convert the written files into dataclass instances.

### *6.3. Plugins for other systems*

Scikit-build-core 0.9.0 includes two plugins, one for setuptools, and one for hatchling. It was designed to support making custom plugins wrapping CMake builds, so these two internal plugins are examples of using scikit-build-core to make plugins. These will eventually be pulled out into separate packages; being internal allowed faster iteration on the scikit-build-core API used by plugins.

## 7. Adoption

Projects moving to scikit-build-core have seen over 800 lines of convoluted `setup.py` code turn into 20 lines of CMake and a minimal configuration file. Often this adds new platforms, like Windows or PyPy, that were not feasible to support before, and faster multithreaded compile times.

[1] Converters are combined using a chain converter, which allows any combination of converters; this is used by the hatchling plugin to support four converters, since it includes hatchling's TOML configuration as well as scikit-build-core's.

### *7.1. Statistics*

Scikit-build-core is currently the most downloaded build backend designed for compiled languages other than Rust[2], with 2.7M monthly downloads[3]. It is used by a dozen packages in the top 8,000 PyPI packages, like `pyzmq`, `lightgbm`, `cmake`, `phik`, and `clang-format`. Other packages like `ninja` and `boost-histogram` have adopted it, but have not made a release with it yet.

It is possible to download every pyproject.toml from PyPI[4] and perform analyses on packages that provided an SDist. You can compute the number of packages published using scikit-build-core (255 as of June 25, up from 82 last year), and investigate the details of how projects are setting configuration.

We also monitor non-fork mentions of `scikit_build_core` in `pyproject.toml`'s using GitHub Code Search, of which there are currently (July 3) 764 results, up from 92 on July 1, 2023). This string is required as part of the build backend, so serves as a reasonable method to track this info. A search for the more precise string `scikit_build_core.build`, which avoids hatchling and setuptools plugins, provides 720 results; a similar search was not performed last year for comparison, though.

### *7.2. Rapids.ai*

Rapids.ai moved their package generator to scikit-build-core, affecting all their packages, like cudf, cugraph, cuml, and rmm. They have several unusual requirements due to the need to support cuda variants. They developed a wrapper for scikit-build-core that changes the name of the package based on the current cuda version (something explicitly disallowed by PEP 621) and injects modified dependencies. Discussions on ways to handle external dependencies like CUDA without such workarounds was a major topic at the packaging summit in PyCon US 24.

### *7.3. Ninja / CMake / clang-format*

One powerful use case for binaries is PyPI redistribution of CLI tools. This is used for `ninja` and `cmake`, which are first-party scikit-build-core projects, along with many third party projects like clang-format, use this to great effect. For example, clang-format provides easily pip-installable wheels that are under 2 MB and work on almost any system. We are not aware of any easier way to get a working copy of clang-format.

The core idea for most of these projects is to install the binary in the `scripts` folder of a wheel, which will be placed directly in a user's path. Scikit-build-core sets CMake variables with the various wheel folders; this one is `${SKBUILD_SCRIPTS_DIR}`.

The other common need, and one very difficult to handle in a setuptools-based wrapper, stems from the fact that these binary packages don't build against Python, so they don't need to be build per Python version. This can be easily indicated in scikit-build-core by setting:

```toml
[tool.scikit-build]
wheel.py-api = "py3"
```

This will build a wheel that doesn't depend on the Python API and will only depend on the system architecture. This same setting can be used for the Limited API / Stable ABI as well, by setting `cp311` or a similar minimum value. Pythons before this value will build traditional wheels, and after will build Stable ABI wheels.

[2] The Rust-only Maturin is much older and has 57 packages in the top 8,000.

[3] https://hugovk.github.io/top-pypi-packages

[4] https://github.com/henryiii/pystats

### 7.4. *PyZMQ*

The python wrapper for the ZeroMQ project uses scikit-buid-core. This is their configuration:

```
[tool.scikit-build]
wheel.packages = ["zmq"]
wheel.license-files = ["licenses/LICENSE*"]
# 3.15 is required by scikit-build-core
cmake.version = ">=3.15"
# only build/install the pyzmq component
cmake.targets = ["pyzmq"]
install.components = ["pyzmq"]
```

They explicitly list the package name (since it doesn't match the project, `pyzmq` and their licences files are in a non-standard location. The `cmake.version` setting is superfluous, since that's already scikit-build-core's minimum. The `cmake.targets` (will be `build.targets` in future versions, with back-compatibility support) and `install.components` are quite powerful, though. The target list allows building just the required target for the binding, saving time. And then installing just things marked with a specific component gives you full control over exactly what goes into the wheel.

## 8. Related work

As part of this project, a lot of other packages were touched on or improved. `pybind11` had some build system improvements, especially related to modern FindPython support. Nanobind improved support for the Stable ABI, and the `nanobind_example` project moved to scikit-build-core. Cibuildwheel and build were improved for all backends with extensive testing and usage in scikit-build-core. validate-pyproject was improved and validate-pyproject-schema-store was added to make it easier to use the Schema Store's schema files, including scikit-build-core's, for validation of pyproject.toml's. Scikit-build-core (along with meson-python and maturin) were added to the Scientific Python Development Guide. Fixes were made in Pyodide to ensure better CMake support for WebAssembly builds (which were also added to cibuildwheel). And scikit-build-core was one of the first backends to support free-threaded Python 3.13 builds.

## 9. Summary

Scikit-build-core is one of the best build systems available today for Python compiled extensions, allowing access to a powerful compiled build tool (CMake) from the comfort of a statically configured `pyproject.toml` file. It supports a wide variety of innovative controls, including many things not possible with a setuptools build, like good config-settings support, wheel tag customization, and more.

Support for this work was provided by NSF grant OAC-2209877.

## References

[1] P. Virtanen *et al.*, "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python," *Nature Methods*, vol. 17, pp. 261–272, 2020, doi: https://doi.org/10.1038/s41592-019-0686-2.

[2] N. J. Smith and T. Kluyver, "A build-system independent format for source trees," 2015. [Online]. Available: https://www.python.org/dev/peps/pep-0517

[3] B. Cannon, N. J. Smith, and D. Stufft, "Specifying Minimum Build System Requirements for Python Projects," 2015. [Online]. Available: https://www.python.org/dev/peps/pep-0518

[4] B. Cannon, D. Ingram, P. Ganssle, S. Eustace, T. Kluyver, and ping Tzu-Chung, "Storing project metadata in pyproject.toml," 2020. [Online]. Available: https://www.python.org/dev/peps/pep-0621

[5] D. Holth and S. Bidoul, “Editable installs for pyproject.toml based builds (wheel based),” 2021. [Online]. Available: https://www.python.org/dev/peps/pep-0660

[6] J.-C. Fillion-Robin, M. McCormick, O. Padron, M. Smolens, M. Grauer, and M. Sarahan, “jcfr/scipy_2018_scikit-build_talk: SciPy 2018 Talk | scikit-build: A Build System Generator for CPython C/C++/Fortran/Cython Extensions.” [Online]. Available: https://doi.org/10.5281/zenodo.2565368

Open Access | https://doi.org/10.25080/HWGA5253

SciPy 2024

July 8 - July 14, 2024

Proceedings of the 23nd Python in Science Conference
ISSN: 2575-9752

# Making Research Data Flow With Python

**Josh Borrow**[1], **Paul La Plante**[2,3], **James Aguirre**[1], and **Peter K. G. Williams**[4]

[1]Department of Physics and Astronomy, University of Pennsylvania, 209 South 33rd Street, Philadelphia, PA, USA 19104, [2]Department of Computer Science, University of Nevada, Las Vegas, NV 89154, [3]Nevada Center for Astrophysics, University of Nevada, Las Vegas, NV 89154, [4]Center for Astrophysics | Harvard & Smithsonian, 60 Garden St., Cambridge, MA 02138

## Abstract

The increasing volume of research data in fields such as astronomy, biology, and engineering necessitates efficient distributed data management. Traditional commercial solutions are often unsuitable for the decentralized infrastructure typical of academic projects. This paper presents the Librarian, a custom framework designed for data transfer in large academic collaborations, designed for the Simons Observatory (SO) as a ground up re-architechture of a previous astronomical data management tool called the 'HERA Librarian' from which it takes its name. SO is a new-generation observatory designed for observing the Cosmic Microwave Background, and is located in the Atacama desert in Chile at over 5000 meters of elevation.

Existing tools like Globus Flows, iRODS, Rucio, and Datalad were evaluated but were found to be lacking in automation or simplicity. Librarian addresses these gaps by integrating with Globus for efficient data transfer and providing a RESTful API for easy interaction. It also supports transfers through the movement of physical media for environments with intermittent connectivity.

Using technologies like Python, FastAPI, and SQLAlchemy, the Librarian ensures robust, scalable, and user-friendly data management tailored to the needs of large-scale scientific projects. This solution demonstrates an effective method for managing the substantial data flows in modern 'big science' endeavors.

**Keywords** data, python, research

**Published** Jul 10, 2024

**Correspondence to**
Josh Borrow
josh@joshborrow.com

## 1. Introduction

Research data is ever-growing, with even small projects now producing terabytes of data. This has been matched by an increase in the typical size of workstation storage and compute, but as many fields like astronomy, biology, and engineering continue their march towards 'big science', distributed data analysis is becoming significantly more common. As such, there is a significant need for software that can seamlessly track and move data between sites without continuous human intervention. For the very largest of these projects (such as the Large Hadron Collider at CERN), massive development effort has been invested to create technologies for data pipelining, but these tools typically assume a high level of control over the software and hardware that is running for the project.

At the same time, data science and data analytics are becoming a larger part of industrial infrastructure, even in what appear to be non-technical fields like law, commerce, and media. This has led to a huge increase in data engineering and pipelining software, but this software is generally only available through private clouds like those provided by Amazon (Amazon Web Services), Microsoft (Azure), and Alphabet (Google Cloud). These services generally assume that there is an ingest point into one of their managed storage (e.g.

Amazon S3), using their compute and tools for analysis (e.g. Amazon Athena and Amazon EC2), and their tools for final data egress (e.g. Amazon API Gateway or Amazon QuickSight). These resources are mature, performant, and a result of hundreds of thousands or millions of developer-hours. However, they are not compatible with the current publicly-funded science model, or the way that large, distributed, academic collaborations work.

A monolithic stack like those provided by public clouds is almost antithetical to large, distributed, academic projects. In these projects, funding may come from a vast array of sources (e.g. faculty members funded through their institution, large funding agencies like the National Science Foundation, or more junior members funded through individual fellowships), and compute may be provided by many different providers at different scales (e.g. on-premises with the data source, small institutional Tier-2 or Tier-3 centers, or international-scale Tier-0 centers through an allocation). Pooling these funds and resources is an almost insurmountable challenge; aside from some $10 B+ projects, even 'massive' multi-hundred million dollar projects must design their staffing, software, and procedures for a level of flexibility (and, by corollary, simplicity) that would be incomprehensible in an industrial context.

This paper is about solving the base-level problem: how to get raw experiment data totaling around 1 PB a year from a remote site (in our case, the top of a mountain in Chile), to various science consumers distributed across the globe. Our data is comprised of folders (termed 'books'), each containing 10-20 GB of data, laid out in a pre-determined scheme in a POSIX filesystem.

## 2. Comparing Data Flows

In Figure 1, we show two example data flows: one for a commercial enterprise (on the top), and one for a typical academic project (bottom; this matches closely with the needs of the Simons Observatory).

A typical challenge for the commercial scenario is collating data from many different sources, and processing it in a latency-sensitive manner. Consider an e-commerce platform, which has sales data, market research, and user tracking data, which all need to be collected and normalised to make decisions. This data, though varied, is typically small (and all 'owned' in some sense by the company), meaning that it can all be placed inside one centralised data warehouse from which analysis can be performed, and data egress can take place (often through a shared service). Because these services must all be directly acquired by the business, they are all at a bare minimum highly configurable and scalable. It can hence make sense to either use a series of interconnecting services (e.g. those provided through the Apache Software Foundation if using on-premises hardware), or a managed service collection (e.g. using tools from AWS). The data flows, represented by arrows in this case, are then made much simpler, and specific business logic can be programmed around the assumption of a shared and well-known interface; this hence leads to our understanding of these interconnected services through the data producers, flows, and products.

For an academic project, there is usually one major data source, and analysis is usually compute-intensive. This could be an experiment or, in our case, an observatory. Connected to this experiment there is typically a small on-site data storage and analysis facility that repackages data and can perform simple analysis. Those running the experiment usually have complete control over only this platform; the rest are highly locked-down shared computing services like national facilities (e.g. NERSC) and university-managed clusters - funding agencies typically do not support the purchase of significant compute resources, as they maintain shared high performance computing clusters. An important corollary to this is that different facilities may have different resources available, for instance cold

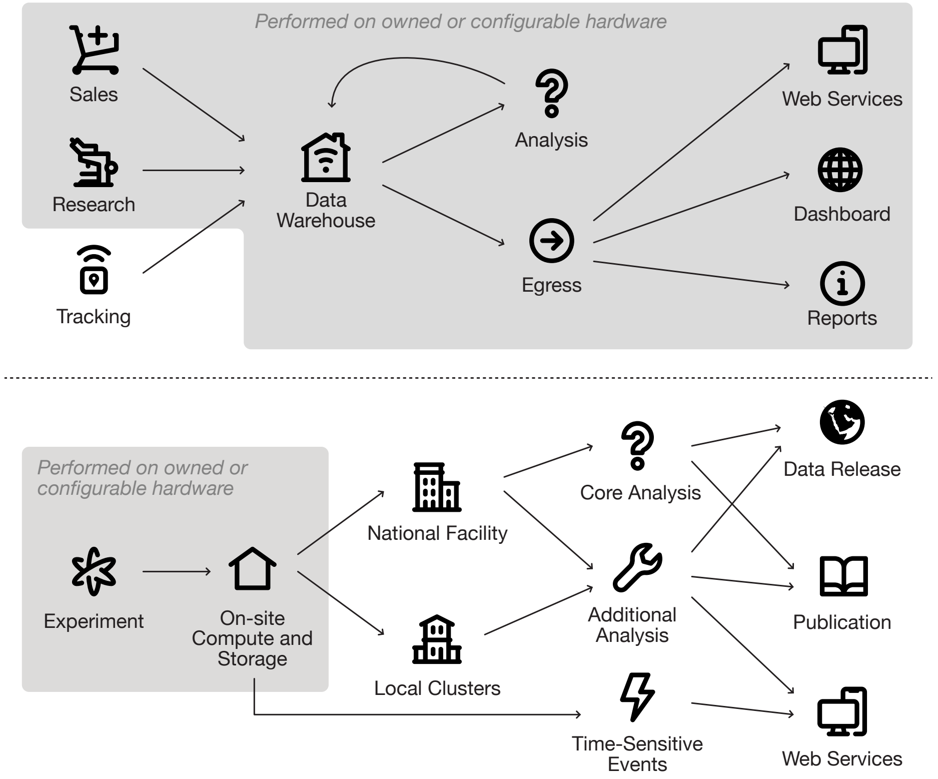


**Figure 1**. *Showing a typical layout of a commercial data engineering structure (top) versus an academic one (bottom). In a commercial context, it is typical that a significantly higher fraction of the resources are owned (or are directly configurable, e.g. cloud services) by the user. It is also much more common to have a centralised 'data warehouse' or 'data lake' which provides ingress and egress for all data products. In the academic context, only a very small fraction of the compute and storage are typically owned by stakeholders. National facilities, and even local clusters at universities, are typically rigid and cannot be re-configured to better suit the needs of an individual experiment. Further, due to limited compute and storage constraints, it is likely that multiple copies of experimental data must be kept at various sites, and tight control needs to be maintained on the specific sub-sets of data stored at each.*

storage for archival data, accelerators, or high-memory nodes. Data must be managed with the understanding that these resources may have finite lifetimes, may be withdrawn, or replaced at a completely different site. Further, as there is no centralised provider, data replication must be managed 'manually' by the collaboration through their network of shared machines, both for integrity and for availability at the sites that have different compute resources. Working on the data, specific archival products are produced: either those included in a data release (e.g. images, maps, etc.), and publications. Both regulation and community standards typically leads to these products being *immutable* and available *forever*, hosted by an external service. This reliance on external compute and storage leads to our understanding of this workflow through the flow of data into and out of the afore-mentioned providers.

Though they may have varied locations and resources, the computing centers are all usually organised in a broadly similar way, with dedicated data ingest (or transfer) nodes with a fast 'grid' internet connection (e.g. Internet2 in the U.S., JISC in the U.K., etc.) and access to shared disks. Compute is managed via a job scheduler (e.g. slurm), with file storage provided on GPFS or Lustre (i.e. POSIX-like, not object store) systems. As such, much of the community software has been built with the strong assumption that data is stored in files on a high-performance POSIX-compliant filesystem, with analysis performed through job submission to compute nodes. Any data management system for such projects must hence be designed to be compatible with this workflow.

In the specific case of the Simons Observatory [1], we have a need for a piece of software to bring down data from our observatory at the rate of around 20 TB a week. This is accomplished using SneakerNet, where physical media is used to transport data, and a fiber internet connection. Our data is immutable, and in the form of 'books' (structured directories of binary data), within a pre-determined directory structure that must be replicated exactly on all of the downstream sites. At certain downstream sites, we have space for all of the data from the entire project, but at others only a recent, rolling, sub-set may be maintained.

### *2.1. Existing Software*

Because of the significantly different constraints in the academic world versus the commercial world, there is a relatively limited set of tools that can be used to accomplish the goal of automated data management and transfer. In our search, we evaluated the following tools:

#### 2.1.1. *Globus:*

Globus [2], [3] is a non-profit service from the University of Chicago, and specialises in the batch transfer of large data products. Globus works by having an 'endpoint server' running at each location, which can communicate over HTTP. Through the use of their web interface, API, or command-line client, it is possible to request asynchronous transfers of massive amounts of data, which are transferred at close to line speeds thanks to GridFTP being used for the underlying data movement. Globus is used extensively in the academic community, and is already available ubiquitously throughout the academic cluster ecosystem.

Globus, however, is not an automated tool; in general, one must tell Globus how, where, and when to move data. It is possible to set up recurring synchronisations of data, but this either requires complete copy at all sites (lest data that was just deleted be moved again) or that all sub-tasks be expressible as part of a 'Globus Flow', which may not always be possible. As such, Globus is more of a 'data transfer' tool, than a data management tool. It does not have significant cataloguing capability.

#### 2.1.2. *Rucio and iRODS:*

Rucio [4] is a distributed data management system that was originally designed for the ATLAS high-energy physics experiment at the LHC, and as such is extremely scalable (i.e. exabytes or more). Rucio can be backed by different levels of storage, from fast (SSD) to slow (tape), and is declarative (meaning that one simply asks the system to follow a set of rules, like 'keep three copies of the data'), with its own user and permissions management system. Rucio is an exceptionally capable piece of software, but this comes with significant complexity. Further, data is managed externally to the underlying filesystem and permissions model of the host, potentially causing issues with the user agreements at shared facilities, and interacting with data generally requires interaction with the Rucio API. Though Rucio is certainly a fantastic tool, we found that it was too integrated for our needs of simply transferring data, and the declarative system is incompatible with our need to leverage SneakerNet.

A similar, albeit older and more mature, project to Rucio is iRODS (the integrated rule-oriented data system; R. Consortium [5]). iRODS, being more mature, integrates nicely with lots of software (providing a FUSE mount, FTP client integration, and more), and can likely be a perfect solution for someone looking for fully integrated data management. However, as it is built on a foundation of rule-based data management, it was discovered to be incompatible with our need to tightly control the specific transfers taking place.

### 2.1.3. *Datalad and git-annex:*

The git-annex [6] tool is an extension to Git, the version control system, to handle large quantities of binary data. Due to its extensive use by home and small business users, it explicitly supports SneakerNet transfers between sites, but is hampered by the fact that it relies on Git to track files (though not explicitly their contents; git-annex tracks checksums instead), which can become slow as repository size scales.

Datalad [7] is effectively a frontend for git-annex with a significant feature pool, and is used extensively in the bioscience community. For projects with small-to-medium size data needs, Datalad would be an exceptional candidate, but as we expect to need to manage multiple petabytes of data over a variety of storage needs, it was unfortunately not appropriate for the Simons Observatory.

## 3. The Librarian

After much consideration, the collaboration decided that building an orchestration and tracking framework on top of Globus was most appropriate. In addition, there was an existing piece of software used for the Hydrogen Epoch of Reionization Array (HERA) [8], a radio telescope designed to study the early Universe, called the 'HERA Librarian' [9], [10] that provided some of the abstractions that were required, though it was not suitable for production use for the much larger Simons Observatory. The HERA Librarian used a significant number of shell commands over SSH in its underlying architecture, and was designed to be fully synchronous. This lead to a complete re-write of the Librarian, primarily to eliminate these deficiencies, but also to migrate the application to a more modern web framework. There are a few key design concepts that were carried over from the HERA Librarian:

- There is a distinction drawn between 'Files' and 'Instances' of files in

the Librarian; a given server may know that 'File' exists but not actually have direct access to a copy of the underlying data.

- Files are immutable, enforced through checksumming, making synchronization

between Librarians uni-directional and much simpler.

- Instances of the Librarian server are free-standing and are loosely-coupled,

meaning that long outages or network downtime do not cause significant issues.

The Librarian[1] is made up of five major pieces:

1. A FastAPI[2] server that allows access through a REST HTTP API.
2. A database server (postgres[3] in production, SQLite[4] for

development) to track state and provide atomicity.

1. A background task[5] thread that performs data-intensive operations

like checksumming, local cloning, and scheduling of transfers.

1. A background transfer thread that manages the transfer queue and communicates

with Globus to query status.

1. A Python and associated command-line client for interacting with the API.

[1] https://github.com/simonsobs/librarian
[2] https://fastapi.tiangolo.com
[3] https://www.postgresql.org
[4] https://www.sqlite.org
[5] https://schedule.readthedocs.io/en/stable/index.html

Crucially, the background threads can directly access the database, without having to interact with the HTTP server. This reduces the level of 'CRUD' (create, read, update, delete) API code that is required for the project significantly, by allowing (e.g.) locking of database rows currently being transferred directly, rather than having this be handled through a series of API calls. These background tasks are scheduled automatically using the lightweight `schedule` library.

The Librarian is a specialized tool that excels in reproducing the data and layout of a POSIX-compatible filesystem on a system with downstream nodes in a loosely-coupled manner, all whilst maintaining tight control on *how* data is transferred. The Librarian is open source and available through GitHub[6], under the BSD-2-Clause license.

### 3.1. Technology Choices

For this project, it was crucial that we used the Python language, as this is the lingua franca of the collaboration; all members have at least some knowledge of Python. No other language comes close (by a significant margin), and all analysis tools use Python. In addition, Globus provide a complete source development kit for Python[7]. As our analysis tools will eventually communicate with the Librarian, and its REST API, it also made sense to employ pydantic[8] to develop the communication schema. By using pydantic, we could ensure that responses and requests were serialized and de-serialized by the exact same models, reducing both development time and errors significantly. With pydantic models as our base data structure, it was natural to choose FastAPI for the web server component.

In an ideal case, we would have used SQLModel to further reduce code duplication in the database layer. However, at the outset of the project, SQLModel[9] was still in very early development, and as such it made sense to use SQLAlchemy[10] as our object-relational mapping (ORM) to translate database operations to object manipulation.

### 3.2. Service Layout

The Librarian service is relatively simple; it allows for the ingestion of 'files' (which can themselves be directories containing many files) into a unified namespace, and this name-

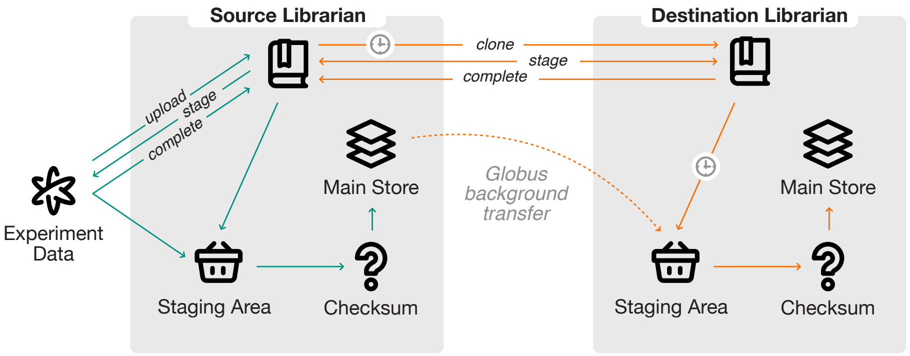


**Figure 2.** *Showing data ingest (green) and cloning between Librarian instances (orange). Data flows from the experiment, through a staging area, and to a main store, before being picked up by the background task (clocks) and copied by Globus in the background to the remote staging area. On the remote end, once the data is marked as being staged, the background task picks it up, checksums it, and it is marked as stored. More information on this process is available in the main text.*

[6] https://github.com/simonsobs/librarian
[7] https://globus-sdk-python.readthedocs.io/en/stable/index.html
[8] https://docs.pydantic.dev/latest/
[9] https://sqlmodel.tiangolo.com
[10] https://www.sqlalchemy.org

space can be synchronised with other sites using Globus. In this section we will give a brief overview of the key abstractions at place and the data flow within the application. This process is summarised in Figure 2.

In contrast to many other tools, and to the 'HERA Librarian' which preceeded the Librarian, files carry minimal science metadata along with them through the system. We found that metadata systems attached to data flow and management systems often lacked necessary features and caused significant complications (such as needing frequent database migrations) when coupled. As such, we chose to explicitly delegate metadata management to other subsystems within the project, focusing only on file transfers.

### 3.2.1. *User and Librarian Management:*

During the initial setup of the system, an administrator user is provisioned to facilitate further management tasks. This primary administrator has the ability to create additional user accounts through the `librarian` command-line tool.

Both 'users' (i.e. those interacting with the system to ingest data) and 'Librarians' (other copies of the application running on other systems) need accounts. These accounts can be configured with different levels of permissions to suit various needs. Specifically, accounts can be granted full administrator privileges, read and append only privileges, or callback-only privileges. Callback-only privalages are crucial for remote sites like telescopes, as they ensure that downstream Librarians only have extremely limited access to both the underlying data and its associated metadata. To ensure security, user passwords are salted and hashed in the database using the Argon2[11] algorithm. The API employs HTTP Basic Authentication for user verification and access control.

### 3.2.2. *Storage Management:*

In the Librarian system, storage is abstracted into entities known as 'stores'. These stores are provisioned during the setup process and can be migrated with a server restart. Each store comprises two components: a staging area for ingested files and a main store area, which has a global namespace for permanently storing data. While stores can have any underlying structure, they must provide methods for both ingestion and egress of data. In all current use cases, we employ a 'local store', which is a thin wrapper around a POSIX filesystem.

Stores provide two methods for data ingestion: transfer managers and async transfer managers. Transfer managers are synchronous, and for the 'local store', this involves a simple local file copy. Async transfer managers, on the other hand, are asynchronous and are used for inter-Librarian transfers, typically employing Globus for this purpose. Individual storage items are referred to as 'files', which can denote either individual files or directories.

Stores on the same device log all the 'instances' of 'files' they contain in the database. Additionally, all files can have 'remote instances', which are known instances of files located on another Librarian.

### 3.2.3. *Data Ingestion:*

Data ingestion follows a systematic process using accounts that have read and append privileges. Initially, a request to upload is made to the Librarian web server prompting it to create a temporary UUID-named directory in the staging area. Simultaneously, the client computes a checksum for the file. The server then provides this UUID and a set of transfer managers to the client.

[11] https://pypi.org/project/argon2-cffi/

Next, the client selects the most appropriate transfer manager to copy the file to the staging area. Once the file transfer is complete, the client informs the server of the completion. The server then verifies that the checksum matches the one provided in the upload request. If the checksums are consistent, the server ingests the file into the storage area.

From the client's perspective, the upload is extremely simple:

```python
from hera_librarian import LibrarianClient
from hera_librarian.settings import client_settings
from pathlib import Path

client_info = client_settings.connections.get(
    "my_librarian_name"
)

client = LibrarianClient.from_info(client_info)

client.upload(
    Path("name_of_a_file_on_disk.txt"),
    Path("/hello/world/this/is/a/file.txt")
)
```

This then leads to a synchronous uploading of the local file to the global namespace, with `upload` returning once this is complete and, crucially, the upload is verified (via a checksum) by the server. For most applications, this fire-and-forget technique is all that is required; if `upload` returns successfully, the file is guaranteed to have been correctly uploaded to the server and is ready for use and export.

### 3.2.4. *Data Cloning:*

Data can be cloned in two main ways: locally and remotely. Local cloning involves making a copy to a different store on the same Librarian, which is often used for SneakerNet. This process is straightforward and handled by a single task that creates a copy on the second device and updates the database.

Remote cloning, on the other hand, is a more complex, multi-stage process. Initially, files without remote instances on the target Librarian are collated into a list. Outgoing transfers are then created in the database to track progress. These transfers are grouped into batches of 128 to 1024 files to be packaged in a single Globus transfer, due to the limitation that there can only be 100 listed Globus transfers.

A request is made to the destination Librarian to batch stage clones of these files, creating a UUID directory for each. The batch transfer is placed in the queue on the local Librarian. The transfer thread picks up this batch and sends it using Globus, periodically polling for status updates on the transfer. Once the transfer is complete, the source Librarian notifies the destination Librarian that the files are staged. The 'receive clone' background task on the destination Librarian picks up the staged files and verifies them against known checksums. Once verified, the files are placed in the main store, and a callback is generated to the source Librarian to complete the transfer and register a new remote instance.

### 3.2.5. *Data Access:*

Data can be accessed through the Librarian, by querying it for the location of individual instances of a file. However, because our stores generally just wrap the POSIX filesystem, users typically already know a-priori where the files that they need to access are, through the use of other science product databases. As such, data access is generally as simple as opening the file; it is where users expect it to be.

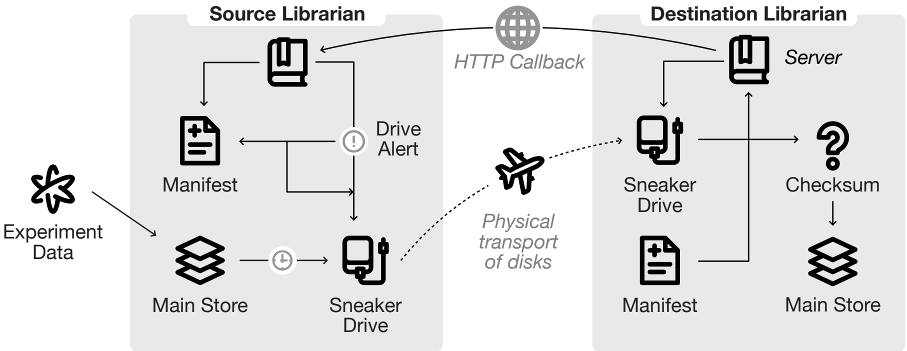


**Figure 3**. *Showing the SneakerNet flow between two Librarian instances. The sneaker drive is physically moved between computing sites, alongside a manifest of all files on the drive, to complete transfers in large batches.*

### *3.3. Data Down the Mountain*

In the specific case of an observatory, there is an additional challenge here: lines representing data flows in Figure 2 no longer simply show bytes sent over HTTP connections to highly available services, but instead represent potentially interruptible inter-continental data movement. In the specific case of the Simons Observatory, the main data flow from the telescope (which is at 5000 meters above sea level in the Atacama Desert) to the main computing center (NERSC, a national facility on the west coast of the U.S.) is fraught with challenges.

The main internet connection from the telescope to the nearest town was, for a long time, a low-bandwidth radio link. This has now been (thankfully) replaced with a high throughput fiber connection, but given that this link may go down at any time due to the remote placement of the site (and no backup), another way to offload the data is required. The observatory produces around 20 TB of data a week, which is much more than the 50 MBit/s radio link can transfer; either way, that bandwidth is required for other uses. This necessitates a more primitive solution: so-called ‘SneakerNet’ transfers, where data is carried by hand on physical media.

SneakerNet transfers are supported natively by the Librarian, and occur through the following steps (as shown in Figure 3):

- A new store is provisioned that refers to the mounted drive that will be

carried in the process.

- A local clone background task catches up on missing files to this drive and,

as new files come in, copies them too.

- Once full, the server sends an alert (implemented using Slack notifications)

to administrators.

- Administrators use the command-line Librarian tools to generate a ‘manifest’

of all files on the device: their checksums, names, etc.

- The drive is hand-carried to the destination, and ingested.
- The Librarian command-line tools are again used to ingest the manifest and

copied files.

- As the files are ingested, HTTP callbacks are made to the source Librarian to

convert the local ‘instance’ rows on the sneaker drive to ‘remote instance’ rows for the destination Librarian.

- The drive is re-formatted and carried back.

This is an unusual process, but comes in extremely handy in bandwidth constrained, remote, environments.

### *3.4. Testing*

Testing complex data flows like those represented in the Librarian is a notoriously difficult task. For instance, testing the components that make up the SneakerNet workflow means being able to test two interacting web-servers, each backed by a database, with several additional threads running in the background. To aid with this goal, we make heavy use of dependency injection and integration testing, alongside more traditional unit testing.

For services that cannot be 'easily' replicated, like Globus, we make sure to always use dependency injection. Dependency injection is a technique where downstream functions and objects 'receive' the things that they require as arguments or parameters. One significant example where this is used in the Librarian is the transfer managers: by having the server tell the client how it can move data, you can return appropriate managers for testing. Instead of using a Globus transfer (which requires setting up an external server, and registering it with Globus), one can instead return a transfer manager that simply wraps `rsync` or a local copy instead, and test all the same code paths except the one line that actually moves the data. The component that performs the Globus transfer is then tested separately in a more appropriate environment, ensuring that it conforms to the exact same specification as the local copy manager. Another example is our database: thanks to SQLAlchemy, and appropriate design patterns, we can very easily swap out our production postgres database for SQLite during development, enabling much more simple testing patterns.

In an attempt to avoid 'dependency hell', we have developed a testing platform that heavily leverages `pytest-xprocess`[12] and pytest fixtures. Using this tool, one can create a real, live, running Librarian server in a separate process. This is used to test the interaction between client and server, and for end-to-end tests of the SneakerNet and inter-Librarian cloning process. Because this server is running locally, it makes it possible to directly query the database for testing, which we have found to be invaluable. The only drawback to this approach is that it is not possible to get test coverage for the lines that are performed inside the `xprocess` fork.

### *3.5. Deployment Details*

As we have made a relatively straightforward selection of technologies, and the fact that the Librarian is a small application (less than 10'000 lines), deployment is a simple process. In addition, it makes it straightforward to explain the code paths that are taken by the Librarian.

Currently, we deploy Librarian using Docker. We require one container for the FastAPI server, and one for postgres, which are usually linked using helm charts (at NERSC) or docker compose (on other systems). We provide access to the database for Grafana dashboards, and have Slack integrations for both logging and notifications.

The focus on simplicity that we have made for the Librarian has made deployment simple; system administrators within the academic community are very familiar with Globus, and have been happy to assist with deployments of Librarian orchestration framework.

[12] https://pytest-xprocess.readthedocs.io/en/latest/

## 4. Conclusions

In this paper, we addressed the significant challenges faced by large academic collaborations in managing and transferring massive datasets across distributed sites. We evaluated existing data management tools such as Globus Flows, Rucio, iRODS, and Datalad, identifying their limitations in terms of automation, complexity, and compatibility with academic workflows.

Our solution, the Librarian was developed to meet these specific needs. By integrating with Globus, Librarian enables efficient, asynchronous data transfers. The system also supports SneakerNet, which is essential for environments with limited or intermittent connectivity like our observatory, facilitating physical data transfers through portable storage.

The use of widely adopted technologies such as Python, FastAPI, and SQLAlchemy ensures that Librarian is robust, scalable, and user-friendly. Its design aligns well with the decentralized and diverse infrastructure typical of academic projects, providing a practical and efficient method for handling the immense data flows inherent in modern 'big science' endeavors.

Librarian has demonstrated its effectiveness through its deployment at the Simons Observatory, highlighting its potential as a versatile and reliable data management solution for large-scale scientific collaborations. We provide Librarian as free, open source, software to the community.

### Acknowledgements

JB acknowledges support from NSF grant AST-2153201. This material is based upon work supported by the National Science Foundation under Grant Nos. 1636646 and 1836019 and institutional support from the HERA collaboration partners. This research is funded in part by the Gordon and Betty Moore Foundation through Grant GBMF5212 to the Massachusetts Institute of Technology.

### References

[1] P. Ade *et al.*, "The Simons Observatory: science goals and forecasts," |*jcap*, vol. 2019, no. 2, p. 56, 2019, doi: 10.1088/1475-7516/2019/02/056.

[2] I. Foster, "Globus Online: Accelerating and Democratizing Science through Cloud-Based Services," *IEEE Internet Computing*, vol. 15, no. 3, pp. 70–73, 2011, doi: 10.1109/MIC.2011.64.

[3] B. Allen *et al.*, "Software as a service for data scientists," *Commun. ACM*, vol. 55, no. 2, pp. 81–88, 2012, doi: 10.1145/2076450.2076468.

[4] M. Barisits *et al.*, "Rucio: Scientific Data Management," *Computing and Software for Big Science*, vol. 3, no. 1, p. 11, 2019, doi: 10.1007/s41781-019-0026-3.

[5] R. Consortium, "iRODS." [Online]. Available: https://irods.org/

[6] J. Hess, "git-annex." [Online]. Available: https://git-annex.branchable.com/

[7] Y. O. Halchenko *et al.*, "DataLad: distributed system for joint management of code, data, and their relationship," *Journal of Open Source Software*, vol. 6, no. 63, p. 3262, 2021, doi: 10.21105/joss.03262.

[8] D. R. DeBoer *et al.*, "Hydrogen Epoch of Reionization Array (HERA)," |*pasp*, vol. 129, no. 974, p. 45001, 2017, doi: 10.1088/1538-3873/129/974/045001.

[9] P. La Plante, P. K. G. Williams, and J. S. Dillon, "Developing a Real-Time Processing System for HERA," *URSI Radio Science Letters*, vol. 2, p. 41, 2020, doi: 10.46620/20-0041.

[10] P. La Plante *et al.*, "A Real Time Processing system for big data in astronomy: Applications to HERA," *Astronomy and Computing*, vol. 36, p. 100489, 2021, doi: 10.1016/j.ascom.2021.100489.

Open Access | https://doi.org/10.25080/NRPV2311

SciPy 2024
July 8 - July 14, 2024

Proceedings of the 23nd
Python in Science Conference
ISSN: 2575-9752

# Any notebook served: authoring and sharing reusable interactive widgets

Trevor Manz[1], Nils Gehlenborg[1], and Nezar Abdennur[2,3]

[1]Department of Biomedical Informatics, Harvard Medical School, Boston, MA, USA, [2]Department of Genomics and Computational Biology, UMass Chan Medical School, Worcester, MA, USA, [3]Department of Systems Biology, UMass Chan Medical School, Worcester, MA, USA

## Abstract

The open-source Jupyter project has fostered a robust ecosystem around notebook-based computing, resulting in diverse Jupyter-compatible platforms (e.g., JupyterLab, Google Colab, VS Code). Jupyter Widgets extend these environments with custom visualizations and interactive elements that communicate directly with user code and data. While this bidirectional communication makes the widget system powerful, its architecture is currently tightly coupled to platforms. As a result, widgets are complex and error-prone to author and distribute, limiting the potential of the wider widget ecosystem. Here we describe the motivation and approach behind the *anywidget* project, a specification and toolset for portable and reusable web-based widgets in interactive computing environments. It ensures cross-platform compatibility by using the web browser's built-in module system to load these modules from the notebook kernel. This design simplifies widget authorship and distribution, enables rapid prototyping, and lowers the barrier to entry for newcomers. Anywidget is compatible with not just Jupyter-compatible platforms but any web-based notebook platform or authoring environment and is already adopted by other projects. Its adoption has sparked a widget renaissance, improving reusability, interoperability, and making interactive computing more accessible.

**Keywords** Computational Notebooks, Jupyter, Widgets, Python, JavaScript, Data Visualization, Interactive Computing

**Published** Jul 10, 2024

**Correspondence to**
Trevor Manz
trevor_manz@g.harvard.edu

## 1. Introduction

Computational notebooks combine live code, equations, prose, visualizations, and other media within a single environment. The Jupyter project [1], [2] has been instrumental the success of notebooks, which have become the tool of choice for interactive computing in data science, research, and education. Key to Jupyter's widespread adoption is its modular architecture and standardization of interacting components, which have fostered an extensive ecosystem of tools that reuse these elements. For example, the programs responsible for executing code written in notebooks, called **kernels**, can be implemented by following the Jupyter Messaging Protocol [3]. This design allows users to install kernels for various different languages and types of computation. Similarly, Jupyter's open-standard notebook format (`.ipynb`) ensures that notebooks can be shared and interpreted across different platforms [4].

Jupyter's modular architecture has also supported innovation in **notebook front ends** — the user interfaces (UIs) for editing and executing code, as well as inspecting kernel outputs. The success of the classic Jupyter Notebook [1] spurred the development of several similar Jupyter-compatible platforms (JCPs), such as JupyterLab, Google Colab, and Visual Studio Code. These platforms provide unique UIs and editing features while reusing Jupyter's other standardized components. This interoperability allows users to choose the platform that best suits their needs, while retaining a familiar interactive computing experience with the ability to share notebooks. Furthermore, the separation of computation from UI offers users a wide selection of both front ends and kernels. However, the proliferation of JCPs has led to significant challenges for Jupyter Widgets, a key component of interactive user interfaces in Jupyter.

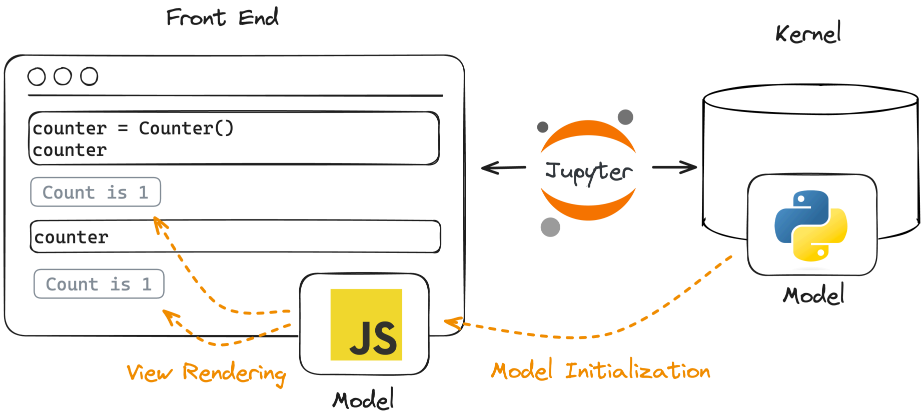
**Jupyter Widgets** extend notebook outputs with interactive views and controls for objects residing in the kernel [5]. For instance, the ipywidgets library, besides defining the widget communication protocol, provides basic form elements like buttons, sliders, and dropdowns to adjust individual variables. Other community projects offer interactive visualizations for domain-specific needs, such as 3D volume rendering (ipyvolume), biological data exploration [6], [7], [8], and mapping (ipyleaflet, pydeck, lonboard), which users can update by executing other code cells or interact with in the UI to update properties in the kernel.

Widgets are unique among Jupyter components in that they consist of two separate programs — kernel-side code and front-end code — that communicate directly via custom messages Figure 1, rather than through a mediating Jupyter process. With widgets, communication is bidirectional: a kernel action (e.g., the execution of a notebook cell) can update the UI, such as causing a slider to move, while a user interaction (e.g., dragging a slider), can drive changes in the kernel, like updating a variable. This two-way communication distinguishes widgets from other interactive elements in notebook outputs, such as HTML displays, which cannot communicate back and forth with the kernel.

Widgets are intended to be pluggable components, similar to kernels. However, only the protocol for communication between kernel and front-end widget code, known as the Jupyter Widgets Message Protocol, is standardized. Critical components, such as the distribution format for front-end modules and methods for discovering, loading, and executing these modules, remain unspecified. As a result, JCPs have adopted diverse third-party module formats, installation procedures, and execution models to support widgets. These inconsistencies place the onus on widget authors to ensure cross-JCP compatibility.

## Note

In this paper, we define a Jupyter Widget as consisting of a pair of programs: kernel-side and front-end, which communicate via the Jupyter Widget Message Protocol. While Python-only widget classes exist, which wrap other Jupyter Widgets as dependencies,



```
counter = Counter()
counter
```

Count is 1

```
counter
```

Count is 1

**Figure 1**. *Jupyter Widget conceptual architecture.*

> this paper focuses on those with dedicated front-end code, where the challenges described apply.

JCPs load front-end widget code by searching in various external sources, such as local file systems or Content Distribution Networks (CDNs) while kernel-side (Python) code loads and runs in the kernel Figure 2. These access patterns split the distribution of custom widgets between Python and JavaScript package registries, complicating releases and requiring widget authors to understand both packaging ecosystems. This division creates challenges, especially in shared, multi-user environments like JupyterHub. Since kernels cannot host static assets, users cannot independently install widgets or manage versions. Instead, widget front-end code must be pre-installed on the Jupyter webserver, typically by an administrator. Consequently, users are restricted to administrator-installed widgets and versions, unable to upgrade or add new ones independently.

These limitations make widget development complex and time-consuming, demanding expertise in multiple domains. They make user experiences across JCPs frustrating and unreliable. The high barrier to entry discourages new developers and domain scientists from contributing to widgets, limiting growth and diversity in the ecosystem. This leaves a small group of authors responsible for adapting their code for cross-JCP compatibility, hindering widget reliability and maintainability.

## 2. Methodology

The **anywidget** project simplifies the authoring, sharing, and distribution of Jupyter Widgets by (i) introducing a standard for widget front-end code based on the web browser's

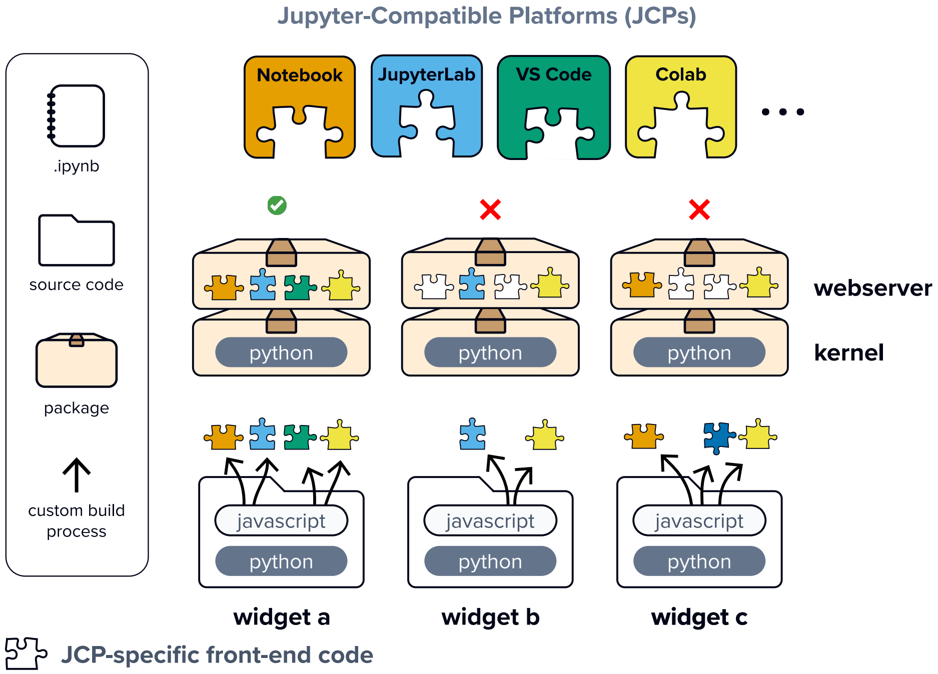


**Figure 2.** *Without anywidget, to ensure compatibility, authors must transform their JavaScript code for each JCP. Since JCPs load front-end widget code from a webserver rather than the kernel, widget front-end and Python code must also be packaged separately and installed consistently on any given platform. Missing puzzle pieces represent missing front-end extensions for specific target JCPs. The misshapen blue piece represents an incorrectly built or broken extension. Both situations contribute to fragmented JCP support.*

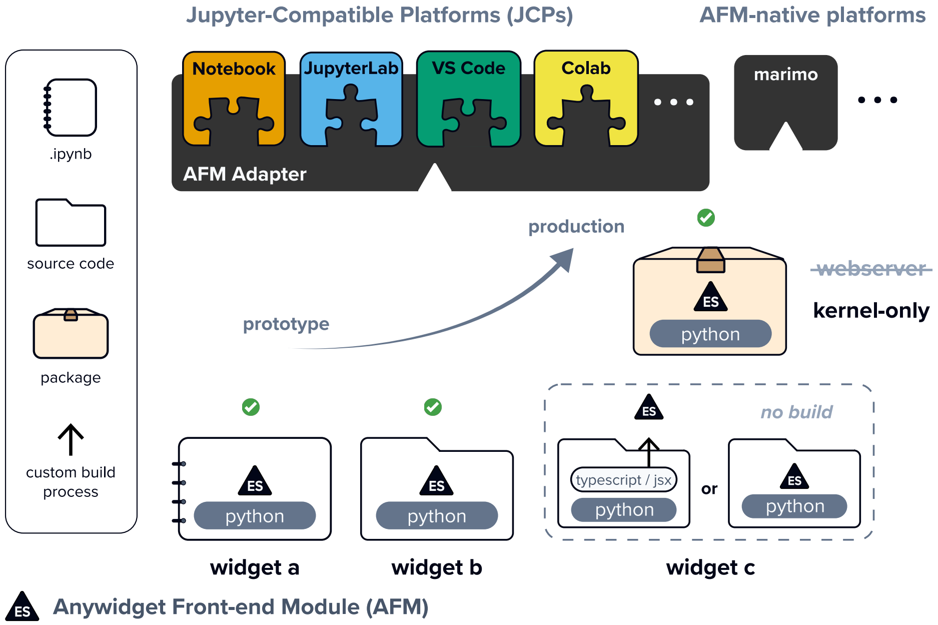


**Figure 3.** *With anywidget, developers author a single, standard portable ES module (AFM), which is loaded from the kernel and executed using the browser's native module system. For existing JCPs, anywidget provides a front-end adapter to load and execute these standardized modules, while new platforms can add native AFM support directly. Widget kernel-side code and AFM can be run directly from within notebooks, from source files, or distributed as single Python packages. With anywidget, each JCP consumes the same AFM, and widget authors can compose an AFM directly or target AFM through a build step to take advantage of advanced tools.*

native module system, (ii) loading these modules from the kernel, and (iii) providing the necessary "glue code" to adapt existing JCPs to load and execute these components Figure 3. This separation of concerns allows widget authors to write portable code that runs consistently across JCPs without manual installation steps.

Packaging custom Jupyter Widgets is complex due to the need to adapt JavaScript source code for various module systems used by JCPs. Initially, JavaScript lacked a built-in module system, leading JCPs to adopt diverse third-party solutions. Without a standardized widget front-end format, authors transform their code for each JCP. In the context of Jupyter Notebook and JupyterLab, this problem is described in the Jupyter Widgets documentation [9] as follows:

> Because the API of any given widget must exist in the kernel, the kernel is the natural place for widgets to be installed. However, kernels, as of now, don't host static assets. Instead, static assets are hosted by the webserver, which is the entity that sits between the kernel and the front-end. This is a problem because it means widgets have components that need to be installed both in the webserver and the kernel. The kernel components are easy to install, because you can rely on the language's built-in tools. The static assets for the webserver complicate things, because an extra step is required to let the webserver know where the assets are.

ECMAScript (ES) modules, introduced in 2015, are an official standard for packaging JavaScript code for reuse [10]. While most JCPs predate its standardization, ES modules are universally supported by browsers today. By adopting ES modules, anywidget is able to use the browser's native import mechanism to load and execute widget front-end code from the Jupyter kernel, thereby bypassing those used by JCPs and eliminating third-party dependencies. This approach not only overcomes many development challenges, but also

```
export default {
  initialize({ model }) {
    // Set up shared state or event handlers.
    return () => {
      // Optional: Called when the widget is destroyed.
    }
  },
  render({ model, el }) {
    // Render the widget's view into the el HTMLElement.
    return () => {
      // Optional: Called when the view is destroyed.
    }
  }
}
```

**Figure 4.** *An anywidget front-end module (AFM) with initialization and rendering lifecycle methods. For familiarity, AFM methods use naming conventions from traditional Jupyter Widgets; however, AFM narrows down the APIs provided to these methods, making it easier to load and execute AFMs in new environments.* Methods*: The `initialize` and `render` methods correspond to different stages in the widget's lifecycle. During model initialization, a front-end model is created and synchronized with the kernel. In the rendering stage, each notebook cell displaying a widget object renders an independent view based on the synced model state.* Arguments*: The interface of `model` is restricted to a minimal set of methods for communicating with the kernel (retrieving, updating, and responding to value changes). The `el` argument is a standard web `HTMLElement`.*

eliminates installation procedures for front-end code. Consequently, developers can prototype and share widgets directly within notebooks, making them more reliable and easier to use.

An anywidget front-end module (AFM) is an ES module with a `default` export defining widget behavior. This export includes lifecycle methods, or "hooks," for managing a widget's lifecycle stages: initialization, rendering, and destruction Figure 4. AFM lifecycle methods receive the interfaces required for kernel communication and notebook output modifications as arguments, rather than creating them internally or relying on global variables provided by the JCP. This practice, known as dependency injection [11], improves AFM portability by making integration interfaces explicit. New runtimes can support AFMs by implementing the required APIs, and existing JCPs can refactor their internals without breaking existing (any)widgets. For projects that want to take advantage of advanced front-end tooling, anywidget also provides authoring utilities to write AFMs such as "bridge" libraries for popular web frameworks [12].

Widget authorship is particularly challenging due to the need to integrate front-end code that communicates with kernel code in a heterogeneous set of environments. The anywidget project addresses these challenges by focusing on the standardization, development, and distribution of widget front-end modules, including the associated API to communicate with a computational kernel. The anywidget Python package serves as an adapter library that turns each JCP into an AFM-compatible host environment. Finally, the anywidget project provides additional tooling to help developers evolve their prototypes into mature packages [12].

An additional goal for an interactive computing paradigm like widgets is composability. For example, ipywidgets provides utilities to link widgets together or lay out grids of widgets on the page. These primitives allow widgets to derive from others, enabling reuse of front-end and kernel integrations in new ways. Importantly, Jupyter Widgets authored with the anywidget Python package extend from ipywidgets, making them interoperable with core ipywidgets and traditional custom widgets. By aligning with ipywidgets, anywidget promotes composability and prevents fragmentation of the ecosystem.

## 3. Features

Adhering to predictable standards benefits both developers and end users in many other ways beyond JCP interoperability, such as…

### *3.1. Web Over Libraries*

Front-end libraries change rapidly and often introduce breaking changes, whereas the web platform remains more backward-compatible. Traditional Jupyter Widgets require extensions from UI libraries provided by JCPs, coupling widget implementations to third-party frameworks. In contrast, AFM defines a minimal set of essential interfaces focused on (1) communicating with the kernel and (2) modifying notebook output cells, without dictating models for state or UI. It allows widgets to be defined without dependencies, reducing boilerplate and preventing lock-in. Authors may still use third-party JavaScript libraries or tooling, but these are no longer necessary for JCP compatibility, publishing, or user installation.

### *3.2. Rapid Iteration*

The web ecosystem's adoption of ES modules has led to new technologies that enhance developer experience and enable rapid prototyping. One such innovation is hot module replacement (HMR), a method that uses the browser's module graph to dynamically update applications without reloading the page or losing state. Since traditional Jupyter Widgets rely on legacy module systems, they cannot benefit from HMR and instead require full page clearance, reload, and re-execution to see changes during development. By contrast, anywidget is able to provide opt-in HMR, implemented through the Jupyter messaging protocol, in order to support live development of custom widgets without any front-end tooling. For example, adjusting a widget's appearance, such as a chart's color scheme, updates the view instantly without re-executing cells or refreshing the page.

### *3.3. Progressive Development*

Anywidget makes it possible to prototype widgets directly within a notebook since all widget code is loaded from the kernel. Custom widgets can start as a few code cells and transition to separate files, gradually evolving into standalone scripts or packages – just like kernel-side programs Figure 3. In contrast, developing traditional Jupyter Widgets is a cumbersome process limited to the Jupyter Notebook and JupyterLab platforms. It involves using a project generator [13], [14] to bootstrap a project with over 50 files, creating and installing a local Python package, bundling JavaScript code, and manually linking build outputs to install extensions [15]. By removing these barriers, anywidget accelerates development and allows prototypes to grow into robust tools over time. In fact, as of January 2024, the JavaScript cookiecutter [13] project has been deprecated and directs users to use anywidget instead.

### *3.4. Simplified Publishing*

Serving AFMs and other static assets from the kernel removes the need to publish widget kernel-side and front-end code separately and coordinate their releases. For example, many JCPs retrieve traditional widget Javascript code from npm, misusing the registry for distributing specialized programs rather than reusable JavaScript modules. Instead, with anywidget, developers can publish a widget (kernel-side module, AFM, and stylesheets) as a unified package to the distribution channels relevant to the kernel language, such as the Python Package Index. Consolidating the distribution process this way greatly simplifies publishing and discovery.

## 4. Impact and Outlook

Anywidget fills in the specification gaps for Jupyter Widgets by embracing open standards and carefully separating developer concerns. It defines an API for authoring portable and reusable widget components that decouples widget authorship from JCPs, resulting in multiple downstream benefits. First, anywidget—not widget authors—ensures compatibility and interoperability across existing JCPs, and authors can focus on important features rather than wrestle with build configuration and tooling. Anywidget's adapter layer simply loads and executes standardized front-end modules, meaning that the compatibility afforded does not introduce overhead to the runtime performance of a widget itself. A widget authored with anywidget has the same performance characteristics as a widget tediously (and correctly) authored using the traditional Jupyter Widgets system. Second, by circumventing bespoke JCP import systems and loading web-standard ES modules from the kernel, anywidget does away with manual installation steps and delivers an improved developer experience during widget authorship. Third, anywidget unifies and simplifies widget distribution. Widgets can be prototyped and shared as notebooks, or mature into `pip`-installable packages and distributed like other tools in the Python data science ecosystem. End users benefit from standardization because widgets are easy to install and behave consistently across platforms.

Since its release, anywidget has led to a proliferation of widgets and a more diverse widget ecosystem Figure 5. New widgets range from educational tools for experimenting with toy datasets (e.g., DrawData) to high-performance data visualization libraries (e.g., Lonboard, Jupyter-Scatter [16], Mosaic [17]) and research projects enhancing notebook interactivity (e.g., Persist [18], cev [19]). Many of these tools use anywidget's binary data transport to enable efficient interactive visualization with minimal overhead by avoiding JSON serialization. Existing widget projects have also migrated to anywidget (e.g., higlass-python, ipyaladin) and other libraries have introduced or refactored existing widget functionality to use anywidget (e.g., Altair [20]) due to the simplified distribution and authoring capabilities.

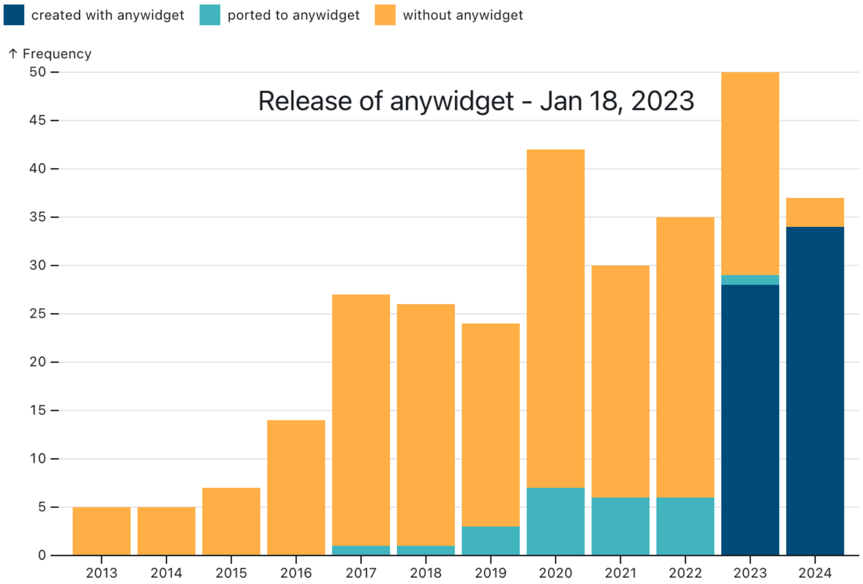


**Figure 5**. *Custom Jupyter Widgets per year as of September 2, 2024. Projects are tracked in a semi-automated process, combining daily code searches against GitHub with manual verification, at https://github.com/manzt/anywidget-usage. See repository for details and contribution guidelines.*

The portable widget standard also extends the anywidget ecosystem to platforms beyond Jupyter. Popular web frameworks and dashboarding libraries such as Voila, Panel, Shiny for Python, and Solara support Jupyter Widgets, and therefore also allow users to embed anywidgets in standalone web applications. Efforts are underway to add more specialized, built-in support for AFM as well. For example, marimo, a new reactive notebook for Python, is standardizing its third-party plugin API on AFM, allowing anywidgets to run natively without additional "glue code." Developers of the Panel web-application framework are also exploring deeper integration with AFM to enable reuse with their kernel-side reactivity systems.

AFM's standardization of widget front-end code extends reusability beyond the Python ecosystem. Its API structure is intentionally agnostic to backend processing, enabling creative and dynamic ways of hosting AFM-based widgets. Many simple widgets can function without any compute backend, while more complex ones can be upgraded with a computational backend when necessary. This flexibility allows for innovative hosting solutions across diverse platforms, from fully static web pages to full-stack web applications. By decoupling front-end interactivity from backend complexity, AFM empowers developers to create scalable, platform-independent scientific visualizations and tools, adaptable to a wide range of computational requirements and user scenarios.

A current limitation of anywidget is that using relative file imports for local dependencies requires bundling into a single ES module, which introduces a build step and prevents shipping AFM source code directly. This build step can be avoided by using a single file and loading third-party dependencies via URLs, though larger projects may still benefit from bundling. Notably, traditional widgets always require a build step and mandate using a specific bundler (Webpack) with a bespoke multi-JCP configuration. In contrast, anywidget makes bundling optional and targets a single standard ES module, which can be produced easily from a variety of bundlers. Future browser support for importmaps may enable local dependencies without bundling, simplifying anywidget development further as web standards evolve.

Tools for data visualization and interactivity have greater impact when compatible with more platforms, but achieving compatibility involves trade-offs [21]. The full capabilities of the widget system, such as bidirectional communication, are often inaccessible to authors due to development difficulty and maintenance efforts. Adopting standards can minimize these impediments, enabling both broad compatibility and advanced capabilities for users. A recent article enumerates these challenges and advocates for standardized solutions to democratize the creation of notebook visualization tools across notebook platforms [21]. Anywidget addresses this by introducing a standard that removes friction in widget development and sharing, making authorship practical and accessible.

## Acknowledgements

We thank Talley Lambert for his contributions to the project, and David Kouril for his suggestions on the manuscript and figures. We also thank the anywidget community as well as members of the Abdennur and HIDIVE labs for helpful discussions.

### Funding

TM, NG, and NA acknowledge funding from the National Institutes of Health (UM1 HG011536, OT2 OD033758, R33 CA263666, R01 HG011773).

## References

[1] T. Kluyver *et al.*, “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., 2016, pp. 87–90. doi: 10.3233/978-1-61499-649-1-87.

[2] B. E. Granger and F. Pérez, “Jupyter: Thinking and Storytelling With Code and Data,” *Comput. Sci. Eng.*, vol. 23, no. 2, pp. 7–14, 2021, doi: 10.1109/MCSE.2021.3059263.

[3] “Jupyter Client documentation.” [Online]. Available: https://jupyter-client.readthedocs.io/en/stable/

[4] “nbformat documentation.” [Online]. Available: https://nbformat.readthedocs.io/en/stable/format_description.html

[5] “Jupyter documentation.” [Online]. Available: https://docs.jupyter.org/en/latest/projects/architecture/content-architecture.html

[6] T. Manz, S. L'Yi, and N. Gehlenborg, “Gos: a declarative library for interactive genomics visualization in Python,” *Bioinformatics*, vol. 39, no. 1, p. btad50, 2023, doi: 10.1093/bioinformatics/btad050.

[7] T. Manz *et al.*, “Viv: multiscale visualization of high-resolution multiplexed bioimaging data on the web,” *Nat. Methods*, vol. 19, no. 5, pp. 515–516, 2022, doi: 10.1038/s41592-022-01482-7.

[8] M. S. Keller, I. Gold, C. McCallum, T. Manz, P. V. Kharchenko, and N. Gehlenborg, “Vitessce: a framework for integrative visualization of multi-modal and spatially-resolved single-cell data,” 2021. doi: 10.31219/osf.io/y8thv.

[9] “Low Level Widget Explanation.” [Online]. Available: https://ipywidgets.readthedocs.io/en/stable/examples/Widget%20Low%20Level.html

[10] yu Shu-Guo, M. Ficarra, K. Gibbons, and E. community, “ECMAScript® Language Specification.” [Online]. Available: https://262.ecma-international.org/14.0/

[11] M. Fowler, “Inversion of Control Containers and the Dependency Injection Pattern.” [Online]. Available: https://martinfowler.com/articles/injection.html

[12] T. Manz, N. Abdennur, and N. Gehlenborg, “anywidget: reusable widgets for interactive analysis and visualization in computational notebooks.” OSF Preprints, 2024. doi: 10.31219/osf.io/tw9sg.

[13] “widget-cookiecutter.” [Online]. Available: https://github.com/jupyter-widgets/widget-cookiecutter

[14] “widget-ts-cookiecutter.” [Online]. Available: https://github.com/jupyter-widgets/widget-ts-cookiecutter

[15] “Anywidget documentation cookiecutter comparison.” [Online]. Available: https://anywidget.dev/blog/introducing-anywidget/#a-solution-with-crumbs

[16] F. Lekschas and T. Manz, “Jupyter Scatter: Interactive Exploration of Large-Scale Datasets,” *Journal of Open Source Software*, vol. 9, no. 101, p. 7059, 2024, doi: 10.21105/joss.07059.

[17] J. Heer and D. Moritz, “Mosaic: An Architecture for Scalable & Interoperable Data Views,” *IEEE Trans. Vis. Comput. Graph.*, vol. 30, no. 1, pp. 436–446, 2024, doi: 10.1109/TVCG.2023.3327189.

[18] K. Gadhave, Z. Cutler, and A. Lex, “Persist: Persistent and reusable interactions in computational notebooks,” 2023. doi: 10.31219/osf.io/9x8eq.

[19] T. Manz, F. Lekschas, E. Greene, G. Finak, and N. Gehlenborg, “A General Framework for Comparing Embedding Visualizations Across Class-Label Hierarchies,” 2024, doi: 10.31219/osf.io/puxnf.

[20] J. VanderPlas *et al.*, “Altair: Interactive Statistical Visualizations for Python,” *Journal of Open Source Software*, vol. 3, no. 32, p. 1057, 2018, doi: 10.21105/joss.01057.

[21] Z. J. Wang, D. Munechika, S. Lee, and D. H. Chau, “SuperNOVA: Design Strategies and Opportunities for Interactive Visualization in Computational Notebooks,” in *Extended Abstracts of the 2024 CHI Conference on Human Factors in Computing Systems*, in CHI EA '24. New York, NY, USA, 2024, pp. 1–17. doi: 10.1145/3613905.3650848.

Open Access | https://doi.org/10.25080/TCFJ5130

# ITK-Wasm

**Matthew McCormick**[1], and **Paul Elliott**[1]

[1] Kitware, Inc.

SciPy 2024

July 8 - July 14, 2024

Proceedings of the 23rd Python in Science Conference

ISSN: 2575-9752

**Published** Jul 10, 2024

**Correspondence to**
Matthew McCormick
matt@mmmccormick.com

## Abstract

In recent years, WebAssembly (Wasm) has emerged as a widely-supported technology that offers high performance, compact binary size, support for multiple languages, hardware independence, security, and universal platform support, enabling developers to bring near-native speeds and portability to applications for the web and beyond. ITK-Wasm brings WebAssembly's capabilities to scientific computing by combining the Insight Toolkit (ITK) and WebAssembly to enable high-performance spatial analysis across programming languages and hardware architectures.

In the scientific Python ecosystem, ITK-Wasm packages work in a web browser via Pyodide but also in system-level environments through the WebAssembly System Interface (WASI). ITK-Wasm bridges WebAssembly with scientific Python through simple, fundamental Python and NumPy-based data structures and Pythonic function interfaces. These interfaces can be accelerated through graphics processing units (GPU) or neural processing unit (NPU) implementations when available.

Beyond Python, ITK-Wasm's integration of the WebAssembly Component Model launches scientific computing into a new world of interoperability, enabling the creation of accessible and sustainable multi-language projects that are easily distributed anywhere.

**Keywords** reproducibility, sustainability, accessibility, interoperability, numpy, polyglot, spatial computing, visualization, imaging, meshes, pointsets, itk, webassembly, wasm

## 1. Introduction

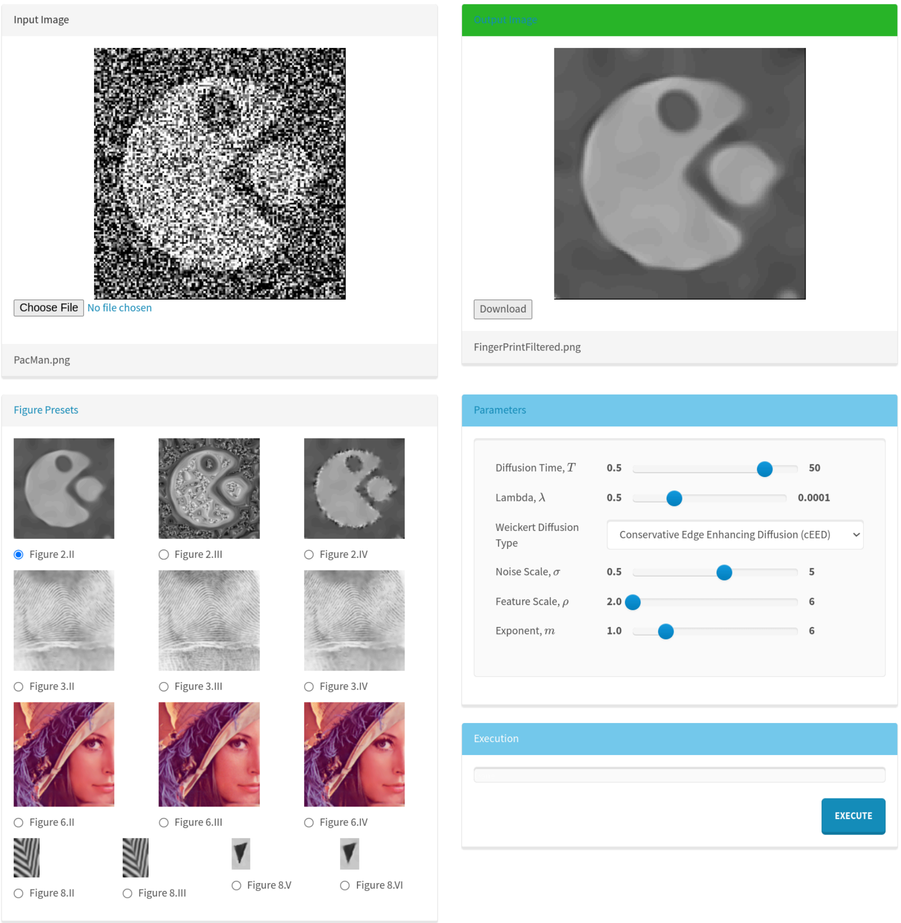
### 1.1. Motivation

In the quest for **enhanced interoperability and sustainability in scientific computing**, *WebAssembly (Wasm)* emerges as a transformative technology. Wasm offers a universal, efficient compilation target, enabling high-performance computing across varied programming languages and hardware architectures [1], [2], [3]. This innovation is pivotal for scientific research, where analytical interoperability, tool sustainability, and computational efficiency are paramount. Wasm's journey began with asm.js and evolved through Wasm and the WebAssembly System Interface (WASI).

### 1.2. Brief history of Wasm

Asm.js was introduced by Alon Zakai via the Emscripten toolchain as a subset of JavaScript designed for high performance [4]. It allowed developers to write code in languages like C and C++, compile it to asm.js, and run it in the browser with near-native performance. Asm.js achieved this by using a statically-typed subset of JavaScript that enabled optimizations by the JavaScript engine.

The predecessor to ITK-Wasm, *ITK.js*, supported compilation of C/C++ code into asm.js to enable reproducible, sustainable scientific computing in a web browser. An illustrative ITK.js application are interactive figures to replicate results from an open science article



Diffusion Time, $T$ 0.5 50

Lambda, $\lambda$ 0.5 0.0001

Weickert Diffusion Type: Conservative Edge Enhancing Diffusion (cEED)

Noise Scale, $\sigma$ 0.5 5

Feature Scale, $\rho$ 2.0 6

Exponent, $m$ 1.0 6

**Figure 1.** *Open science interactive figures built with ITK.js, the predecessor to ITK-Wasm [6]. The sustainability and accessibility of this interactive open science figure are remarkable, thanks to the utilization of open web technologies. These technologies, based on open standards, ensure compatibility across different systems and long-term support. This web application has continued to function flawlessly for over nine years without any maintenance, demonstrating its resilience and sustainability. It is hosted for free, with computations running directly on the reader's system, eliminating the need for server-side resources. The application requires zero installation and can be accessed with just a few clicks, allowing users to easily reproduce results or apply the technique to their own data.*

on an imaging denoising technique called anisotropic non-linear diffusion [5]. Results on a simple webpage, hosted for free on GitHub Pages, are dynamically generated by the reader's web browser, using the same code and data presented in the article. Presets load input data and set analysis parameters to dynamically reproduce the article's figures. Additionally, readers can modify parameters to observe their effects or run the algorithm on their own image data.

The key performance improvement with asm.js was its ability to utilize the JavaScript engine's just-in-time (JIT) compilation to execute code faster than traditional JavaScript. However, asm.js had limitations, including verbose code and the overhead of JavaScript's garbage collection and dynamic typing.

WebAssembly emerged from the limitations of asm.js. Announced in 2015 and reaching its initial MVP (Minimum Viable Product) in 2017, Wasm provides a compact binary format that could be executed at near-native speed [3], [7]. This innovation marked a significant leap in web performance, enabling complex applications to run efficiently in the browser.

Key features of Wasm include [7]:

*Security*

- Safe to execute
- Maintains the sandboxing paradigms of web browsers

*Portability*

- Language-, hardware-, and platform-independent
- Deterministic and easy to reason about
- Simple interoperability with the Web platform

*Speed*

- Fast to execute
- Maximally compact
- Easy to decode, validate and compile
- Easy to generate for producers
- Streamable and parallelizable

The *WebAssembly System Interface (WASI)*, introduced by the WebAssembly Community Group, extends Wasm's capabilities beyond the browser [8]. WASI provides a standardized system interface for WebAssembly, enabling it to interact with the underlying operating system. This development was crucial for running Wasm in desktop, server, and other non-browser environments.

Key features of WASI include:

- File system access
- Network access
- A modular architecture

With WASI, WebAssembly can be used to develop portable, high-performance applications that run anywhere. A plethora of WASI runtimes are available that vary in their focus, such as embedding in programming languages, specialized hardware such as field-programmable gate arrays (FPGAs), embedded devices, security, speed, or high performance computing (HPC) environments [9], [10], [11].

### *1.3. Wasm and scientific computing*

Throughout its evolution, WebAssembly has focused on performance improvements. Some notable advancements relevant to scientific computing include:

- **Bulk memory operations**: efficient copy and movement of data in memory
- **SIMD support**: Single Instruction, Multiple Data (SIMD) capabilities, allowing Wasm to perform parallel operations on multiple data points simultaneously with specialized instruction support available on modern CPUs
- **Multithreading support**: support for operating system threads and atomics for CPU parallelism

The evolution of WebAssembly from asm.js to Wasm to WASI has been marked by continuous improvements in performance, interoperability, and support for a wide range of programming languages and deployment environments. This journey has transformed WebAssembly into a versatile and powerful technology, capable of running high-performance applications anywhere, from the browser to the cloud.

Wasm has been embraced in commercial and industrial contexts for web applications, game development, edge computing, and server-side computing. However, its adoption in scientific computing has been more limited. This is partly due to the established reliance specialized software stacks in the scientific community. Additionally, the integration of Wasm into existing scientific workflows requires overcoming challenges related to data interoperability, toolchain compatibility, and the inertia of entrenched computational practices.

Enter **ITK-Wasm**, a pioneering resource that marries the Insight Toolkit (ITK) and open standards to seemlessly integrate Wasm for high-performance scientific spatial analysis or visualization[12], [13], [14]. ITK-Wasm supports both Emscripten-based Wasm in a web browser or WASI-SDK Wasm for system-level environments. ITK-Wasm is crafted to adhere to Wasm community standards, thereby facilitating the creation of Wasm modules that are **simple, performant, portable, modular, and interoperable**.

ITK-Wasm provides infrastructure that empowers research software engineers to:

- *Build scientific C/C++ codes to Wasm*
- *Generate idiomatic programming language bindings, packages, and documentation*
- *Bridge Wasm with*
    - *Local filesystems*
    - *Canonical scientific programming data interfaces such as NumPy arrays*
    - *Traditional scientific file formats with an emphasis on multi-dimensional spatial data*
- *Transfer data efficiently in and out of the Wasm runtime*
- *Support asynchronous and parallel execution of processing pipelines in way that is easy to understand and implement*

## 2. Methods

### *2.1. Overview*

ITK-Wasm provides powerful, joyful tooling for scientific computation in Wasm through a number of distinct but related parts.

1. C++ core tooling
2. Build environment Docker images
3. A Node.js CLI to build Wasm, generate language bindings, run tests, and publish packages
4. Small, language-specific libraries that facilicate idiomatic integration
5. Scientific file format support
6. Artificial intelligence and the semantic web integration

This tooling supports a straightforward programming model that aligns with functional programming paradigms and leverages Wasm's simple stack-based virtual machine and Component Model architecture. All tooling is built around two key concepts:

1. **Interface Types**: High-level, programming-language types for scientific computing, derived from Wasm's low-level types.
2. **Processing Pipelines**: Functions implemented in Wasm modules that operate on these interface types.

### *2.2. C++ core*

ITK-Wasm's C++ core tooling provides:

1. Fundamental numerical methods and multi-dimensional scientific data structures
2. An elegant, modern interface to create processing pipelines
3. A bridge to interoperable web techologies

4. A bridge to Web3 and traditional desktop computing

These are embodied in the C++ core with:

1. ITK
2. CLI11
3. Glaze
4. libcbor

The Insight Toolkit (ITK) is an open-source, cross-platform library that provides developers with an extensive suite of software tools based on a proven, spatially-oriented architecture for processing scientific data in two, three, or more dimensions [M. McCormick, X. Liu, J. Jomier, C. Marion, and L. Ibanez [13]]. ITK includes fundamental numerical libraries, such as Eigen. ITK's C++ template-based architecture inherently helps keep Wasm modules small while enabling the compiler to add extensive performance optimizations. The *itk-wasm* GitHub repository is also an ITK Remote Module, `WebAssemblyInterface`, that implements Wasm-interface specific functionality. As an ITK Remote Module, wasm interface capabilities are available in an easily consumed library form. To access wasm interface functionality in a build configuration,

```
find_package(ITK
  COMPONENTS
    WebAssemblyInterface
    # Other desired components
  )
include(${ITK_USE_FILE})

add_executable(a-wasm-pipeline ${srcs})
target_link_libraries(a-wasm-pipeline PUBLIC ${ITK_LIBRARIES})
```

Glaze provides an elegent C++ JSON interface. This library was integrated as it is not only extremely fast but also small as a header-only library, which is critical for efficient WebAssembly deployment. The ability to read and write interface types to files, providing a bridge to Web3 and traditional desktop computing, is built on libcbor, which is another tiny footprint library.

Wasm module C++ processing pipelines are written with CLI11's simple and intuitive argument parsing interface [15]. A C++ Wasm processing pipeline is defined with the familiar context of a command line executable. Processing pipelines can be built with native binary toolchains into usable command line executables, which facilites development and debugging. CLI11 extensions for the interface types and parsing enable an efficient embedding interface in addition to the command line interface. All modules support an `--interface-json` flag that outputs a description of the module's interface for binding generation.

## *2.3. Build environment Docker images*

Build environment Docker images encapsulate

1. The ITK-Wasm C++ core
2. An Emscripten or WASI toolchain
3. Additional Wasm tools and configurations

These `itkwasm/emscripten` and `itkwasm/wasi` Docker images are *dockcross* images – Docker images with pre-configured C++ cross-compiling toolchains that enable easy-application, reproducible builds, and a clean separation of the build environment, source tree, and build artifacts [16].

These images include not only the CMake pre-configured toolchains, but pre-built versions of the ITK-Wasm C++ core. Moreover, Wasm tools for optimization, debugging, system execution, and testing are bundled. A number of build and system configurations are included to for optimized and debuggable builds.

### 2.4. *Command line interface (CLI)*

An `itk-wasm` Node.js *command line interface (CLI)* drives

- Wasm module builds
- Generation of language bindings and language package configurations
- Testing for Wasm binaries

New projects are typically created with the `create-itk-wasm` CLI:

```
npx create-itk-wasm
```

The `create-itk-wasm` tool, which can be used interactively or programmatically, will generate C++ code with the required ITK-Wasm CLI11 interfaces, other support configuration files such as CMake build configuration scripts, a *package.json* file, language binding generation, and testing.

### 2.5. *Language-specific libraries and idiomatic bindings*

Small, language-specific libraries are used by generated bindings to provide simple, clean, performant, and idiomatic interfaces in the host languages.

In TypeScript / JavaScript, this is the NPM `itk-wasm` package and in Python this is the PyPI `itkwasm` package.

TypeScript / JavaScript bindings and packages are generated from Emscripten toolchain builds of ITK-Wasm. TypeScript bindings are generated along with an NPM *package.json* to support package builds and deployment. Bindings are generated that support both browser-based execution and server-side execution in Node.js. Build scripts are provided to build TypeScript to JavaScript and also generate a demo app with an HTML interface. JavaScript bindings load Zstandard-compressed versions the Wasm modules on-demand in a web worker to support progressive and performant execution.

Python packages for new modules are generated for both system execution and web brower-execution. In the browser, Pyodide-compatible packages provide client-side web app scripting in Python, including via PyScript, and sustainable, scalable Jupyter deployments via JupyterLite. At a system level, Linux, macOS, and Windows operating systems are supported on x86_64 and ARM via wasmtime-py.

#### 2.5.1. *Python environment dispatch:*

Bindings produce a primary, pip-installable Python package. In browser environments, this will pull a corresponding Emscripten-enabled Python package. For system Python distributions, this will bring in a corresponding WASI-enabled Python package. When GPU-accelerated implementations of functions are available in other packages along with required hardware and software, simply pip-installing the accelerator package will cause function calls to invoke accelerated overrides registered with modern package metadata, Figure 2.

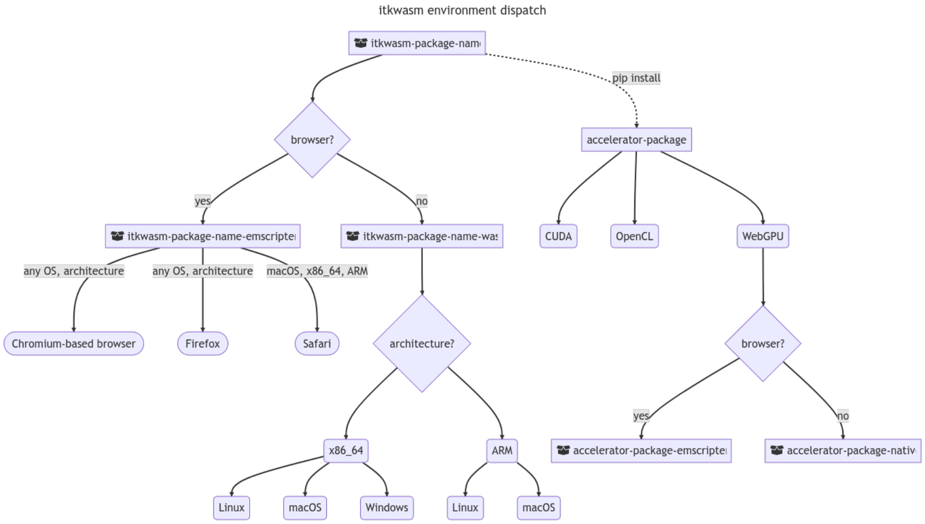


**Figure 2.** *Cross-platform, cross-environment support with optional non-Wasm accelerator packages is made possible by a generated environment dispatch Python package, a WASI-based Python package, and a Pyodide package. The Pyodide package does not have typical Pyodide-Emscripten ABI limitations due to the application of principals from the Wasm Component Model.*

### 2.5.2. *Browser and system APIs:*

While synchronous functions are available in system packages, browser packages provide asynchronous functions for non-blocking, performant execution in the JavaScript runtime event loop. These functions are called with modern Python's async / await support.

For example, to install the itkwasm-compress-stringify package:

**System**

```
pip install itkwasm-compress-stringify
```

**Browser**

In Pyodide, e.g. the Pyodide REPL or JupyterLite,

```
import micropip
await micropip.install('itkwasm-compress-stringify')
```

In the browser, call the async `*_async` function with the `await` keyword.

**System**

```
from itkwasm_compress_stringify import compress_stringify

data = bytes([33,44,55])
compressed = compress_stringify(data)
```

**Browser**

```
from itkwasm_compress_stringify import compress_stringify_async

data = bytes([33,44,55])
compressed = await compress_stringify_async(data)
```

### 2.6. *Traditional file format support*

Assistance for handling data serialized in file formats plays a crucial role in enabling comprehensive analysis using a variety of software tools.

ITK-Wasm offers IO modules designed to interact with various standard scientific image and mesh file formats. These modules allow for the loading of data into the language-native interface type bindings.

Scientific image file formats supported include:

- AIM,ISQ
- BioRad
- BMP
- DICOM
- DICOM Series
- ITK HDF5
- JPEG
- GIPL (Guy's Image Processing Lab)
- LSM
- MetaImage
- MGH
- MINC 2.0
- MRC
- NIfTI
- NRRD
- VTK legacy file format for images
- Varian FDF

Scientific mesh file formats supported include:

- BYU
- FreeSurfer surface, binary and ASCII
- OFF
- STL
- SWC Neuron Morphology
- OBJ
- VTK legacy file format for vtkPolyData

In addition to supporting external file formats, ITK-Wasm also introduces its own file formats. These ITK-Wasm file formats are optimized and offer a direct correspondence to spatial interface types, utilizing a straightforward JSON + binary array buffer format.

Execution pipeline WebAssembly modules only support ITK-Wasm formats by default – this ensures that size of the WebAssembly pipeline binary is minimal. When using ITK-Wasm pipelines on with the command line interface, Wasm modules from the image-io and mesh-io packages can transform data in other formats into the format supported by all ITK-Wasm modules.

ITK-Wasm formats can be output in a directory or bundled in a single `.cbor` file. The Concise Binary Object Representation (CBOR) format supports JSON + binary array data in the data schema, is extremely small and lightweight in implementations, has support across programming languages, is highly performant, and provides a link to Web3 storage mechanisms.

ITK-Wasm file formats are available in ITK-Wasm IO functions but also in C++ via the *WebAssemblyInterface* ITK module. This module can be enabled in an ITK build by setting the `-DModule_WebAssemblyInterface:BOOL=ON` flag in CMake. And, loading and conversion is also available native-binary Python bindings via the *itk-webassemblyinterface* Python package.

### *2.7. Artificial Intelligence and the Semantic Web*

An ITK-Wasm LinkML [17] model provides FAIR definitions of the interface types that enable high-performance, portable, sustainable, and reproducible spatial analysis.

The interface types include:

- *BinaryFile* - Representation of a binary file on a filesystem. For performance reasons, use BinaryStream when possible, instead of BinaryFile.
- *BinaryStream* - Representation of a binary stream. For performance reasons, use BinaryStream when possible, instead of BinaryFile.
- *Image* - Representation of an N-dimensional scientific image.
- *JsonCompatible* - A type that can be represented in JSON.
- *Mesh* - Representation of an N-dimensional mesh.
- *PointSet* - Representation of a set of N-dimension points.
- *PolyData* - Representation of a polydata, 3D geometric data for rendering that represents a collection of points, lines, polygons, and/or triangle strips.
- *TextFile* - Representation of a text file on a filesystem. For performance reasons, use TextStream when possible, instead of TextFile.
- *TextStream* - Representation of a text stream. For performance reasons, use TextStream when possible, instead of TextFile.
- *Transform* - Representation of a parametric spatial transformation that can be applied to an Image, Mesh, PointSet, PolyData.

This model, combined with ITK-Wasm's architecture, can perform analysis and visualization using natural language inputs provided to large-language artificial intelligence models. LinkML's Pydantic models and TypeScript models enable interfaces like the Bioimage Chatbot [18], [19] to semantically understand the needs of biologists without programming knowledge, allowing the system to execute desired operations or generate scripts for batch execution.

## 3. Results

### *3.1. Example application: generation of multiscale OME-Zarr images*

A notable application of ITK-Wasm is the generation of Figure 3, a cloud-optimized bioimaging format with broad international adoption [21], [22]. To generate OME-Zarr's multiscale representation of multidimensional bioimages, anti-aliasing filters must be applied.

As detailed by the Nyquist-Shannon Sampling Theorem, high frequency content in an image must be reduced before downsampling to avoid Figure 5.

In the NGFF-Zarr package [23], ITK-Wasm anti-aliasing filters efficiently produce OME-Zarr images suitable for Pyodide, JupyterLite, traditional CPython environments, or analysis and visualization with other programming languages. While ITK-Wasm supports making general scientific C++ codes accessibly in wasm, in this example we will examine how the `itk::DiscreteGaussianImageFilter` is applied to address this problem [13], [24], [25].

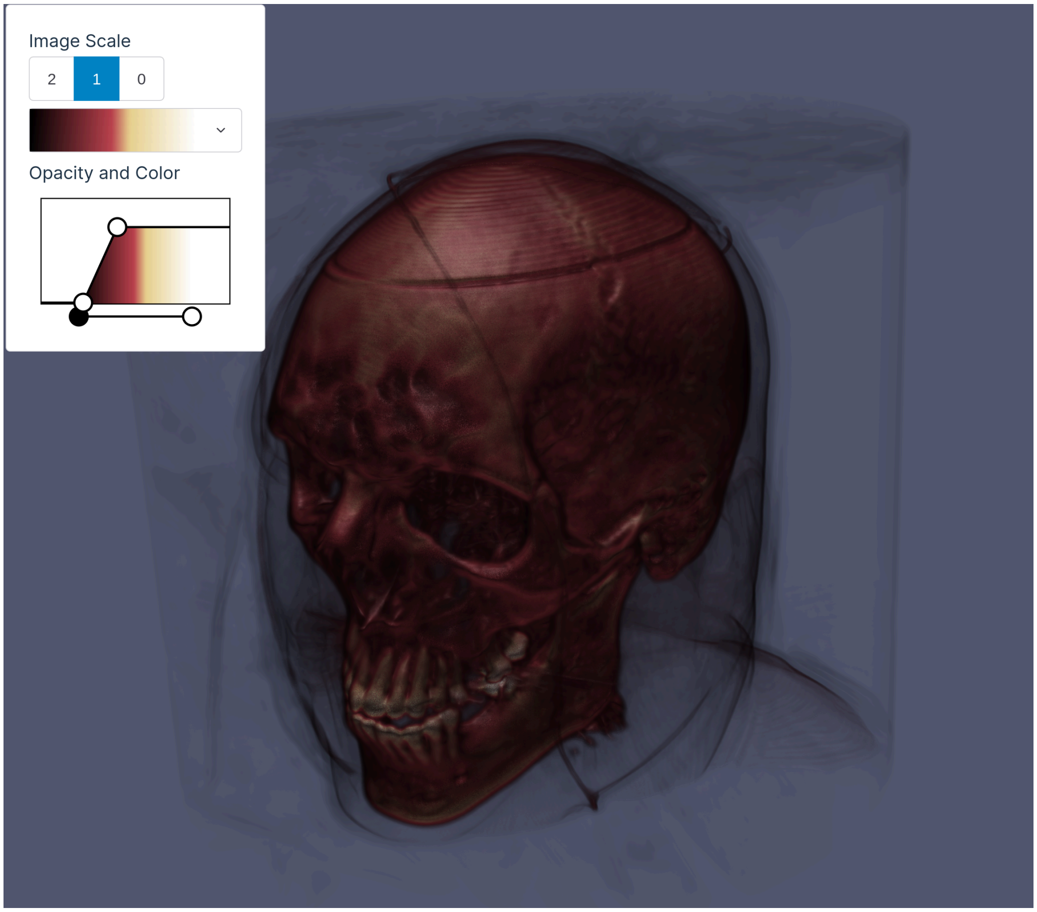


**Figure 3.** *Visible Male [20] frozen head computed tomography (CT) OME-Zarr volume, generated with ITK-Wasm. There are three resolution scales, which can be selected with the* Image Scale *buttons. Reduced resolutions are smoothed with a gaussian filter to avoid aliasing artifacts.*

We apply the N-dimensional gaussian convolution filter:

$$G(\boldsymbol{x};\sigma) = \frac{1}{(2\pi\sigma^2)^{n/2}} \exp\left(-\frac{\mid \boldsymbol{x} \mid^2}{2\sigma^2}\right) \tag{1}$$

Where:

- $G(\boldsymbol{x};\sigma)$ is the Gaussian filter function
- $\boldsymbol{x}$ is an n-dimensional vector representing spatial coordinates
- $\sigma$ is the standard deviation of the Gaussian distribution
- $n$ is the number of dimensions
- $\mid \boldsymbol{x} \mid^2$ is the squared Euclidean norm of $\boldsymbol{x}$

For downsampling, we use $\sigma^2 = (k^2 - 1^2)/\left(2\sqrt{2\ln(2)}\right)^2$ where $k$ is the downsampling factor, which is optimal [26].

### 3.2. C++ pipeline definition

The C++ wasm pipeline function, a pure function, is defined as a CLI11 executable [15]. It uses an `itk::wasm::Pipeline` interface definition that operates on `itk::wasm` interface types.

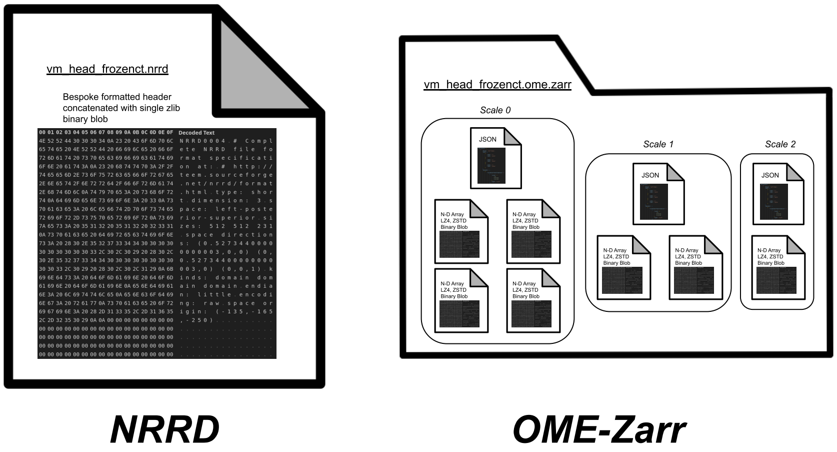

vm_head_frozenct.nrrd
Bespoke formatted header concatenated with single zlib binary blob
Decoded Text
vm_head_frozenct.ome.zarr
Scale 0
Scale 1
Scale 2
JSON
N-D Array LZ4, ZSTD Binary Blob
NRRD
OME-Zarr

**Figure 4.** *A comparison of NRRD (Nearly Raw Raster Data), a traditional scientific image file format to the OME-Zarr scientific image file format, a modern, web-friendly file format. NRRD is a single, monolithic file with header and image pixel data formats concatenated. The header has a bespoke format and the pixel data is often compressed with the zlib compression codec. In contrast, OME-Zarr has a hierarchical structure, sometimes encoded in folders and files. Metadata is stored in JSON files. Image pixel data is stored in separate N-dimensional chunks that are compressed with fast, high compression ratio modern codecs like LZ4 or Zstd. Additionally, the image is represented at multiple resolutions. OME-Zarr's chunk, highly-compressed, multiscale representation makes it ideal for uses cases like cloud-computing or extremely large images. However, this requires generation of the reduced resolution scales.*

```
int main(int argc, char * argv[])
{
  itk::wasm::Pipeline pipeline("downsample", "Apply a smoothing anti-alias filter and subsample the
input image.", argc, argv);

  return itk::wasm::SupportInputImageTypes<PipelineFunctor,
    uint8_t,
    int8_t,
    uint16_t,
    int16_t,
    uint32_t,
    int32_t,
    uint64_t,
    int64_t,
    float,
    double
    >
  ::Dimensions<2U, 3U, 4U, 5U>("input", pipeline);
}
```

Here `downsample` defines the name of the pipeline function. A description for the pipeline is also provided – this propagates to command line and language interface documentation.

In this function, we use the `itk::wasm::SupportInputImageTypes` utility to dispatch compile-time optimized pipeline based on the pixel type and dimension of the input image. This ensures excellent performance while limiting the wasm module binary size, critical for performance and distribution, to the code that is used by the pipeline.

Next, pipeline inputs, outputs, and parameters are defined:

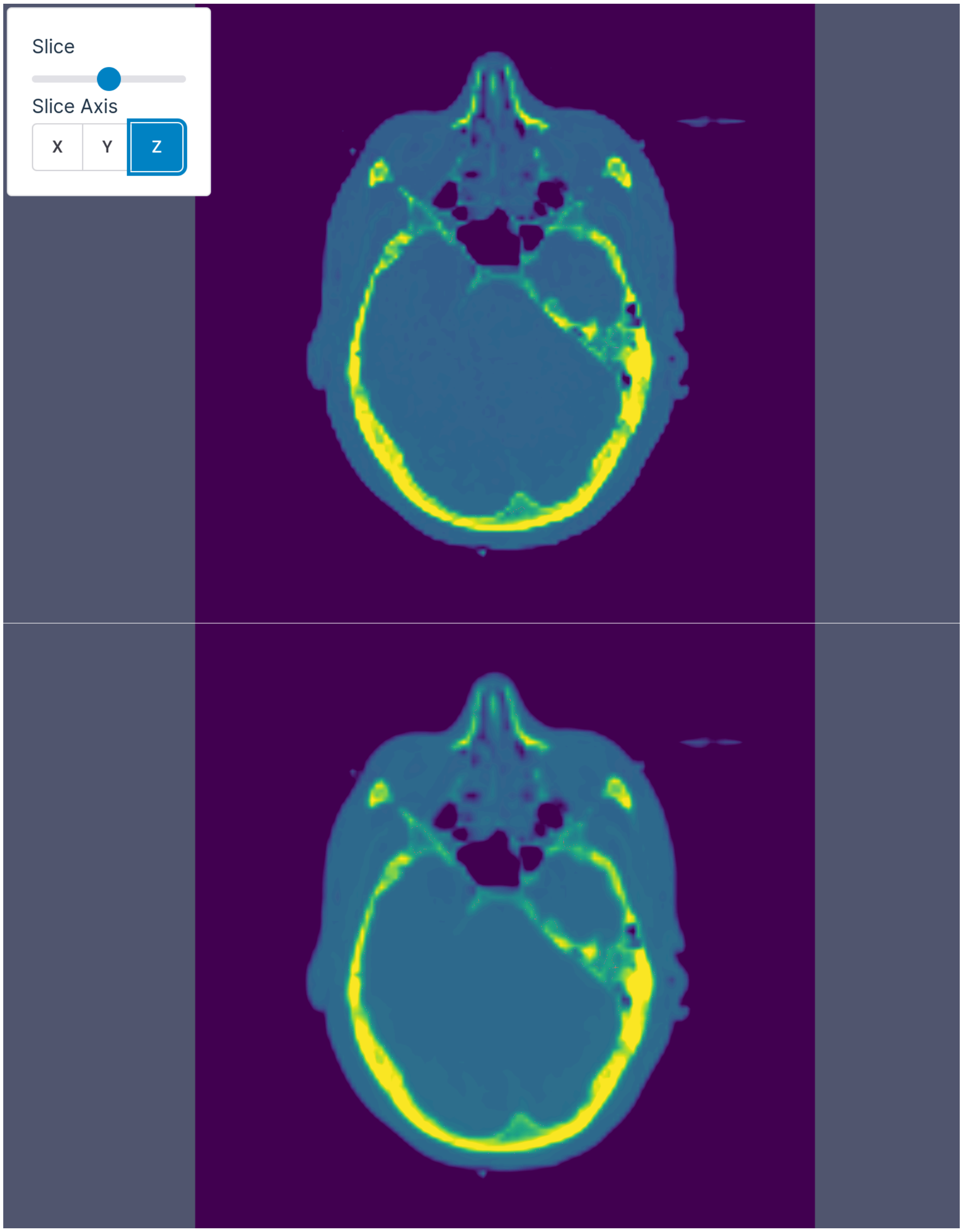


**Figure 5.** *Aliasing artifacts at the second resolution scale. With naive subsampling (top) aliasing artifacts introduce noise in the image at frequencies not supported by the sampling frequency. With gaussian anti-aliasing filtering prior to downsampling (bottom), signal fidelity is preserved.*

```
template<typename TImage>
class PipelineFunctor
{
public:
  int operator()(itk::wasm::Pipeline & pipeline)
  {
    using ImageType = TImage;
    constexpr unsigned int ImageDimension = ImageType::ImageDimension;

    using InputImageType = itk::wasm::InputImage<ImageType>;
```

```
InputImageType inputImage;
pipeline.add_option("input", inputImage, "Input image")
  ->required()->type_name("INPUT_IMAGE");

std::vector<unsigned int> shrinkFactors { 2, 2 };
pipeline.add_option("-s,--shrink-factors", shrinkFactors, "Shrink factors")
  ->required()->type_size(ImageDimension);

std::vector<unsigned int> cropRadius;
pipeline.add_option("-r,--crop-radius", cropRadius, "Optional crop radius in pixel units.")
  ->type_size(ImageDimension);

using OutputImageType = itk::wasm::OutputImage<ImageType>;
OutputImageType downsampledImage;
pipeline.add_option("downsampled", downsampledImage, "Output downsampled image")
  ->required()->type_name("OUTPUT_IMAGE");

ITK_WASM_PARSE(pipeline);
```

The types used are integers, floating point numbers, `std` containers of the same, or `itk::wasm` interface types. Long flags define parameter names in their language bindings, and their descriptions are propagated to their documentation.

The pipeline interface syntax can be generated from a set of interactive prompts provided by the `create-itk-wasm` CLI tool.

Once `ITK_WASM_PARSE(pipeline)` is called, input argument parsing and error handling is performed and the input pipeline options are populated with their values. During CLI execution, this means reading input files. When used with language bindings, files are not used and inputs are populated with in-memory content that was *lowered* into the wasm module with internal `itk_wasm*` functions.

Next comes the computational logic of the pipeline:

```
using GaussianFilterType = itk::DiscreteGaussianImageFilter<ImageType, ImageType>;
auto gaussianFilter = GaussianFilterType::New();
gaussianFilter->SetInput(inputImage.Get());

[...]
ITK_WASM_CATCH_EXCEPTION(pipeline, shrinkFilter->UpdateLargestPossibleRegion());

typename ImageType::ConstPointer result = shrinkFilter->GetOutput();
downsampledImage.Set(result);

return EXIT_SUCCESS;
```

In this example, we are using ITK library C++ functionality, but this can be arbitrary C++ code.

The `.Get()` and `.Set()` methods on the interface types supply the C++ interface to the input values for computation and outputs. When the output interface type’s destructors are called, they are written to files on disk in a CLI context or prepared for wasm module *lifting* in an embedded language context.

The build is configured with simple, standard CMake:

```
cmake_minimum_required(VERSION 3.16)
project(itkwasm-downsample LANGUAGES CXX)

find_package(ITK REQUIRED
 COMPONENTS
   WebAssemblyInterface
   ITKSmoothing
[...]

add_executable(downsample downsample.cxx)
target_link_libraries(downsample PUBLIC ${ITK_LIBRARIES})
```

The same C++ and CMake code can be used for a native toolchain along with the wasm toolchain builds. This facilitates rapid development and easy debugging with native development tooling.

Additionally, CTest tests can be defined for native or WASI execution, e.g.:

```
enable_testing()

add_test(NAME downsample
  COMMAND downsample
    ${CMAKE_CURRENT_SOURCE_DIR}/test/data/input/cthead1.png
    ${CMAKE_CURRENT_BINARY_DIR}/cthead1_downsampled.png
    --shrink-factors 2 2
    )
```

In the WASI case, ITK-Wasm enables execution via a wasm interpreter and by enabling interpreter access to local input and output file directories.

### *3.3. Command line invocation*

If our example `downsample` pipeline module is built into a native binary, help output is Figure 6:

```
❯ ./downsample --help
```

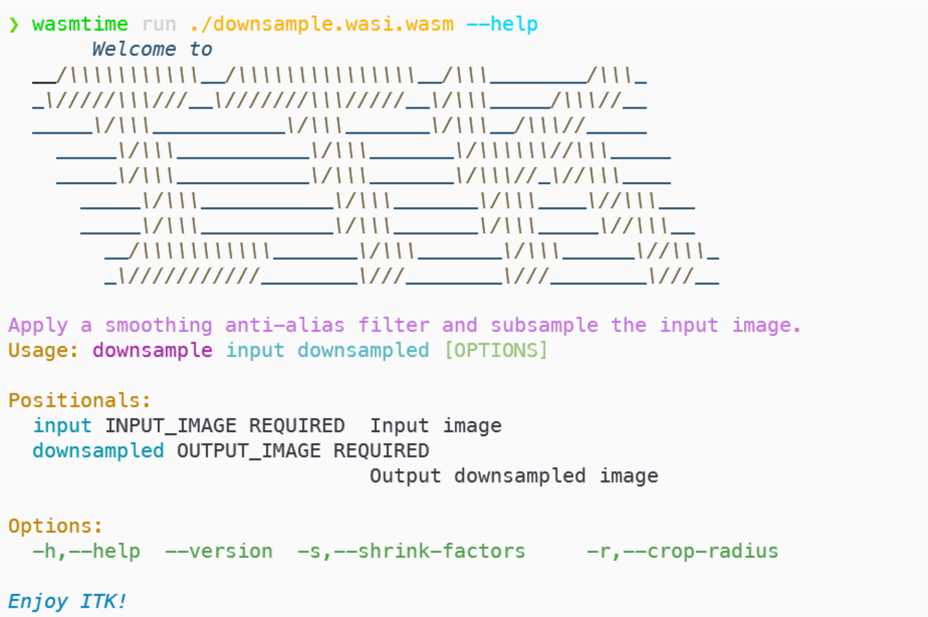
The equivalent invocation with the wasmtime wasm runtime for the WASI wasm module built from the same sources is:

```
❯ wasmtime run ./downsample.wasi.wasm --help
```

Where a native binary invocation is:

```
❯ ./downsample \
  ./vm_head.iwi.cbor ./downsampled.iwi.cbor \
  --shrink-factors 4 4 2
```

The equivalent wasm module invocation is:



```
❯ wasmtime run ./downsample.wasi.wasm --help
       Welcome to
  __/\\\\\\\\\\\__/\\\\\\\\\\\\\\\__/\\\________/\\\_
  _\/////\\\///__\///////\\\/////__\/\\\_____/\\\//__
  _____\/\\\___________\/\\\_______\/\\\__/\\\//_____
    _____\/\\\___________\/\\\_______\/\\\\\\//\\\_____
    _____\/\\\___________\/\\\_______\/\\\//_\//\\\____
      _____\/\\\___________\/\\\_______\/\\\____\//\\\___
      _____\/\\\___________\/\\\_______\/\\\_____\//\\\__
        __/\\\\\\\\\\\_______\/\\\_______\/\\\______\//\\\_
        _\///////////________\///________\///________\///__

Apply a smoothing anti-alias filter and subsample the input image.
Usage: downsample input downsampled [OPTIONS]

Positionals:
  input INPUT_IMAGE REQUIRED  Input image
  downsampled OUTPUT_IMAGE REQUIRED
                              Output downsampled image

Options:
  -h,--help  --version  -s,--shrink-factors    -r,--crop-radius

Enjoy ITK!
```

**Figure 6**. *Wasm module help invocation and generated help output.*

```
❯ wasmtime run --dir=./ -- ./downsample.wasi.wasm \
  ./vm_head.iwi.cbor ./downsampled.iwi.cbor \
  --shrink-factors 4 4 2
```

All ITK-Wasm pipeline modules also support an `--interface-json` flag, which allows a module to self-describe its interface for documentation and language binding generation, described in the following sections.

```
❯ wasmtime run ./downsample.wasi.wasm --interface-json
{
   "description": "Apply a smoothing anti-alias filter and subsample the input image.",
   "name": "downsample",
   "version": "0.1.0",
   "inputs": [
      {
         "description": "Input image",
         "name": "input",
         "type": "INPUT_IMAGE",
         "required": true,
         "itemsExpected": 1,
         "itemsExpectedMin": 1,
         "itemsExpectedMax": 1,
         "default": ""
      }
   ],
   "outputs": [
      {
         "description": "Output downsampled image",
         "name": "downsampled",
         "type": "OUTPUT_IMAGE",
         "required": true,
         "itemsExpected": 1,
         "itemsExpectedMin": 1,
         "itemsExpectedMax": 1,
         "default": ""
      }
   ],
   "parameters": [
    [...]
      {
         "description": "Shrink factors",
         "name": "shrink-factors",
         "type": "UINT",
         "required": true,
         "itemsExpected": 2,
         "itemsExpectedMin": 2,
         "itemsExpectedMax": 1073741824,
         "default": "[2,2]"
      },
    [...]
   ]
}
```

### *3.4. Generating TypeScript packages from ITK-Wasm modules*

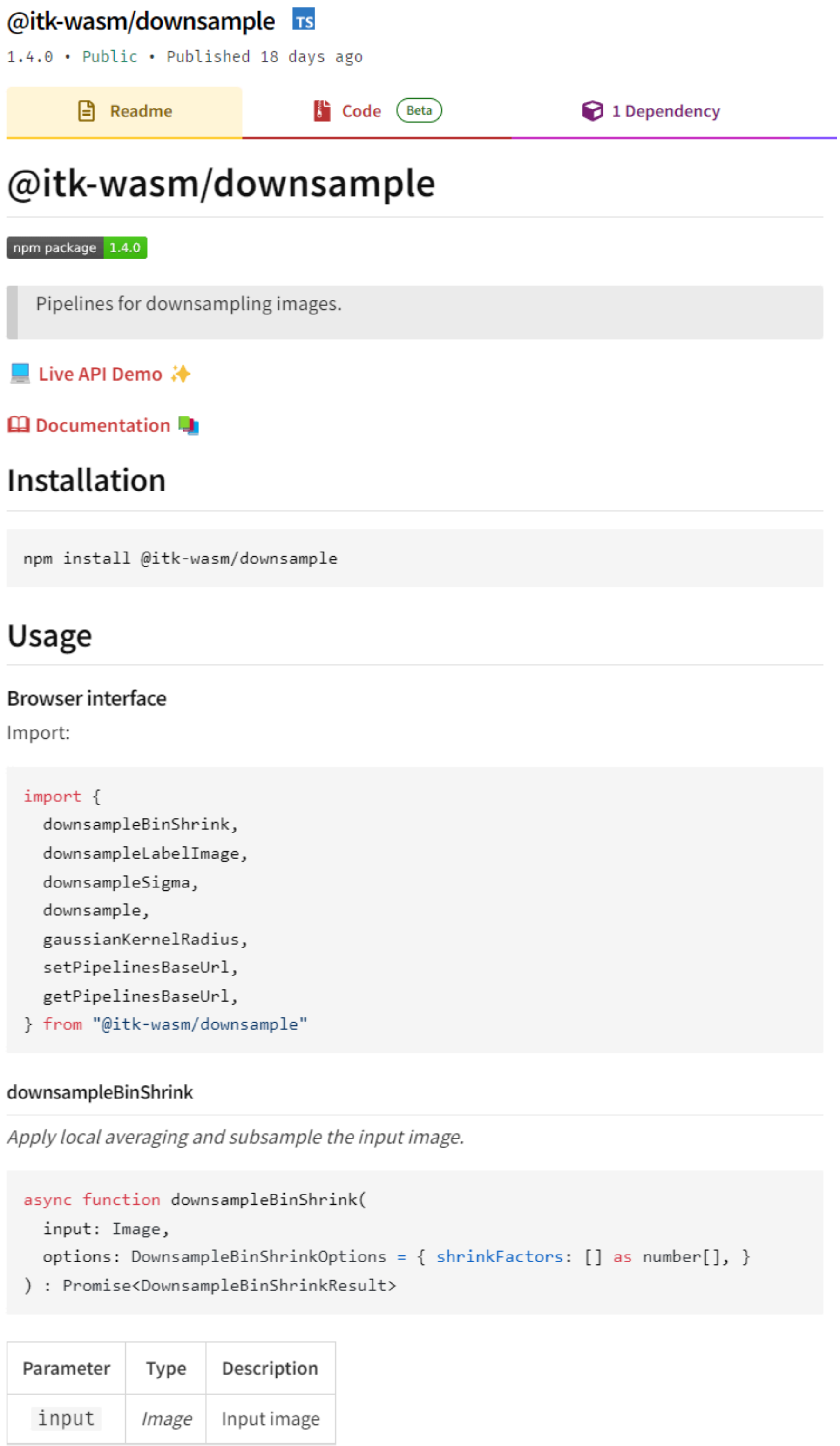
One of the toolchains that ITK-Wasm supports is Emscripten. To facilitate seamless integration with modern web development, we generate TypeScript packages from Emscripten-generated wasm modules. These packages include Node.js bindings for server-side execution and browser-compatible interfaces for client-side execution.

#### 3.4.1. *Node.js bindings:*

For server-side execution, our Node.js bindings provide direct access to the wasm module's functionality. Local filesystem paths are made available to the wasm module, enabling pipelines to operate on files stored on the server. This allows developers to leverage ITK-Wasm pipelines in a Node.js environment, ideal for tasks such as image processing or analysis on a server.

#### 3.4.2. *Browser bindings:*

In the browser, our bindings support pipelines that operate on files using the `File` object, as well as a custom `BinaryFile` object. The `BinaryFile` object consists of a string path and a `Uint8Array` binary, enabling efficient binary data transfer between the browser and wasm module. This allows developers to create web applications that interact with ITK-Wasm pipelines, enabling tasks such as image filtering and segmentation in the browser.



@itk-wasm/downsample

1.4.0 • Public • Published 18 days ago

# @itk-wasm/downsample

> Pipelines for downsampling images.

💻 Live API Demo ✨

📖 Documentation 📚

## Installation

```
npm install @itk-wasm/downsample
```

## Usage

### Browser interface

Import:

```
import {
  downsampleBinShrink,
  downsampleLabelImage,
  downsampleSigma,
  downsample,
  gaussianKernelRadius,
  setPipelinesBaseUrl,
  getPipelinesBaseUrl,
} from "@itk-wasm/downsample"
```

#### downsampleBinShrink

*Apply local averaging and subsample the input image.*

```
async function downsampleBinShrink(
  input: Image,
  options: DownsampleBinShrinkOptions = { shrinkFactors: [] as number[], }
) : Promise<DownsampleBinShrinkResult>
```

| Parameter | Type | Description |
|---|---|---|
| `input` | *Image* | Input image |

**Figure 7.** *JavaScript and TypeScript package* README *rendered on* npmjs.com.

#### 3.4.3. *TypeScript Interface Types:*

Our TypeScript interface types are designed to be idiomatic classes comprised of JSON-compatible JavaScript types and TypedArrays. This ensures seamless interaction between the wasm module and TypeScript code, enabling developers to write efficient, natural, and type-safe code.

#### 3.4.4. *WebWorker-based execution:*

To prevent interruption of the main user interface thread and support asynchronous background wasm compilation, pipelines are executed in a `WebWorker` when running in the browser. This ensures a responsive user experience while ITK-Wasm pipelines are executing.

#### 3.4.5. *Parallelism with WebWorkerPool:*

For tasks that require parallel processing, a compatible `WebWorkerPool` is available. This enables developers to leverage multiple CPU cores to accelerate computationally intensive tasks, such as image registration and segmentation.

#### 3.4.6. *Automated package configuration:*

From an ITK-Wasm project, we generate a complete TypeScript/JavaScript package configuration. This includes:

- *TypeScript Bindings*: Generated for both Node.js and browser environments.
- *TypeScript Compilation Configuration*: Pre-configured for optimal performance and compatibility.
- *NPM Package Configuration*: Ready for Figure 7.

#### 3.4.7. *Documentation and demo app:*

To facilitate adoption and ease of use, we generate:

- *README*: A concise introduction to the project and its capabilities.
- *Docsify Documentation*: Detailed API documentation for the pipeline APIs.
- *Demo App*: An Figure 8 for sample data or user-provided data, allowing developers to experiment with API parameters and visualize results.

By providing a comprehensive set of tools and configurations, we empower developers to harness the full potential of ITK-Wasm in modern web applications, streamlining the development of scientific imaging and analysis tools.

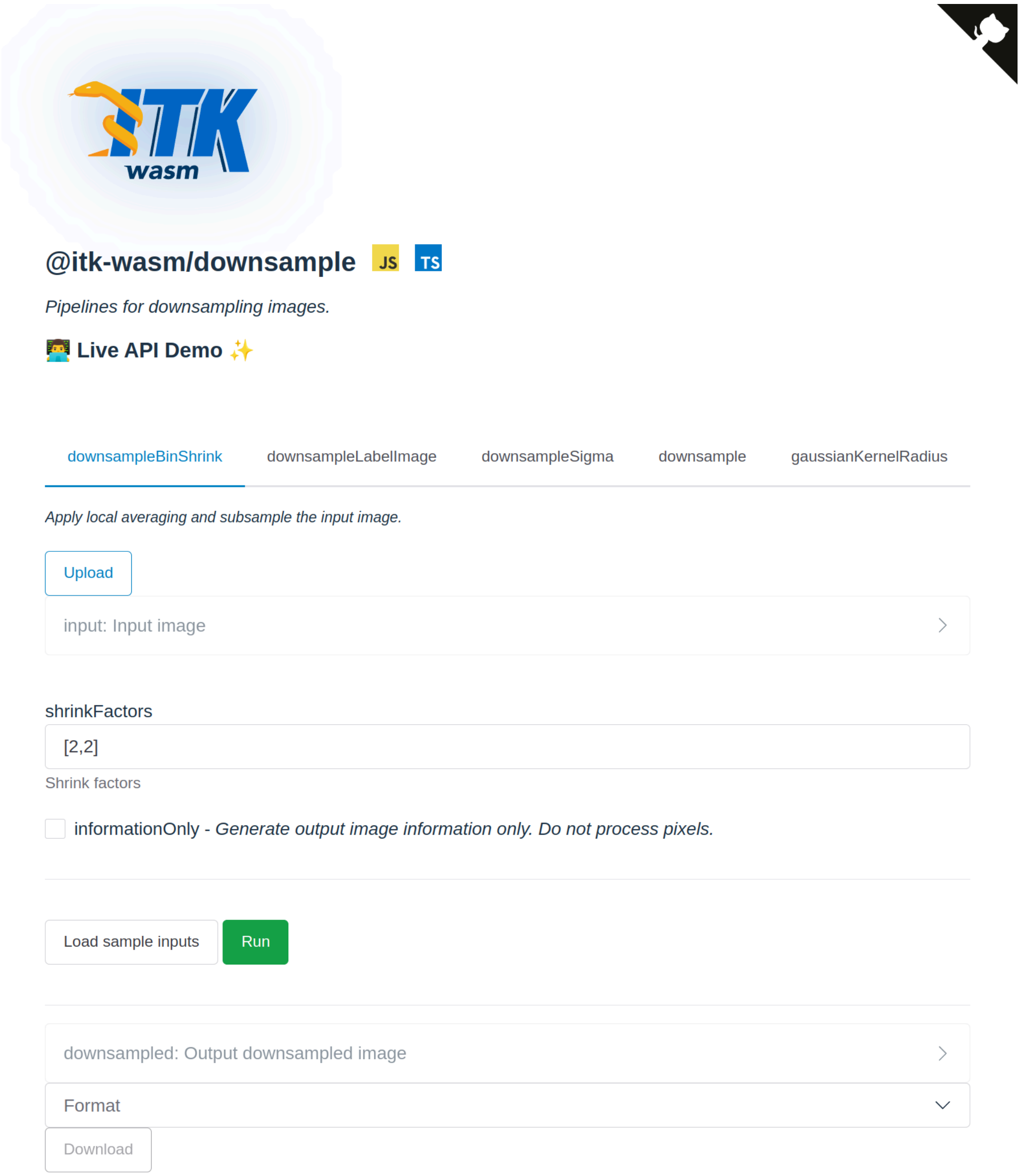
### *3.5. Seamless Integration with Python Ecosystem*

#### 3.5.1. *Browser-based Pyodide Packages:*

We leverage the Emscripten toolchain to generate bindings for browser-based Pyodide Python packages. This enables seamless integration of ITK-Wasm with Pyodide, allowing developers to utilize ITK's algorithms in web-based Python applications.

#### 3.5.2. *Cross-Platform Compatibility with WASI:*

For system applications, we provide a WASI-based Python package that ensures cross-platform compatibility across all major platforms and architectures. This broadens the reach of ITK-Wasm, enabling developers to deploy ITK-based applications on a wide range of systems.



@itk-wasm/downsample JS TS

*Pipelines for downsampling images.*

🧑‍💻 **Live API Demo** ✨

*Apply local averaging and subsample the input image.*

input: Input image

shrinkFactors

[2,2]

Shrink factors

□ informationOnly - *Generate output image information only. Do not process pixels.*

downsampled: Output downsampled image

Format

**Figure 8.** *Interactive live API demo application.*

### 3.5.3. *GPU Acceleration with cuCIM:*

To further enhance performance, we utilize the dispatch Python package's capabilities in conjunction with a cuCIM accelerator package. This enables GPU acceleration, to enable improvements the execution speed, especially in applications where bulk data resides on the GPU, which is often the case for AI-enabled workflows.

### 3.5.4. *API Documentation and Pythonic Interfaces:*

We generate API documentation for the simple, Pythonic interfaces, ensuring that developers can easily understand and utilize ITK-Wasm's functionality. The interfaces are designed to be intuitive and easy to use, streamlining the development process.

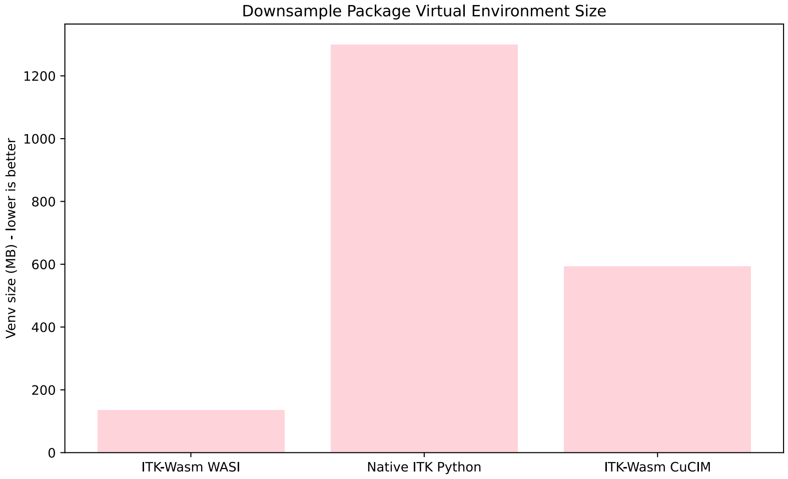


**Figure 9**. *Downsample example package virtual environment size. ITK-Wasm WASI packages are simple, small, self-contained, and have minimal dependencies. The associated accessibility and sustainability for software that depends on these packages is reflected by the virtual environment size. When compared to a native binary or CUDA-based implementations, the WASI implementation has significantly reduced size and complexity.*

### 3.5.5. *Efficient Serialization for Parallel Computing:*

The interface types' Python representation are built using Python data classes, comprising standard Python data types and NumPy arrays. This design enables trivial and efficient

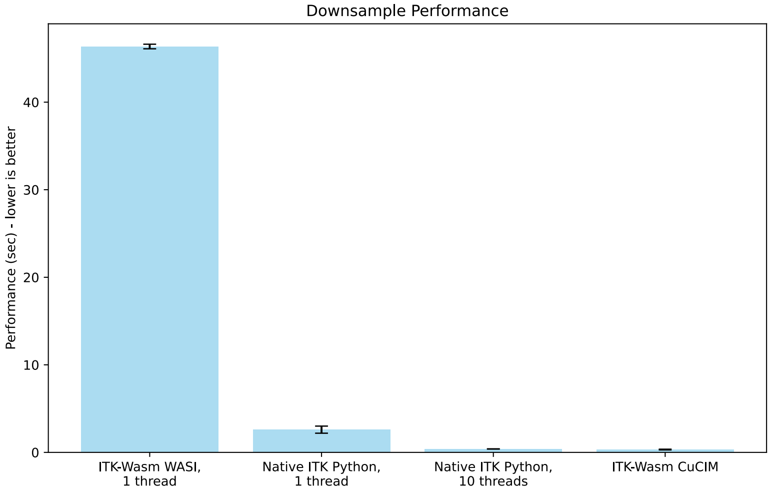
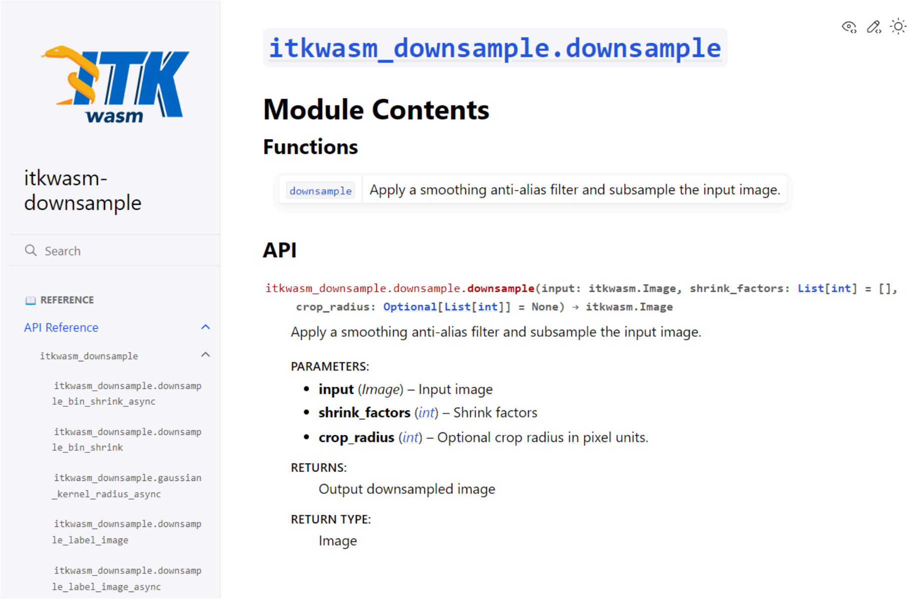


**Figure 10**. *Downsample example performance comparison between ITK-Wasm WASI, an equivalent ITK Python native binary implementation, and the ITK-Wasm CuCIM implementation. Executed on an Ryzen 9, 7940HS CPU, NVIDIA RTX 4070 Laptop GPU, Ubuntu 24.04 Linux system. Mean and standard deviation for ten iterations. While in this particular example the currently single-threaded WASI implementation is significantly slower than the native binary implementation, multi-threaded improvements with the native binaries hold promise for when this is enabled on the WASI binary (future work). NVIDIA CUDA-based ITK-Wasm CuCIM, applied without any other code changes when the `itkwasm-downsample-cucim` package is installed, demonstrates easy access to GPU acceleration when NVIDIA GPUs and CUDA software is available.*


itkwasm-downsample
Search
REFERENCE
API Reference
itkwasm_downsample
itkwasm_downsample.downsample_bin_shrink_async
itkwasm_downsample.downsample_bin_shrink
itkwasm_downsample.gaussian_kernel_radius_async
itkwasm_downsample.downsample_label_image
itkwasm_downsample.downsample_label_image_async

`itkwasm_downsample.downsample`

Module Contents

Functions

| | |
|---|---|
| `downsample` | Apply a smoothing anti-alias filter and subsample the input image. |

API

```
itkwasm_downsample.downsample.downsample(input: itkwasm.Image, shrink_factors: List[int] = [],
    crop_radius: Optional[List[int]] = None) → itkwasm.Image
```

Apply a smoothing anti-alias filter and subsample the input image.

PARAMETERS:

- **input** (*Image*) – Input image
- **shrink_factors** (*int*) – Shrink factors
- **crop_radius** (*int*) – Optional crop radius in pixel units.

RETURNS:

Output downsampled image

RETURN TYPE:

Image

**Figure 11**. *Generated API documentation describes the Pythonic interfaces.*

serialization, making it ideal for parallel computing with Dask. Developers can leverage this capability to scale their applications and tackle large-scale computing tasks.

### 3.5.6. *Broad Applicability in Scientific Computing:*

The utility of ITK-Wasm extends beyond web applications, as it can be seamlessly integrated into desktop applications like 3D Slicer [27]. This versatility demonstrates the broad applicability of ITK-Wasm in the scientific computing ecosystem, making it an invaluable tool for researchers and developers alike.

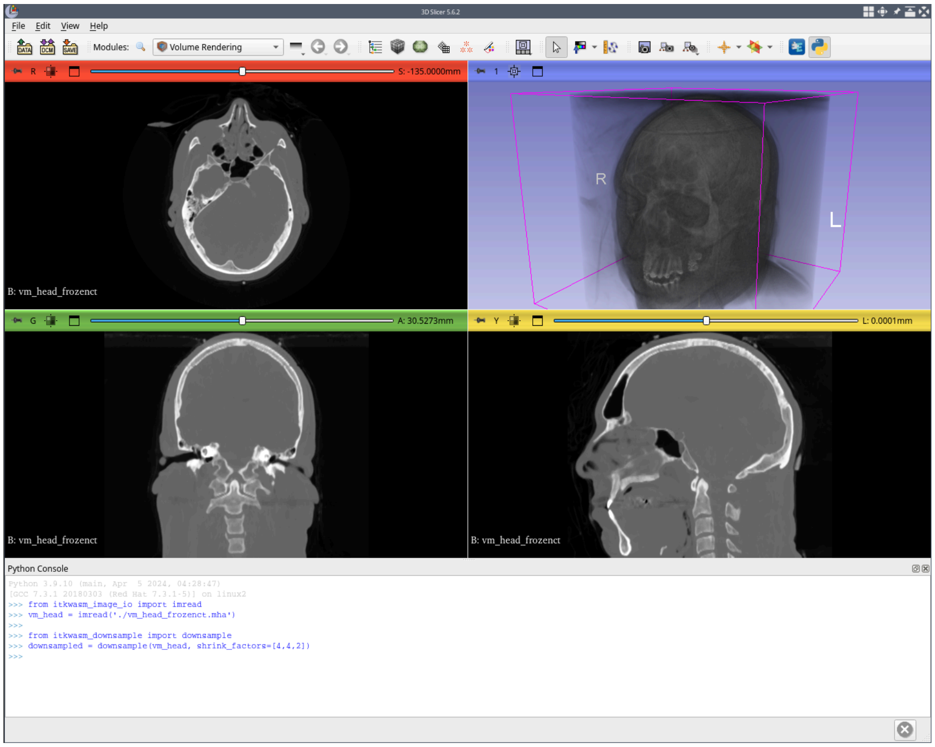
## 4. Discussion

WebAssembly was designed with interoperability in mind. Initially supporting languages like C and C++, the ecosystem has grown to include Rust, Go, Python, and more. This broad language support makes WebAssembly a versatile tool for developers across different domains.

ITK-Wasm's approach, which focuses on bringing wasm's capabilities to scientific software, excels in sustainability and composability thanks to small, self-contained, and idiomatic packages that are platform-agnostic and have minimal dependencies. This design enables outstanding computational reproducibility.

Future work will focus on enhanced integration of wasm community tools and standards:

1. **Multi-language Support**: Support for bindings and package generation in additional languages like Java, C#, and Rust, broaden wasm module applicability.
2. **WebAssembly Interface Types**: Standardizes the way Wasm modules interact with each other and with host environments, simplifying the integration process. We plan to bridge our interface types with the emerging Wasm Interface Type (WIT) definition.
3. **Component Model**: An emerging standard that aims to improve modularity and reuse of Wasm components, further enhancing interoperability. Further instrumentation with the Component Model standard will enable generation of composite



```
Python 3.9.10 (main, Apr  5 2024, 04:28:47)
[GCC 7.3.1 20180303 (Red Hat 7.3.1-5)] on linux2
>>> from itkwasm_image_io import imread
>>> vm_head = imread('./vm_head_frozenct.mha')
>>>
>>> from itkwasm_downsample import downsample
>>> downsampled = downsample(vm_head, shrink_factors=[4,4,2])
>>>
```

**Figure 12**. *The `itkwasm-downsample` Python package in a traditional native desktop application, 3D Slicer.*

processing pipeline wasm modules that could be built from wasm component modules written in multiple languages.

ITK-Wasm provides a robust framework for scientific computing that leverages WebAssembly's strengths. The framework bridges the gap between web-based and native applications, enabling high-performance, cross-platform scientific analysis. By integrating the principals of the WebAssembly Component Model, ITK-Wasm enhances interoperability and sustainability, allowing scientific Python to thrive in a multi-language ecosystem.

## 5. Conclusion

ITK-Wasm stands at the forefront of fostering interoperability, multi-language program support, sustainability, accessibility, and reproducibility in scientific computing. By integrating the WebAssembly Component Model, ITK-Wasm not only enhances scientific Python's capabilities but also sets a new standard for developing and distributing multi-language projects. The future of scientific computing is bright with ITK-Wasm's contributions to the field, providing a universal platform for spatial analysis and visualization.

## 6. Links:

- Documentation: https://wasm.itk.org/
- Source code: https://github.com/InsightSoftwareConsortium/ITK-Wasm

### Acknowledgments

The development of ITK-Wasm has been supported, in part, by the National Institute of Mental Health (NIMH) of the National Institutes of Health (NIH) under the BRAIN Initiative award number 1RF1MH126732.

## References

[1] A. Rossberg, Ed., “WebAssembly Core Specification,” Dec. 2019. [Online]. Available: https://www.w3.org/TR/wasm-core-1/

[2] A. Rossberg, Ed., “WebAssembly Core Specification,” Apr. 2022. [Online]. Available: https://www.w3.org/TR/wasm-core-2/

[3] A. Haas *et al.*, “Bringing the web up to speed with WebAssembly,” *SIGPLAN Not.*, vol. 52, no. 6, pp. 185–200, 2017, doi: 10.1145/3140587.3062363.

[4] A. Zakai, “Emscripten: an LLVM-to-JavaScript compiler,” in *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, in OOPSLA '11. Portland, Oregon, USA, 2011, pp. 301–312. doi: 10.1145/2048147.2048224.

[5] J.-m. Mirebeau, J. Fehrenbach, L. Risser, and S. Tobji, “Anisotropic Diffusion in ITK,” *The Insight Journal*, 2014, doi: 10.54294/en3833.

[6] M. McCormick, “Anisotropic Diffusion LBR.” [Online]. Available: https://insightsoftwareconsortium.github.io/ITKAnisotropicDiffusionLBR/

[7] A. Rossberg *et al.*, “Bringing the web up to speed with WebAssembly,” *Commun. ACM*, vol. 61, no. 12, pp. 107–115, 2018, doi: 10.1145/3282510.

[8] Dan Gohman *et al.*, “WebAssembly/WASI: v0.2.2.” [Online]. Available: https://zenodo.org/doi/10.5281/zenodo.4323446

[9] “The WebAssembly System Interface (WASI).” [Online]. Available: https://wasi.dev/

[10] M. Chadha, N. Krueger, J. John, A. Jindal, M. Gerndt, and S. Benedict, “Exploring the Use of WebAssembly in HPC,” in *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, in PPoPP '23. <conf-loc>, <city>Montreal</city>, <state>QC</state>, <country>Canada</country>, </conf-loc>, 2023, pp. 92–106. doi: 10.1145/3572848.3577436.

[11] Y. Zhang, M. Liu, H. Wang, Y. Ma, G. Huang, and X. Liu, “Research on WebAssembly Runtimes: A Survey,” 2024, [Online]. Available: http://arxiv.org/abs/2404.12621

[12] M. McCormick, “itk-wasm: high-performance spatial analysis in a web browser, Node.js, and reproducible execution across programming languages and hardware architectures..” [Online]. Available: https://zenodo.org/doi/10.5281/zenodo.3688880

[13] M. McCormick, X. Liu, J. Jomier, C. Marion, and L. Ibanez, “ITK: enabling reproducible research and open science,” *Front. Neuroinform.*, vol. 8, p. 13, 2014, doi: 10.3389/fninf.2014.00013.

[14] L. Ibanez *et al.*, “InsightSoftwareConsortium/ITK: ITK 5.4 Release Candidate 4: ALL THE DICOMs.” [Online]. Available: https://zenodo.org/doi/10.5281/zenodo.889843

[15] Henry Schreiner *et al.*, “CLIUtils/CLI11: Version 2.4.2: Build systems.” [Online]. Available: https://zenodo.org/doi/10.5281/zenodo.804964

[16] D. Developers, “dockcross: Cross compiling toolchains in Docker images.” [Online]. Available: https://github.com/dockcross/dockcross

[17] S. Moxon *et al.*, “LinkML.” [Online]. Available: https://zenodo.org/doi/10.5281/zenodo.5703670

[18] W. Lei, C. Fuster-Barceló, G. Reder, A. Muñoz-Barrutia, and W. Ouyang, “BioImage.IO Chatbot: A Community-Driven AI Assistant for Integrative Computational Bioimaging,” 2023, doi: 10.48550/ARXIV.2310.18351.

[19] W. Lei, C. Fuster-Barceló, G. Reder, A. Muñoz-Barrutia, and W. Ouyang, “BioImage.IO Chatbot: A Community-Driven AI Assistant for Integrative Computational Bioimaging.” [Online]. Available: https://zenodo.org/doi/10.5281/zenodo.10032227

[20] National Library of Medicine, “The Visible Human Project: Visible Human Male.” [Online]. Available: https://www.nlm.nih.gov/research/visible/visible_human.html

[21] J. Moore *et al.*, “OME-Zarr: a cloud-optimized bioimaging file format with international community support,” *Histochemistry and Cell Biology*, vol. 160, no. 3, pp. 223–251, 2023, doi: 10.1007/s00418-023-02209-1.

[22] J. Moore *et al.*, “OME-NGFF: a next-generation file format for expanding bioimaging data-access strategies,” *Nature Methods*, vol. 18, no. 12, pp. 1496–1498, 2021, doi: 10.1038/s41592-021-01326-w.

[23] Matt McCormick, Benedikt Best, and Tom Birdsong, “thewtex/ngff-zarr: ngff-zarr 0.8.7.” [Online]. Available: https://zenodo.org/doi/10.5281/zenodo.8092821

[24] H. J. Johnson, M. M. McCormick, L. Ibáñez, and Insight Software Consortium, *The ITK Software Guide: Introduction and development guidelines*. Kitware, 2015. [Online]. Available: https://play.google.com/store/books/details?id=JmzwrQEACAAJ

[25] H. J. Johnson, M. M. McCormick, L. Ibáñez, and Insight Software Consortium, *The ITK Software Guide: Design and functionality*. Kitware, 2015. [Online]. Available: https://play.google.com/store/books/details?id=SMwdrgEACAAJ

[26] M. J. Cardoso, M. Modat, T. Vercauteren, and S. Ourselin, “Scale Factor Point Spread Function Matching: Beyond Aliasing in Image Resampling,” in *Medical Image Computing and Computer-Assisted Intervention – MICCAI 2015*, Springer International Publishing, 2015, pp. 675–683. doi: 10.1007/978-3-319-24571-3_81.

[27] A. Fedorov *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, doi: 10.1016/j.mri.2012.05.001.

Open Access | https://doi.org/10.25080/KMXN4784

# How the Scientific Python ecosystem helps answer fundamental questions of the Universe

**Matthew Feickert**[1], **Nikolai Hartmann**[2], **Lukas Heinrich**[3], **Alexander Held**[1], **Vangelis Kourlitis**[3], **Nils Krumnack**[4], **Giordon Stark**[5], **Matthias Vigl**[3], and **Gordon Watts**[6]

[1]University of Wisconsin--Madison, [2]Ludwig Maximilians Universitat, [3]Technical University of Munich, [4]Iowa State University, [5]Santa Cruz Institute for Particle Physics, [6]University of Washington

SciPy 2024

July 8 - July 14, 2024

Proceedings of the 23$^{nd}$ Python in Science Conference

ISSN: 2575-9752

## Abstract

The ATLAS experiment at CERN explores vast amounts of physics data to answer the most fundamental questions of the Universe. The prevalence of Python in scientific computing motivated ATLAS to adopt it for its data analysis workflows while enhancing users' experience. This paper will describe to a broad audience how a large scientific collaboration leverages the power of the Scientific Python ecosystem to tackle domain-specific challenges and advance our understanding of the Cosmos. Through a simplified example of the renowned Higgs boson discovery, attendees will gain insights into the utilization of Python libraries to discriminate a signal in immersive noise, through tasks such as data cleaning, feature engineering, statistical interpretation and visualization at scale.

**Keywords** ATLAS, particle physics, Scikit-HEP

## 1. Introduction

The field of high energy physics (HEP) is devoted to the study of the fundamental forces of Nature and their interactions with matter. To study the structure of the Universe on the smallest scales requires the highest energy density environments possible — similar to those of the early Universe. These extreme energy density environments are created at the CERN international laboratory, in Geneva, Switzerland, using the Large Hadron Collider (LHC) to collide "bunches" of billions of protons at a center-of-mass energy of $\sqrt{s} = 13\,\mathrm{TeV}$. The resulting collisions are recorded with building-sized particle detectors positioned around the LHC's $27\,\mathrm{km}$ ring that are designed to measure subatomic particle properties. Given the rarity of the subatomic phenomena of interest, the rate of the beam crossings is a tremendous $40\,\mathrm{MHz}$ to maximize the number of high quality collisions that can be captured and read out by the detectors. Even with real-time onboard processing ("triggering") of the experiment detector readout to save only the most interesting collisions, detectors like the ATLAS experiment [1] still produce multiple petabytes of data per year. These data are then further filtered through selection criteria on the topology and kinematic quantities of the particle collision "events" recorded into specialized datasets for different kinds of physics analysis. The final datasets that physicists use in their physics analyses in ATLAS is still on the order of hundreds of terabytes, which poses challenges of compute scale and analyst time to efficiently use while maximizing physics value.

Traditionally, the ATLAS and the other LHC experiment have created experiment-specific custom C++ frameworks to handle all stages of the data processing pipeline, from the initial

**Published** Jul 10, 2024

**Correspondence to**
Matthew Feickert
matthew.feickert@cern.ch

construction of high-level physics objects from the raw data to the final statistical analyses. Motivated by the broad success of the Scientific Python ecosystem across many domains of science, and the rise of the Scikit-HEP ecosystem of Pythonic tooling for particle physics [2], [3] and community tools produced by the Institute for Research and Innovation in Software for High Energy Physics (IRIS-HEP) [4], [5], there has been a broad community-driven shift in HEP towards use of the Scientific Python ecosystem for analysis of physics data — a PyHEP ecosystem [6]. The use of dataframes and array programming for data analysis has enhanced the user experience while providing efficient computations without the need of coding optimized low-level routines. The ATLAS collaboration is further extending this ecosystem of tooling to include high-level custom Python bindings to the low level `C++` frameworks using `nanobind` [7]. Collectively, these tools are modernizing the methods which researchers are engaging data analysis at large scale and providing a novel end-to-end analysis ecosystem for the ATLAS collaboration.

## 2. Employing the Scientific Python ecosystem

The multiple stages of physics data processing and analysis map onto different parts of the Scientific Python ecosystem. This begins with the highly-structured but jagged nature of the event data in HEP. The data structure of each event consists of variable length lists of physics objects (e.g. electrons, collections of tracks from charged objects). To study the properties of the physics objects in a statistical manner, a fixed event analysis procedure is repeated over billions of events. This has traditionally motivated the use of “event loops” that implicitly construct event-level quantities of interest and leveraged the `C++` compiler to produce efficient iterative code. This precedent made it difficult to take advantage of array programming paradigms that are common in Scientific Python given NumPy [8] vector operations. The Scikit-HEP library Awkward Array [9] provides a path forward by providing NumPy-like idioms for nested, variable-sized (JSON-like) data and also brings analysts into an array programming paradigm [10].

With the ability to operate on HEP data structures in an array programming — or “columnar” — approach, the next step is to be able to read and write with the HEP domain specific ROOT [11] file format — which has given the particle physics community columnar data structures with efficient compression since 1997 [12]. This is accomplished with use of the `uproot` library [13], which allows for efficient transformation of ROOT data to NumPy or Awkward arrays. The data is then filtered through kinematic and physics signature motivated selections using Awkward manipulations and queries to create array collections that contain the passing events. Through intense detector characterization and calibration efforts, the ATLAS collaboration has developed robust methods and tooling to apply corrections to the data and evaluate systematic uncertainties. For instance, corrections to the signal collected by a specific calorimeter subsystem along with systematic uncertainties due to the imperfect knowledge of the subsystem. Given the custom nature of the detector and correction implementations, these corrections are implemented in custom `C++` libraries in the ATLAS software framework, Athena [14], [15]. To expose these `C++` libraries to the Pythonic tooling layer, custom Python bindings are written using `nanobind` for high efficiency, as seen in Figure 1.

To contend with the extreme data volume, efficient distributed computing is an essential requirement. Given the success of Dask [17] in the Scientific Python ecosystem, and its ability to be deployed across both traditional batch systems and cloud based infrastructure with Kubernetes, the Scikit-HEP ecosystem has built extensions to Dask that allow for native Dask collections of Awkward arrays [18] and computing multidimensional `boost-histogram` objects [19] with Dask collections [20]. Using Dask and these extensions, the data selection and systematic correction workflow is able to be horizontally scaled out across ATLAS collaboration compute resources to provide the data throughput necessary to make analysis

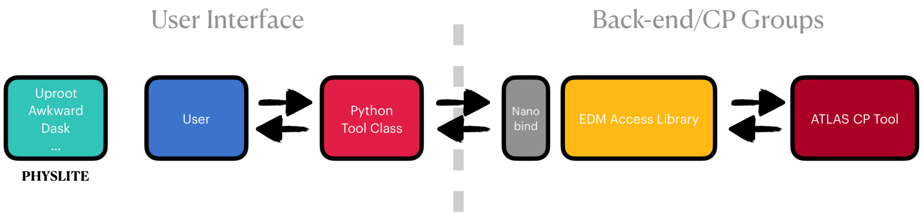


**Figure 1**. *The data access abstract interface from the high level user facing Python API to the ATLAS Event Data Model (EDM) access library that exposes the shared ATLAS combined performance (CP) tools for reconstruction, identification, and measurement of physics objects. [16]*

feasible. This is often achieved through use of the high level `coffea` columnar analysis framework [21] which was designed to integrate with Dask and these HEP specific Dask extensions.

The resulting data objects that are returned to analysts are histograms of physics quantity distributions — such as the reconstructed invariant-mass of a collection of particles or particle momentum. Using the `hist` library [22] for higher level data exploration and manipulation, physicists are then able to efficiently further manipulate the data distributions using tooling from the broader Scientific Python ecosystem and create domain-centric visualizations using the `mplhep` [23] extension of Matplotlib [24]. From these high level data representations of the relevant physics, physicists are then able to serialize the distributions and use them for the final stages of data analysis and statistical modeling and inference.

## 3. Uncovering the Higgs boson

The most famous and revolutionary discovery in particle physics this century is the discovery of the Higgs boson — the particle corresponding to the quantum field that gives mass to fundamental particles through the Brout-Englert-Higgs mechanism — by the ATLAS and CMS experimental collaborations in 2012. [25], [26] This discovery work was done using large amounts of customized `C++` software, but in the following decade the state of the PyHEP community has advanced enough that the workflow can now be done using community Python tooling. To provide an overview of the Pythonic tooling and functionality, a high level summary of a simplified analysis workflow [27] of a Higgs "decay" to two intermediate $Z$ bosons that decay to charged leptons $(\ell)$ (i.e. electrons $(e)$ and muons $(\mu)$), $H \to ZZ^* \to 4\ell$, on ATLAS open data [28] is summarized in this section.

### 3.1. *Loading data*

Given the size of the data, the files used in a real analysis will usually be cached at a national level "analysis facility" where the analysis code will run. Using `coffea`, `uproot`, and Dask, these files can then be efficiently read and the tree structure of the data can populate Awkward arrays.

### 3.2. *Cleaning and selecting data*

Once the data is in Awkward arrays, additional selections need to be applied before it can be analyzed. Only physics objects of adequate quality are kept for further analysis and those should reconstruct the topology of interest. In this particular case, due to the decay of the Higgs boson to two leptons, the data selected contain four charged leptons grouped in two opposite flavor lepton pairs (so that the total charge is zero, as the Higgs and the $Z$-bosons are electrically neutral). Additionally, in order to compare various kinds of simulated data, the events need to be normalized/weighted given their relative appearance in reality and the amount of actual data collected by the experiment.

```
from coffea.nanoevents import NanoEventsFactory, PHYSLITESchema


def get_uris_from_cache(): ...


def filter_name(name):
    return name in (
        "AnalysisElectronsAuxDyn.pt",
        "AnalysisElectronsAuxDyn.eta",
        "AnalysisElectronsAuxDyn.phi",
        "AnalysisElectronsAuxDyn.m",
        "AnalysisElectronsAuxDyn.charge",
        ...,
    )


file_uris = get_uris_from_cache(...)

# uproot used internally to read files into Awkward arrays
events_mc = NanoEventsFactory.from_root(
    file_uris,
    schemaclass=PHYSLITESchema,
    uproot_options=dict(filter_name=filter_name),
    permit_dask=True,
).events()
```

**Program 1.** *Using `coffea`, tree structured ROOT files are read with `uproot` from an efficient file cache, and the relevant branches for physics are filtered out into Awkward arrays. The operation is scaled out on a Dask cluster for read performance.*

These selection and weighting can then be implemented in an analysis specific `coffea` processor, and then the processor can be executed used a Dask executor to horizontally scale out the analysis selection across the available compute.

### *3.3. Feature engineering: The invariant mass*

In order to discriminate the events of interest, i.e. candidates of the Higgs boson decay, from the vast background which has the same experimental signature, a discriminating feature is constructed. The example shown uses a simple, physics-inspired discriminant the “invariant mass” but the methods used can use complex feature engineering that involve machine learning methods to calculate more efficient discriminants. The invariant mass is the mass of a system that remains constant regardless of the system’s motion or the reference frame in which it is measured. Invariant mass is derived from the energy and momentum of a system of particles and is a fundamental property of the system:

$$m = \frac{\sqrt{E^2 - p(c)^2}}{c^2} \tag{1}$$

where $E$ and $p$ is the total energy and momentum of the particles, respectively.

By detecting and measuring the energies and momenta of the detected particles at the experiment, we can reconstruct the invariant mass of the decay system. Particle systems originating from the decay of the Higgs boson will have a characteristic value of the invariant mass, which after the discovery in 2012 we know it is about $125\ \mathrm{GeV}/c^2$. This is the quantity that will allow us to discriminate from particle systems that originate from background processes.

### *3.4. Measurement uncertainties*

One of the most expensive operations that happens during the event selections is the computation of systematic variations of the events to accommodate for imperfect knowledge of the detector systems. This in practice requires applying complex, experiment specific cor-

```
import awkward as ak
import hist
import vector
from coffea import processor
from distributed import Client


def get_xsec_weight(sample, infofile):
        """Returns normalization weight for a given
sample."""
    lumi = 10_000  # pb^-1
    xsec_map = infofile.infos[sample]  # dictionary with
event weighting information
    xsec_weight = (lumi * xsec_map["xsec"]) /
(xsec_map["sumw"] * xsec_map["red_eff"])
    return xsec_weight


def lepton_filter(lep_charge, lep_type):
    """Filters leptons: sum of charges is required to
be 0, and sum of lepton types 44/48/52.
    Electrons have type 11, muons have 13, so this
means 4e/4mu/2e2mu.
    """
    sum_lep_charge = ak.sum(lep_charge, axis=1)
    sum_lep_type = ak.sum(lep_type, axis=1)
    good_lep_type = ak.any(
        [sum_lep_type == 44, sum_lep_type == 48,
sum_lep_type == 52], axis=0
    )
    return ak.all([sum_lep_charge == 0, good_lep_type],
axis=0)


class HZZAnalysis(processor.ProcessorABC):
    """The coffea processor used in this analysis."""

    def __init__(self):
        pass

    def process(self, events):
        # The process function performs columnar operations
on the events
        # passed to it and applies all the corrections
and selections to
        # either the simulation or the data (e.g.
get_xsec_weight and
        # lepton_filter). All the event level data
selection occurs here
        # and returns accumulators with the selections.

        vector.register_awkward()
        # type of dataset being processed, provided via
metadata (comes originally from fileset)
        dataset_category =
events.metadata["dataset_name"]

        # apply a cut to events, based on lepton charge
and lepton type
        events = events[lepton_filter(events.lep_charge,
events.lep_typeid)]

        # construct lepton four-vectors
        leptons = ak.zip(
            {
                "pt": events.lep_pt,
                "eta": events.lep_eta,
                "phi": events.lep_phi,
                "energy": events.lep_energy,
            },
            with_name="Momentum4D",
        )

        # calculate the 4-lepton invariant mass for each
remaining event
        # this could also be an expensive calculation
using external tools
        mllll = (
            leptons[:, 0] + leptons[:, 1] + leptons[:,
2] + leptons[:, 3]
        ).mass / 1000

        # create histogram holding outputs, for data
just binned in m4l
        mllllhist_data = hist.Hist.new.Reg(
            num_bins,
            bin_edge_low,
            bin_edge_high,
            name="mllll",
            label="$\mathrm{m_{4l}}$ [GeV]",
        ).Weight()  # using weighted storage here for
plotting later, but not needed

        # three histogram axes for MC: m4l, category,
and variation (nominal and
        # systematic variations)
        mllllhist_MC = (
            hist.Hist.new.Reg(
                num_bins,
                bin_edge_low,
                bin_edge_high,
                name="mllll",
                label="$\mathrm{m_{4l}}$ [GeV]",
            )
            .StrCat([k for k in fileset.keys() if k !=
"Data"], name="dataset")
            .StrCat(
                ["nominal", "scaleFactorUP",
"scaleFactorDOWN", "m4lUP", "m4lDOWN"],
                name="variation",
            )
            .Weight()
        )

        # ...
        # fill histograms based on dataset_category
        # ...

        return {"data": mllllhist_data, "MC":
mllllhist_MC}

    def postprocess(self, accumulator):
        pass
```

**Program 2.** *A* `coffea` *processor designed to make physics motivated event selections to create accumulators of the 4-lepton invariant mass.*

```python
import awkward as ak

from atlascp import EgammaTools  # ATLAS CP tool Python nanobind bindings


def get_corrected_mass(energyCorrectionTool, electrons, sys=None):
    electron_vectors = ak.zip(
        {
            "pt": energyCorrectionTool(electrons, sys=sys).newPt,
            "eta": electrons.eta,
            "phi": electrons.phi,
            "mass": electrons.m,
        },
        with_name="Momentum4D",
    )
    return (electron_vectors[:, 0] + electron_vectors[:, 1]).mass / 1000  # GeV


energy_correction_tool = EgammaTools.EgammaCalibrationAndSmearingTool()
# ...
# configure and initialize correction algorithm
# ...
energy_correction_tool.initialize()

corrected_m_Res_UP = get_corrected_mass(
    energy_correction_tool, electrons, "Res_up"
).compute()
```

**Program 3**. *Simplified example of what the Python API for a systematic correction tool with a columnar implementation looks like.*

rections to each event, using algorithms implemented in C++. Historically these tools were implemented for an event loop processing paradigm, but with recent tooling additions, as shown in Figure 1, efficient on-the-fly systematic corrections can be computed for array programming paradigms.

The following provides an example of high level Python APIs that provide handles to these tools to use in the workflows described so far. These tools are efficient enough to be able to apply multiple systematic variations in analysis workflows, as seen in Figure 2.

### 3.5. *The "discovery" plot*

After running the `coffea` processors, the resulting data from the selections is accumulated into `boost-histogram` objects, as seen visualized in Figure 3.

These histograms are then serialized into files with `uproot` and used by the statistical modeling and inference libraries `pyhf` [29], [30] and `cabinetry` [31] to build binned statistical models and efficiently fit the models to the observed data using vectorized computations and the optimization library `iminuit` [32] for full uncertainties on all model parameters. The resulting best-fit model parameters — such as the scale factor on the signal component of the model corresponding to the normalization on the Higgs contributions — are visualized in Figure 4, where good agreement between the model predictions and the data is observed. The signal component, clearly visible above the additional "background" components of the model, are Higgs boson events, with an observed count in agreement with theoretical expectations.

## 4. Conclusions

When the Higgs boson was discovered in 2012, the idea of being able to perform real Pythonic data analysis in HEP, let alone *efficient* analysis, was viewed as unfeasible. Though investment in the broader Scientific Python ecosystem, and development of the domain specific pieces in the Scikit-HEP organization the field of particle physics successfully cre-

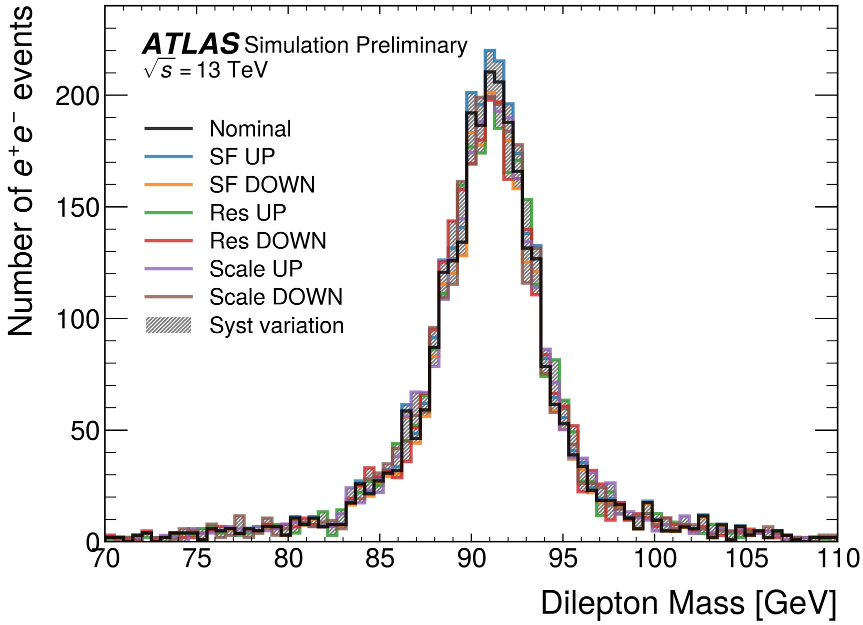


**Figure 2.** *Example of the reconstructed dilepton invariant mass distribution in simulation with the electron reconstruction and identification efficiency scale factor (SF) and corrections to the energy resolution (res) energy scale (scale) computed on-the-fly using the `nanobind` Python bindings to the ATLAS `C++` correction tools. The total variation in the systematic corrections is plotted as a hashed band. [16]*

ated a PyHEP ecosystem of robust tooling. Further investment by the ATLAS collaboration has resulted in new performant tooling for complex systematic corrections that will allow for more full and complex operations to be performed entirely within a Python workflow, helping to further reduce the time to insight for physics analysts.

```
import hist
import mplhep

mplhep.histplot(
    all_histograms["data"], histtype="errorbar", color="black", label="Data"
)
hist.Hist.plot1d(
    all_histograms["MC"][:, :, "nominal"],
    stack=True,
    histtype="fill",
    color=["purple", "red", "lightblue"],
)
```

**Program 4.** *Using `mplhep`, `hist`, and `matplotlib` the post-processed histograms of the simulation and the data are visualized in advance of any statistical inference of best-fit model parameters.*

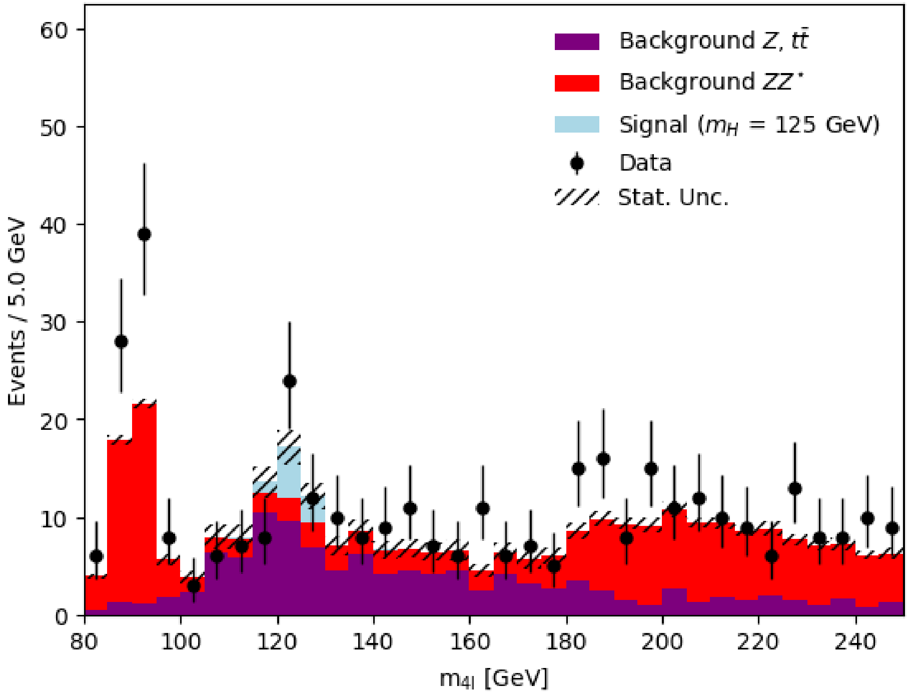


**Figure 3.** *Using `mplhep`, `hist`, and `matplotlib` the post-processed histograms of the simulation and the data are visualized in advance of any statistical inference of best-fit model parameters.*

```python
import cabinetry
import numpy as np

config = cabinetry.configuration.load("config.yml")

cabinetry.templates.collect(config)
cabinetry.templates.postprocess(config)  # optional post-processing (e.g. smoothing)
workspace = cabinetry.workspace.build(config)

model, data = cabinetry.model_utils.model_and_data(workspace)
fit_results = cabinetry.fit.fit(model, data)

# create post-fit model prediction
postfit_model = cabinetry.model_utils.prediction(model, fit_results=fit_results)

# binning to use in plot
plot_config = {
    "Regions": [
        {
            "Name": "Signal_region",
            "Binning": list(np.linspace(bin_edge_low, bin_edge_high, num_bins + 1)),
        }
    ]
}

figure_dict = cabinetry.visualize.data_mc(
    postfit_model, data, config=plot_config, save_figure=False
)

# modify x-axis label
fig = figure_dict[0]["figure"]
fig.axes[1].set_xlabel("m4l [GeV]")
```

**Program 5.** *Using `cabinetry`, `pyhf`, and `matplotlib` the data and the post-fit model prediction are visualized.*

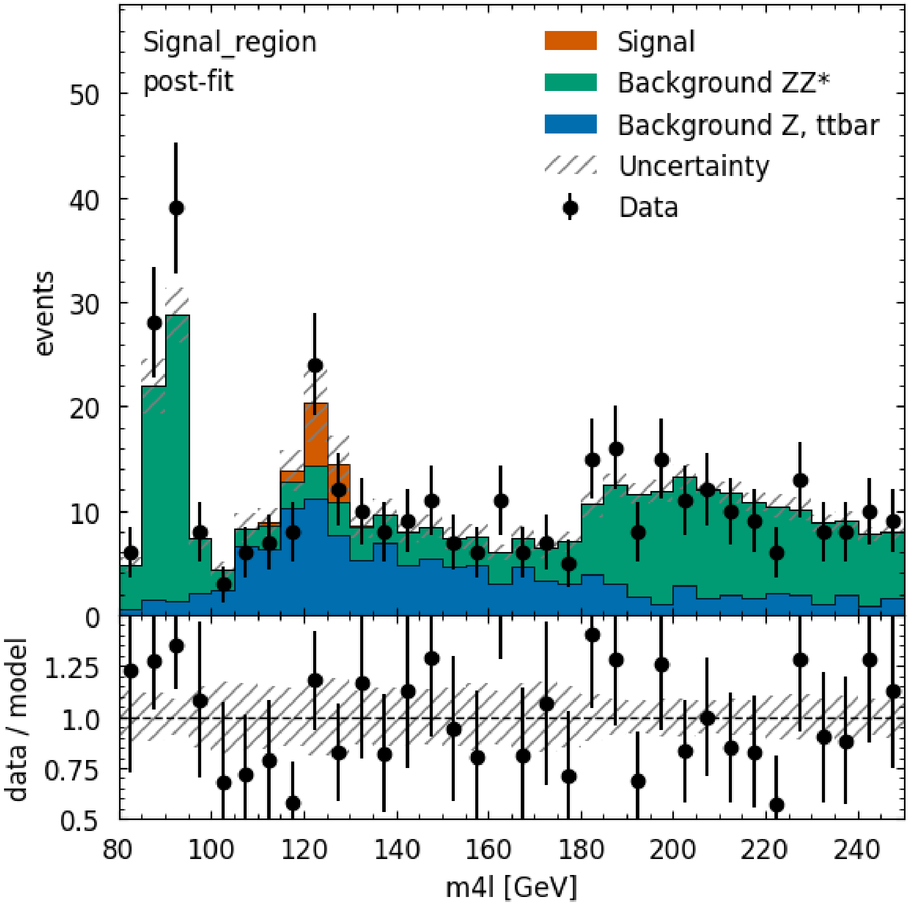


**Figure 4.** *Using* `cabinetry`*,* `pyhf`*, and* `matplotlib` *the data and the post-fit model prediction are visualized.*

## References

[1] ATLAS Collaboration, "The ATLAS Experiment at the CERN Large Hadron Collider," *JINST*, vol. 3, p. S8003, 2008, doi: 10.1088/1748-0221/3/08/S08003.

[2] E. Rodrigues and others, "The Scikit HEP Project – overview and prospects," *EPJ Web Conf.*, vol. 245, p. 6028, 2020, doi: 10.1051/epjconf/202024506028.

[3] Henry Schreiner, Jim Pivarski, and Eduardo Rodrigues, "Awkward Packaging: building Scikit-HEP," in *Proceedings of the 21st Python in Science Conference*, Meghann Agarwal, Chris Calloway, Dillon Niederhut, and David Shupe, Eds., 2022, pp. 115–120. doi: 10.25080/majora-212e5952-012.

[4] P. Elmer, M. Neubauer, and M. D. Sokoloff, "Strategic Plan for a Scientific Software Innovation Institute (S2I2) for High Energy Physics," 2017.

[5] J. Albrecht and others, "A Roadmap for HEP Software and Computing R&D for the 2020s," *Comput. Softw. Big Sci.*, vol. 3, no. 1, p. 7, 2019, doi: 10.1007/s41781-018-0018-8.

[6] HEP Software Foundation, "Python in HEP (PyHEP) HSF Working Group." [Online]. Available: https://hepsoftwarefoundation.org/workinggroups/pyhep.html

[7] W. Jakob, "nanobind: tiny and efficient C++/Python bindings." [Online]. Available: https://github.com/wjakob/nanobind

[8] C. R. Harris *et al.*, "Array programming with NumPy," *Nature*, vol. 585, no. 7825, pp. 357–362, 2020, doi: https://doi.org/10.1038/s41586-020-2649-2.

[9] J. Pivarski *et al.*, "Awkward Array." 2018. doi: 10.5281/zenodo.4341376.

[10] N. Hartmann, J. Elmsheuser, and G. Duckeck, "Columnar data analysis with ATLAS analysis formats," *EPJ Web Conf.*, vol. 251, p. 3001, 2021, doi: 10.1051/epjconf/202125103001.

[11] R. Brun and F. Rademakers, "ROOT: An object oriented data analysis framework," *Nucl. Instrum. Meth. A*, vol. 389, pp. 81–86, 1997, doi: 10.1016/S0168-9002(97)00048-X.

[12] J. Pivarski, P. Elmer, and D. Lange, "Awkward Arrays in Python, C++, and Numba," *EPJ Web Conf.*, vol. 245, p. 5023, 2020, doi: 10.1051/epjconf/202024505023.

[13] J. Pivarski *et al.*, "Uproot." 2017. doi: 10.5281/zenodo.4340632.

[14] ATLAS Collaboration, *The ATLAS Collaboration Software and Firmware*. ATL-SOFT-PUB-2021-001, 2021. [Online]. Available: https://cds.cern.ch/record/2767187

[15] ATLAS Collaboration, "Athena." 2021. doi: 10.5281/zenodo.4772550.

[16] V. Kourlitis *et al.*, "Using Legacy ATLAS C++ Calibration Tools in Modern Columnar Analysis Environments - Poster for ACAT 2024," Geneva, 2024. [Online]. Available: https://indico.cern.ch/event/1330797/contributions/5796636/

[17] Dask Development Team, "Dask: Library for dynamic task scheduling," 2016. [Online]. Available: https://dask.pydata.org/

[18] dask-awkward Development Team, "dask-awkward." [Online]. Available: https://github.com/dask-contrib/dask-awkward

[19] H. Schreiner *et al.*, "boost-histogram." 2018. doi: 10.5281/zenodo.3492034.

[20] dask-histogram Development Team, "dask-histogram." [Online]. Available: https://github.com/dask-contrib/dask-histogram

[21] L. Gray *et al.*, "coffea." [Online]. Available: https://github.com/CoffeaTeam/coffea

[22] H. Schreiner, S. Liu, and A. Goel, "hist." [Online]. Available: https://github.com/scikit-hep/hist

[23] A. Novak, H. Schreiner, and M. Feickert, "mplhep." [Online]. Available: https://github.com/scikit-hep/mplhep

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

[25] ATLAS Collaboration, "Observation of a new particle in the search for the Standard Model Higgs boson with the ATLAS detector at the LHC," *Phys. Lett. B*, vol. 716, p. 1, 2012, doi: 10.1016/j.physletb.2012.08.020.

[26] CMS Collaboration, "Observation of a new boson at a mass of 125 GeV with the CMS experiment at the LHC," *Phys. Lett. B*, vol. 716, p. 30, 2012, doi: 10.1016/j.physletb.2012.08.021.

[27] A. Held *et al.*, "IRIS-HEP Analysis Grand Challenge." [Online]. Available: https://doi.org/10.5281/zenodo.7274936

[28] ATLAS Collaboration, "ATLAS 13 TeV samples collection at least four leptons (electron or muon), for 2020 Open Data release." CERN Open Data Portal, 2020. doi: 10.7483/OPENDATA.ATLAS.2Y1T.TLGL.

[29] L. Heinrich, M. Feickert, and G. Stark, "pyhf: v0.7.6." [Online]. Available: https://doi.org/10.5281/zenodo.1169739

[30] L. Heinrich, M. Feickert, G. Stark, and K. Cranmer, "pyhf: pure-Python implementation of HistFactory statistical models," *Journal of Open Source Software*, vol. 6, no. 58, p. 2823, 2021, doi: 10.21105/joss.02823.

[31] A. Held and M. Feickert, "cabinetry." [Online]. Available: https://doi.org/10.5281/10.5281/zenodo.4742752

[32] H. Dembinski *et al.*, “iminuit.” [Online]. Available: https://github.com/scikit-hep/iminuit

Open Access | https://doi.org/10.25080/GVHT1072

# Supporting Greater Interactivity in the IPython Visualization Ecosystem

**Nathan Martindale**[1], **Jacob Smith**[1], and **Lisa Linville**[2]

[1]Oak Ridge National Laboratory, [2]Sandia National Laboratories

**SciPy 2024**

*July 8 - July 14, 2024*

Proceedings of the 23nd
Python in Science Conference
ISSN: 2575-9752

## Abstract

Interactive visualizations are invaluable tools for building intuition and supporting rapid exploration of datasets and models. Numerous libraries in Python support interactivity, and workflows that combine Jupyter and IPyWidgets in particular make it straightforward to build data analysis tools on the fly. However, the field is missing the ability to arbitrarily overlay widgets and plots on top of others to support more flexible details-on-demand techniques. This work discusses some limitations of the base IPyWidgets library, explains the benefits of IPyVuetify and how it addresses these limitations, and finally presents a new open-source solution that builds on IPyVuetify to provide easily integrated widget overlays in Jupyter.

**Keywords** interactive visualization, UX, python, javascript, web development, custom visualizations

## 1. Introduction

As the tech industry and computation-based research continues to gravitate toward using more and more data to solve problems, the capability to interact with and understand that underlying data becomes increasingly important. Development of effective visualization tools and frameworks to address this need has turned into an ecosystem where components from different sources all work in unison and developers can piece together what they need for their specific goals. For many reasons, including the ease of organization and allowing documentation, development, and analysis to all exist in the same place, Jupyter notebooks [1] have become a common platform for data science workflows, and by extension have become the common ground where these different tools can coexist. An important piece to this platform is the ability to interact with and visualize data, which is crucial in many data science/analysis workflows. Interactive visualization allows users to manipulate and explore data, which enhances understanding, improves engagement, and often leads to a quicker intuitive grasp of the data.

IPyWidgets [2], a framework for bridging between the IPython kernel and HTML widgets within Jupyter, has been particularly influential for enabling interactive visualization. Many libraries such as Panel [3] and Solara [4] integrate closely with IPyWidgets and thus contribute to the greater visualization ecosystem within Python. Although a great deal of progress has been made in this community, a missing element is a more generalized and flexible approach to aspects of the Section 1.1 paradigm. To support this approach, the authors have developed a library that provides wrapper components that can render any widget as an overlay on top of other widgets. These components simplify designing layouts and visualizations that would be difficult to implement from scratch.

In this work, we first discuss IPyWidgets and what makes it such an important piece of the overall system, as well as some of the limitations encountered when building only with the base IPyWidgets framework. We then briefly cover some of the available libraries that build on top of IPyWidgets, highlighting IPyVuetify, which addresses most of the limitations of

**Published** Jul 10, 2024

**Correspondence to**
Nathan Martindale
martindalena@ornl.gov

base IPyWidgets. Finally, we discuss a library we have open sourced called IPyOverlay and an example use case that motivated its design.

### *1.1. Details on Demand*

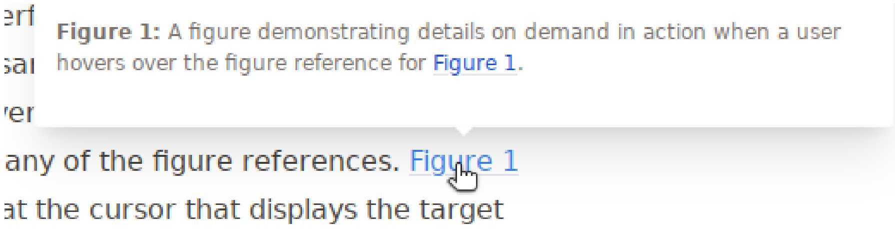
Details on demand is one piece of the "visual information-seeking mantra" of Schneiderman (1996) describing an overall design approach to building interactive tooling. The general flow of interaction proposed by this mantra is "overview first, zoom and filter, then details-on-demand" [5]. Overview ideally starts with broader aggregate visualizations: something to gain a sense of the general scope, complexity, or shape of the data. From there a user can identify areas of interest to hone in on with whatever zoom and filter tools are applicable for their data's modality. Finally, details on demand means that the user can see or interact with more in-depth and specific information for pieces of their data, often in the form of a pop-up window or inset graph. This selective detail capability provides a flexible user experience that keeps the initial interface uncluttered without sacrificing the ability to organically explore the data.

A simple and widespread example of details on demand are tooltips. Descriptions of components or data point values in visualizations would often create an unusably noisy interface if they were always visible. Instead, a more concise view is maintained while providing the necessary information in a spatially relevant view when the user specifically requests it. Readers viewing the web version of these proceedings can explore a simple example of details on demand live by hovering over any of the figure references. Figure 1 shows how hovering over a reference (the demand) opens a window at the cursor that displays the target of the reference (the details).

## 2. IPyWidgets

IPyWidgets is a library for implementing user interactivity inside of a Jupyter Notebook. A key challenge this framework has to overcome is the separation between the Python kernel and the browser-based Jupyter frontend, which operates through HTML, CSS, and JavaScript as a web application. Web application development is often a challenging software engineering problem in its own right, and many Python developers and scientists may not have either the experience or the desire to write custom frontend code in JavaScript. IPyWidgets addresses this problem by supporting mechanisms for automatic state synchronization between a Python model and a JavaScript model, expanded on below, and providing a set of predefined components that a developer can initialize and use in their Python code. While IPyWidgets has limitations, it is a valuable tool for the scientific Python community by allowing the combination of two language ecosystems with many visualization libraries and capabilities.

To highlight the simplicity of IPyWidgets, some minimal examples are provided here. In the simplest UI cases, IPyWidgets allows wrapping a function call that produces a visual output based on some set of input parameters with `ipywidgets.interactive()`, which automatically



erf
sa
er
any of the figure references. Figure 1
at the cursor that displays the target

**Figure 1:** A figure demonstrating details on demand in action when a user hovers over the figure reference for Figure 1.

**Figure 1.** *A figure demonstrating details on demand in action when a user hovers over the figure reference for Figure 1.*

```
import matplotlib.pyplot as plt
import numpy as np
import ipywidgets as ipw

def graph(coefficient):
    x = np.arange(0, 10, .1)
    y = np.sin(coefficient * x)
    plt.plot(x, y)

ipw.interactive(graph, coefficient=(0.1, 10.1))
```

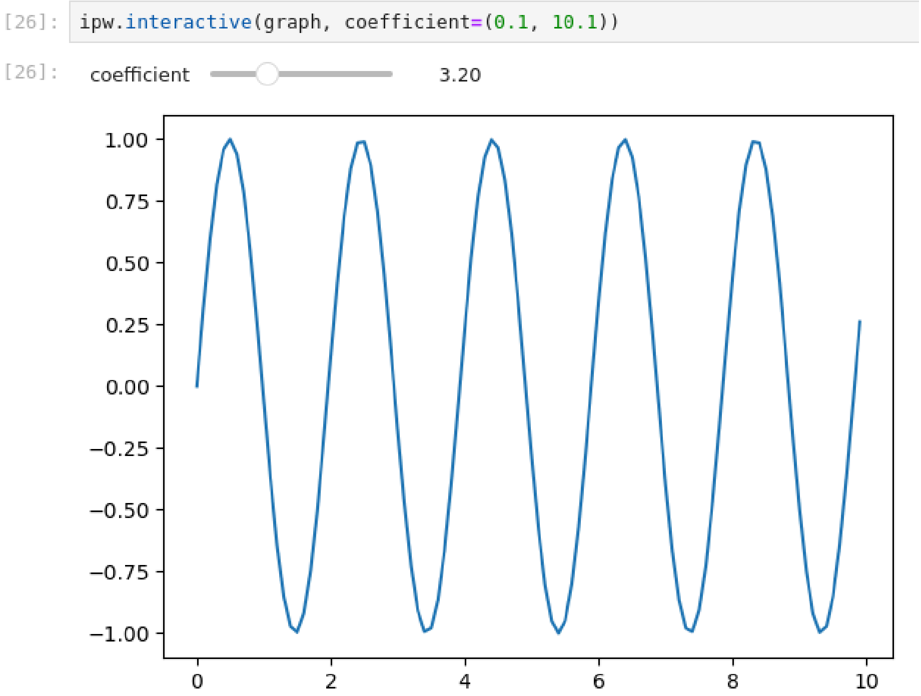
**Program 1.** *Minimal IPyWidget usage for a plot of a sine wave based on one parameter/coefficient. The* `interactive()` *call turns the passed tuple for* `coefficient` *into a slider widget with the range set to this tuple. See Figure 2 for output.*

constructs an appropriate input widget for each parameter. In this case, the developer need not manually initialize any Python widgets, as seen in the below code sample Program 1 and Figure 2.

More complex interfaces often require displaying multiple visualizations and outputs and may have controls that must interact. Thus, in practice a more common setup might follow a pattern shown in Program 2, where several user input and output widgets are defined and placed into a layout widget. UI code is generally event driven, so functionality is added by attaching event handlers to input widgets with `.observe()`, which will then call the associated event functions when the user interacts with the input widget. IPyWidget's generic `Output` widget supports IPython's set of display functions, and anything that can be rendered as the output to a Jupyter cell can be placed into a specified output widget.

### 2.1. State Synchronization

One of the primary challenges that IPyWidgets helps solve is the problem of state synchronization, where updates to variables made on either side of the Python-JavaScript link



```
[26]: ipw.interactive(graph, coefficient=(0.1, 10.1))
```

**Figure 2.** *A simple IPyWidgets example using* `interactive()`*. Every time the slider is moved, the* `graph(coefficient)` *function is called with the updated value and regenerates/displays the new plot.*

```
import ipywidgets as ipw

# initialize the widgets
graph1_out = ipw.Output()
graph2_out = ipw.Output()

graph1_param = ipw.FloatSlider(min=0, max=10, value=1)  # a parameter only influencing graph 1
graph2_param = ipw.IntSlider(min=0, max=10, value=1)  # a parameter only influencing graph 2
both_graphs_param = ipw.IntText(value=1)  # a parameter that affects both graphs

# create the dashboard layout, combining all the widgets
dashboard = ipw.VBox([
    ipw.HBox([graph1_out, graph2_out]),
    ipw.HBox([graph1_param, both_graphs_param, graph2_param]),
])

# event handler functions
def on_graph1_param_change(change):
    with graph1_out:
        print(f"Updating graph 1 based on: graph1_param={change['new']}, {both_graphs_param.value=}")

def on_graph2_param_change(change):
    with graph2_out:
        print(f"Updating graph 2 based on: graph2_param={change['new']}, {both_graphs_param.value=}")

def on_both_graphs_param_change(change):
    with graph1_out:
        print(f"Updating graph 1 based on: {graph1_param.value=}, both_graphs_param={change['new']}")
    with graph2_out:
        print(f"Updating graph 2 based on: {graph2_param.value=}, both_graphs_param={change['new']}")

# attach event handlers
graph1_param.observe(on_graph1_param_change, names=["value"])
graph2_param.observe(on_graph2_param_change, names=["value"])
both_graphs_param.observe(on_both_graphs_param_change, names=["value"])

# render the dashboard
dashboard
```

**Program 2.** *An example of a more typical structure for a dashboard with several inputs and outputs that might have more complicated interdependencies than can be handled with* `interactive()`*.*

are monitored and communicated across when they occur. Based on this communication, changes to variables in Python are reflected in the browser's JavaScript and vice versa. Under the hood, IPyWidgets makes use of backbone.js [6], a model-view-_ (MV_) framework for JavaScript that implements synchronization between a JavaScript *model* (where data/attributes/values for a widget are stored) and a JavaScript *view* (the code that creates the visual UI elements the user sees and interacts with in their browser). IPyWidgets connects to this backbone model via WebSockets, through which model changes and events are sent as serialized JSON.

Figure 3 shows a simplified view of the communication channels between the Python model and the JavaScript model/view for an example slider widget. Modifying an attribute on the Python model sends the modification to the JavaScript model, where any changes that would modify the visual aspects of the widget are communicated to the JavaScript view (e.g., a change in the slider value would correspondingly update the position of the slider indicator within the track). Similarly, a user interacting with the widget (e.g., moving the slider within the track) updates the JavaScript state, which communicates this change to the Python model and calls any event handlers listening for changes to the value.

## 2.2. *Common Framework*

One of the most important benefits of IPyWidgets is that it provides a low-level framework abstracting away many of the specifics for how communication between a kernel, server, and frontend takes place. This framework can be used to design new widgets by providing

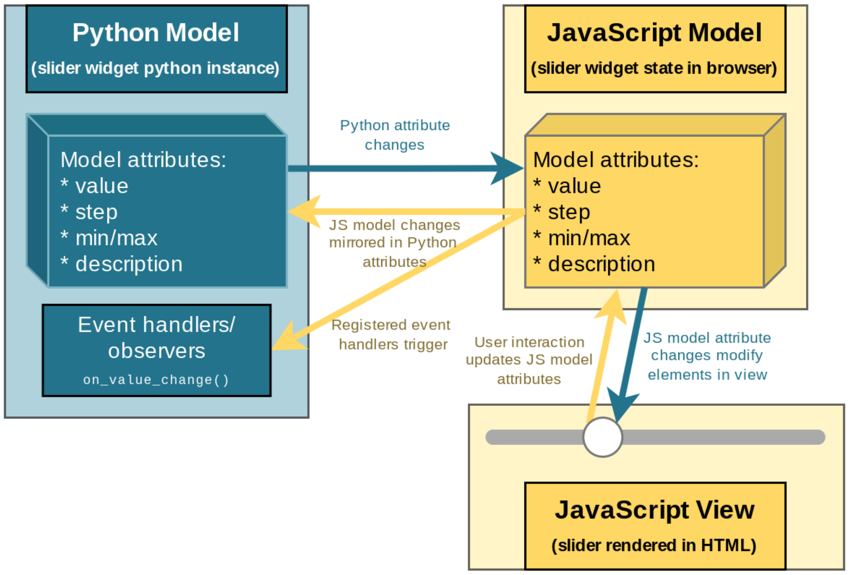

Python Model
(slider widget python instance)
Model attributes:
* value
* step
* min/max
* description
Event handlers/
observers
on_value_change()
Python attribute changes
JS model changes mirrored in Python attributes
Registered event handlers trigger
User interaction updates JS model attributes
JS model attribute changes modify elements in view
JavaScript Model
(slider widget state in browser)
Model attributes:
* value
* step
* min/max
* description
JavaScript View
(slider rendered in HTML)

**Figure 3**. *State mirroring between models and view interaction for an example slider widget.*

implementations for the models and view code, without needing to reinvent the various messaging system mechanisms that solve the state synchronization problem discussed previously. All of this has allowed many libraries to emerge, providing novel functionality, a great diversity of new types of widgets, and other frameworks that can support rendering IPyWidgets within other contexts.

## *2.3. Limitations*

Despite the flexibility of IPyWidgets as a framework and its value as a core-enabling technology for user interaction, we explicitly highlight some of the limitations of using IPyWidgets as a library of components alone. These limitations motivate the use of some of the many existing libraries that extend and build on top of the IPyWidgets framework, as well as our own work in the IPyOverlay implementation.

### 2.3.1. *Components:*

The set of default components included in the IPyWidgets library cover all standard form elements such as buttons, drop-down menus, and text fields, as well as a generic `Output` widget. The `Output` widget can render any of the MIME-types supported by normal Jupyter cell outputs, including Markdown, images, plots, and even raw HTML. Additionally, there are basic layout components to stack other widgets vertically, horizontally, or via tabs, all of which can be nested to allow grouping widgets. Although these components are often more than sufficient for simple visualizations, more complex interfaces with richer interaction paradigms can be hard to achieve. Some examples of dashboard elements that cannot be created with IPyWidgets' default components include interactive data frames, dialog windows, and floating buttons.

### 2.3.2. *Events:*

Another limiting factor for creating more complex interactivity is the type of events that can be observed on the components. UIs are inherently event-driven; the user interacts with widgets/visualizations, and these events run code that update the interface. This is most often implemented by registering “observers” or event-handlers, which tells IPyWidgets to run a specified function every time a particular event is raised. In practice, for the default components provided by the IPyWidgets library, this is mostly limited to value changes (e.g., a slider is moved, the text in a textbox is changed, or an item in a drop-down menu is selected).

These types of events are sufficient for basic interfaces, but often more complex types of interactions can greatly enhance the capabilities of a tool. The ability to detect mouse events can be crucial, enabling possibilities such as highlighting parts of a visualization when the mouse hovers over a row in a dataframe, or running an analysis when a user has highlighted a piece of text from a document. These types of user actions are not feasible with the default widgets.

### 2.3.3. *Custom Styling and Layout:*

Web pages and dashboards that need more complicated layouts than conventional linear stacks and grids of elements generally need to use CSS to control positioning. More than just a mechanism for changing aesthetic and colors, CSS provides rules for structuring the interface. However, this often comes at the cost of greater code complexity and excessive layers of nested elements and rule sets. IPyWidgets components have some capability to modify their CSS but are limited to a subset of rules available through Python attributes. This makes common style changes easier to work with, but the lack of any direct access to arbitrary CSS rules limits more complex layouts, namely the ability to float one widget on top of another.

### 2.3.4. *Custom IPyWidget Complexity:*

A key benefit of IPyWidgets as a framework is that it allows developers to design, build, and publish new components that other people can use in place of or alongside the default widgets. However, the complexity of the infrastructure required to do this with the base IPyWidgets library still makes it challenging to prototype or put together a one-off custom component in a project. Several libraries we discuss in the next section have improved this process substantially, but traditionally a custom component library was developed by starting from a Jupyter-provided cookie-cutter template [7] in a separate project.

One of the difficulties that comes with a project created in this way is the need to include the development infrastructure of both a Python and a JavaScript project. JavaScript toolchains and dependency management can quickly become complicated with the number of packages, build system tools, and config files often necessary. Prototyping the component can be a slow process as well, requiring a rebuild on every JavaScript change. Setting up Jupyter to use these unpublished components requires additional environment management to work correctly. Finally, custom widgets built through this method often require fairly significant boilerplate code because the developer is directly in charge of defining all pieces of the Python model, equivalent JavaScript model, and JavaScript view.

## 3. Beyond Base IPyWidgets

IPyWidgets as a low-level widget framework has allowed an ecosystem of other libraries and tools to develop on top of it, dramatically pushing the boundary of what can be constructed and incorporated into interactive dashboards with relative ease. From general plotting tools like interactive Matplotlib (ipympl) [8], plotly [9], mpld3 [10], altair [11], and bqplot [12], to more complex or specialized visualization widgets like pythreejs [13] and ipyvolume [14] for 3D rendering, ipyleaflet [15] for interactive maps, and ipycytoscape [16] for interactive networks, there are dozens of packages providing specific components and bridges to JavaScript visualization libraries. Other libraries such as Panel, Solara, and

Streamlit provide more holistic integrations between IPyWidgets and other visualization techniques, which can support production-style dashboards for use outside of Jupyter alone. Although these libraries address the limited component set included with IPyWidgets, another class of Python packages exists in this ecosystem that completely changes and improves the workflow for defining new widgets. These workflow improvements eliminate the need for JavaScript toolchains, and in some cases, they also provide the capabilities from various JavaScript web frameworks. A few we briefly touch on here are AnyWidget, IPyReact, and IPyVuetify.

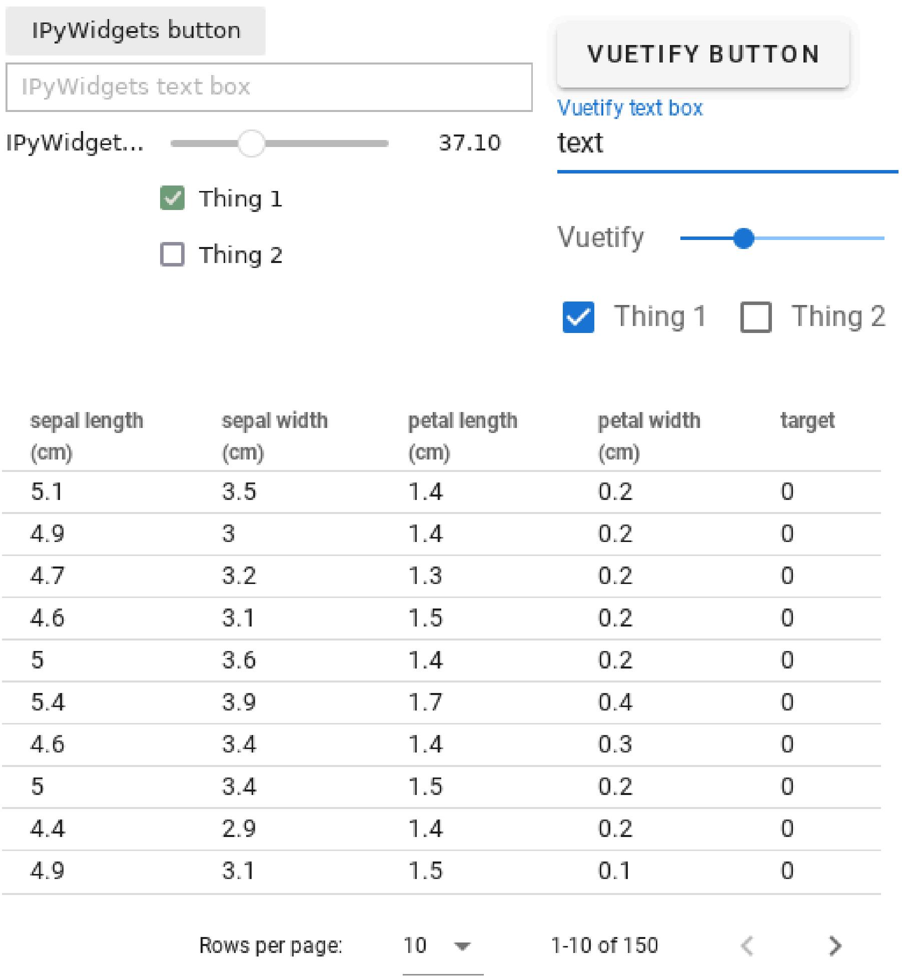
AnyWidget [17] is a recent library that has become influential in the past year. New widgets can be created by simply extending the Python `anywidget.AnyWidget` class, defining the class attributes that are to be synced between the Python and JavaScript models using the `traitlets` library [18], and providing a string of JavaScript code that specifies the view rendering logic. Much of the JavaScript MV* boilerplate code is abstracted away, and component development can be done without any JavaScript project infrastructure, specialized cookie cutter, or separate environment setup. Fundamentally, this type of approach allows even developing widgets directly within a notebook, rather than doing so in a separate project. AnyWidget is used in another project called IPyReact [19], which brings the ReactJS library to support component-based design and allows wrapping many existing React components.

A similar library we focus on in this work that addresses all of the Section 2.3 is IPyVuetify [20], which brings many of the capabilities of Vue.js into IPyWidgets. Vue.js is a web component framework similar to React and emphasises a component-driven design, where a single Vue file can contain an entire component: the template HTML to render, the CSS to apply, and the JavaScript to provide any interactivity and mechanisms to connect it to the rest of the application. Attributes within the HTML can be bound to JavaScript variables, which makes it possible to easily automate the JavaScript model → view updates. IPyVuetify brings two main advantages: a large library of flexible premade components and the ability to easily define new components on the fly using Vue to simplify the resulting JavaScript code.

### 3.1. *Vuetify Component Library*

IPyVuetify is an IPyWidgets port of the Vuetify component library, a set of more than 70 premade material design [21] UI widgets. Most of these components support a fairly extensive set of attributes and properties to change both appearance and functionality. More importantly, each IPyVuetify widget generally has multiple types of events that can be observed. A simple example would be a textbox form input component. In IPyWidgets, the `Text` widget can mostly only observe a value change when the user changes the text inside. The IPyVuetify `TextField` can additionally raise events for key presses, mouse clicks, the element gaining and losing focus, as well as any HTML event. This level of control can be useful, for example, when a text value change triggers a heavy processing task. The interaction is more responsive when the computation is delayed until the field loses focus or the user has pressed enter, rather than restarting the computation on every letter the user adds or removes.

All of the default components in IPyWidgets similarly have more flexible versions and many additional components in IPyVuetify. Figure 4 shows a small sample of default IPyWidgets on the left with their corresponding IPyVuetify versions on the right. Also pictured at the bottom is the Vuetify DataTable, which is an example of a component that is not in the default IPyWidgets. The DataTable allows interactivity to be added to a table supporting filtering, searching, and both client/server side pagination. Other examples of useful UI elements IPyVuetify adds are various layout widgets like navigation drawers, dialog windows, and steppers.



| sepal length (cm) | sepal width (cm) | petal length (cm) | petal width (cm) | target |
|---|---|---|---|---|
| 5.1 | 3.5 | 1.4 | 0.2 | 0 |
| 4.9 | 3 | 1.4 | 0.2 | 0 |
| 4.7 | 3.2 | 1.3 | 0.2 | 0 |
| 4.6 | 3.1 | 1.5 | 0.2 | 0 |
| 5 | 3.6 | 1.4 | 0.2 | 0 |
| 5.4 | 3.9 | 1.7 | 0.4 | 0 |
| 4.6 | 3.4 | 1.4 | 0.3 | 0 |
| 5 | 3.4 | 1.5 | 0.2 | 0 |
| 4.4 | 2.9 | 1.4 | 0.2 | 0 |
| 4.9 | 3.1 | 1.5 | 0.1 | 0 |

Rows per page: 10 1-10 of 150

**Figure 4**. *A visual comparison of a few IPyWidget components on the upper left and their corresponding IPyVuetify components on the upper right. The table on the bottom is a DataTable component, an example of a widget which does not have a default IPyWidgets equivalent.*

Finally, the Python API for every IPyVuetify component includes the ability to set CSS class and style attributes. The former of these provides access to a long list of predefined classes for easy and consistent control of colors, spacing, and other attributes. For anything not covered by these classes or separate attributes already part of the Vuetify components, the style attribute controls the raw in-line CSS applied to the element.

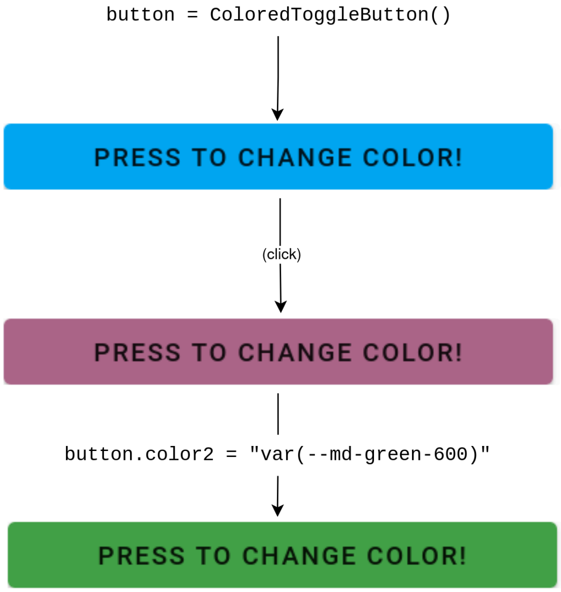
### *3.2. Custom Vue Components*

IPyVuetify provides clean abstractions and mechanisms for creating custom components. Similar to AnyWidget, IPyVuetify allows defining a Python model class with synced `traitlets`, all of which are automatically available within the JavaScript code. In IPyVuetify's case, the frontend code is expected to be a string of Vue code or a path to a Vue file, which allows for fairly self-contained HTML, JavaScript, and styling. The design of Vue.js lends itself well to reducing boilerplate by allowing bindings in the HTML to directly refer to Python attribute traitlets, removing both the need to specify the attribute in the JavaScript model as well as the function to update the view from the model. We discuss this further in the section on Section 4.1. Program 3 exemplifies this, defining a custom button with a color change on toggle, where the colors can be modified from Python as well, shown in Figure 5. In this particular example, both the `@click` event and `:color` property directly bind

```
import traitlets
import ipyvuetify as v

class ColoredToggleButton(v.VuetifyTemplate):
    color1 = traitlets.Unicode("var(--md-light-blue-500)").tag(sync=True)
    color2 = traitlets.Unicode("#AA6688").tag(sync=True)
    state = traitlets.Bool(True).tag(sync=True)

    @traitlets.default("template")
    def _template(self):
        return """
        <template>
            <v-btn
                width="300px"
                :color="state ? color1 : color2"
                @click="state = !state"
            >Press to change color!</v-btn>
        </template>
        """

button = ColoredToggleButton()

# render the button
button
```

**Program 3**. *An example of a custom IPyVuetify component.* `color1`*,* `color2`*, and* `state` *are all part of the automatically synced models. We can refer to them in the Vue bindings as shown on lines 15 and 16, where line 16 is an inline event handler that modifies the* `state` *property when the button is clicked. Updating the traitlets on the Python model updates the referenced values in the Vue bindings, e.g.* `button.color2 = "var(--md-green-600)"` *changes the second color programmatically.*

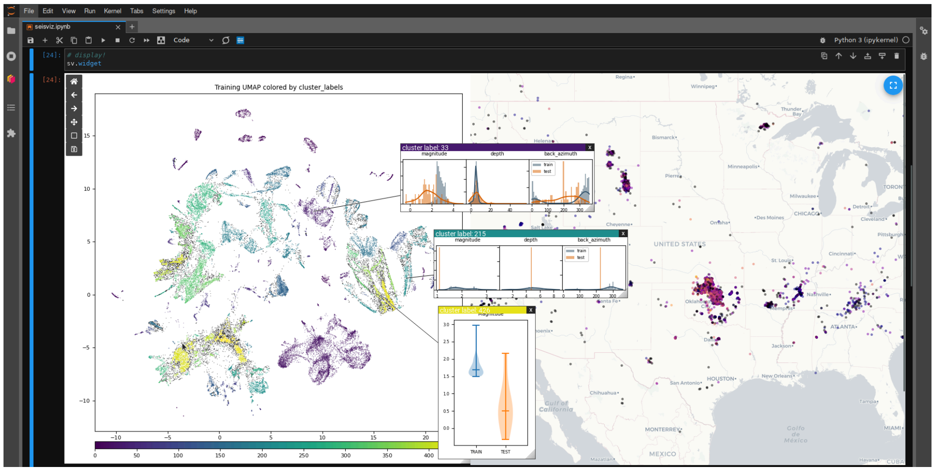
to the Python attributes, and no explicit JavaScript is necessary. Finally, IPyVuetify makes it easy to directly call functions across the Python–JavaScript link by making any Python functions prefixed with `vue*`available within JavaScript and any JavaScript functions prefixed with`jupyter\_` callable from Python.



```
button = ColoredToggleButton()
```

```
button.color2 = "var(--md-green-600)"
```

**Figure 5**. *An instance of the* `ColoredToggleButton` *from* Program 3*. Clicking the button switches to color2, and modifying the Python property* `color2` *immediately updates the button to the new second color.*



```
[24]: # display!
      sv.widget
```

**Figure 6**. *A dashboard for exploring seismic station data embedded with a self-supervised encoder model. The background layout is a Matplotlib UMAP and a plotly geographic map. Pictured are three draggable overlay ("windowed") Matplotlib plots (with colored bars on top) that provide additional details for user-selected clusters found within the UMAP.*

## 4. IPyOverlay

IPyOverlay is a Python library the authors have implemented and open-sourced[1] that uses IPyVuetify to provide several wrapper components for other IPyWidgets. These components add novel UI capabilities within the IPython ecosystem designed to enable Section 1.1. Namely, we implement support for rendering widgets on top of other widgets with arbitrary and controllable positioning. This capability enables techniques such as click-and-draggable overlay windows containing any other widget. An example of IPyOverlay in action is shown in Figure 6. Throughout this section we will discuss an example use case where we worked with one of the authors, a seismologist, to construct a dashboard for exploring models trained on seismic data. This use case led to the development of IPyOverlay as a library.

The application in this use case is a tool to assist in evaluating how a model internally represents data from seismic events. Determining properties about a seismic event's source is a critical part of analyzing and interpreting observed seismic signals, and large deep neural networks are increasingly useful for this task. Qualitatively assessing how well trained models perform on these types of problems, both as a model developer as well as an end user, requires hypothesis testing that visualization can help make substantially easier.

In this use case, the dashboard is constructed with IPyWidgets and IPyOverlay, taking advantage of the techniques described in this section, to determine the success of training a model to meet our domain-specific criteria. More specifically, we use Barlow Twins [22] as a self-supervised learning objective to try to force a model's learned representations to reflect source properties rather than other observational characteristics. For example, different geographic locations will observe slightly different signals of the same source event based on geological differences in the paths to those locations from the source. An ideal model would exhibit path invariance and represent these signals similarly. We use the constructed visualization tool along with metadata (information other than the timeseries the model was trained on, e.g. event location, depth, distance) to understand how well the training objective succeeds in learning invariance to the metadata attributes that are unrelated to the event source properties. Additionally, we use the dashboard to help end users understand how well the model generalizes to new sources and to explore uncertainty.

[1] https://github.com/ORNL/ipyoverlay

The dashboard includes a UMAP [23] plot of signals embedded by the model under exploration. Each datapoint is a single observed seismic signal, and these data points are then clustered with DBSCAN [24]. Metadata attributes aggregated by cluster can be explored by clicking within the UMAP ipympl plot, which aids in determining how much correlation exists for those attributes within that cluster. Invariant properties would tend to show as spread out distributions, while attributes that heavily inform a models' embedding would tend to cause clustering and produce narrow distributions.

### *4.1. Implementing in IPyWidgets*

Within base IPyWidgets, there is no straightforward way of constructing a layout in which one widget is overlaid on top of another. Certain other libraries have limited functionality for this within their own widget instances. For example Matplotlib and plotly can display an axes within another axes. Furthermore, many libraries support basic tooltips with arbitrary textual content. Others such as IPyVuetify support certain types of limited "dialog" overlays, which can contain inner widgets and display overtop of the rest of the webpage, but do not support the same type of arbitrary positioning necessary for some details-on-demand approaches.

This limitation seems to arise primarily from the conventional layout mechanisms employed by many widget libraries, which consists of nesting rows and columns (`HBox` and `VBox` from IPyWidgets) of CSS-flexbox style layouts, or even stricter grid layouts.

These approaches are generally easier to design and work with, and abstract away or entirely avoid some of the messier CSS layout rules required to render HTML elements that do not follow an ordered top-to-bottom/left-to-right flow. Fundamentally, to enable overlay layouts, we implement custom IPyWidget containers that use specific CSS rules to control for nonlinear layouts, namely relying on absolute positioning (by setting `position: absolute`), and each child widget setting `left` and `top` CSS properties to have pixel-level control of its location relative to the parent container. A diagram demonstrating this design principle is shown in Figure 7.

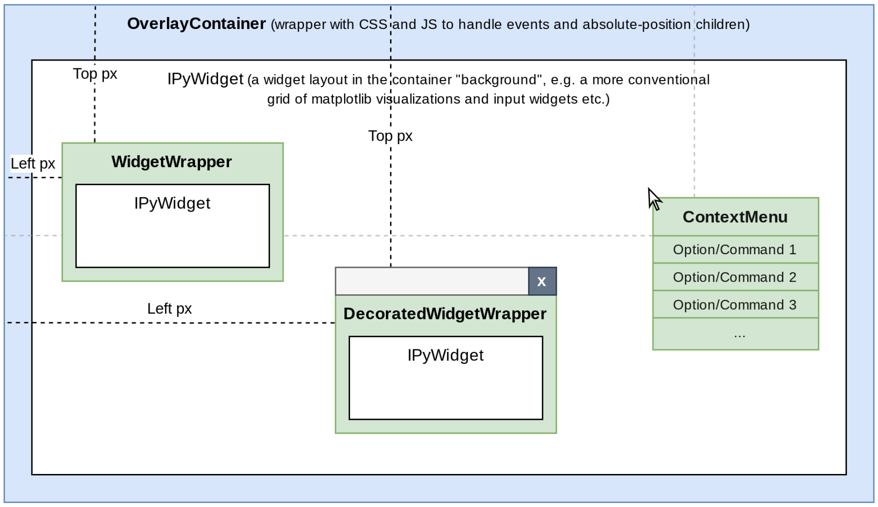

OverlayContainer (wrapper with CSS and JS to handle events and absolute-position children)
Top px
IPyWidget (a widget layout in the container "background", e.g. a more conventional grid of matplotlib visualizations and input widgets etc.)
Top px
Left px
WidgetWrapper
IPyWidget
ContextMenu
Option/Command 1
Option/Command 2
Option/Command 3
...
x
Left px
DecoratedWidgetWrapper
IPyWidget

**Figure 7**. *Component wrappers provided by IPyOverlay (discussed in the following section) demonstrating how the positioning works for absolutely positioned child widgets.*

### *4.2. Embedding IPyWidgets in IPyOverlay Containers*

A key enabler is IPyVuetify's ability to embed other IPyWidgets inside of the template HTML, allowing us to construct custom layouts of one or more child widgets, similar to `HBox` and `VBox` in IPyWidgets. In effect, we can wrap any other widget with custom HTML, CSS, and JavaScript. The Vue framework is particularly well suited to concise implementation with this approach. Vue-flavored template HTML supports binding reactive properties on attributes (e.g., `<div :width="some_JS_variable_name">`), meaning we can reference any model variables in the template syntax, and any time those attributes update, the corresponding HTML is automatically updated without other code having to monitor and orchestrate the JavaScript model → view update. Additionally, Vue's template syntax supports several programmatic control structures such as looping (`v-for`) and conditionals (`v-if`), all of which combined allows declaratively constructing a bound list of child widgets each wrapped in custom HTML that reactively responds to list updates and other model property changes. A simplified example is shown in Program 4.

Our primary contribution with IPyOverlay is an `OverlayContainer` widget, which creates a container `<div>` tag embedding a background (nonoverlay) widget, which can consist of a more traditional grid-like layout of nested rows/columns of other IPyWidgets, and absolutely positioned floating `<div>` tags for each child widget. To support additional functionalities in any of the child overlay widgets, such as the ability to click and drag to

```
<template>
    <div class="ipyoverlay-container">
        <!-- iterate and create a div container for every child in the python model
        (children is a list of IPyWidgets) -->
        <div v-for="(child, index) in children"
            class="ipyoverlay-child-container"
            :style="{ left: childPosLeft[index]+'px', top: childPosTop[index]+'px' }"
        >
            <!-- embed the IPyWidget -->
            <jupyter-widget :widget='child' />
        </div>
    </div>
</template>

<script>
module.exports = {
    data() {
        return {
            // any values in the data dictionary can be used reactively within
            // the template HTML
            childPosLeft: [],
            childPosTop: [],
        };
    },
}
</script>

<style>
.ipyoverlay-container {
    display: block;
}
.ipyoverlay-child-container {
    display: block;
    position: absolute;
}
</style>
```

**Program 4**. *A simplified version of the Vue template for `OverlayContainer`. The `<div>` tag with `v-for` reactively produces a `<div>` tag for each widget in the `children` list attribute within the Python model. Note, the style attribute is bound and references the position arrays on lines 21–22. Changing the position of each overlay widget is done by modifying the corresponding values in these arrays. We keep these arrays only on the JavaScript side to prevent undue traffic between the Python and JavaScript models from high-frequency events (e.g., mouse movements) that can cause significant lag in the interface.*

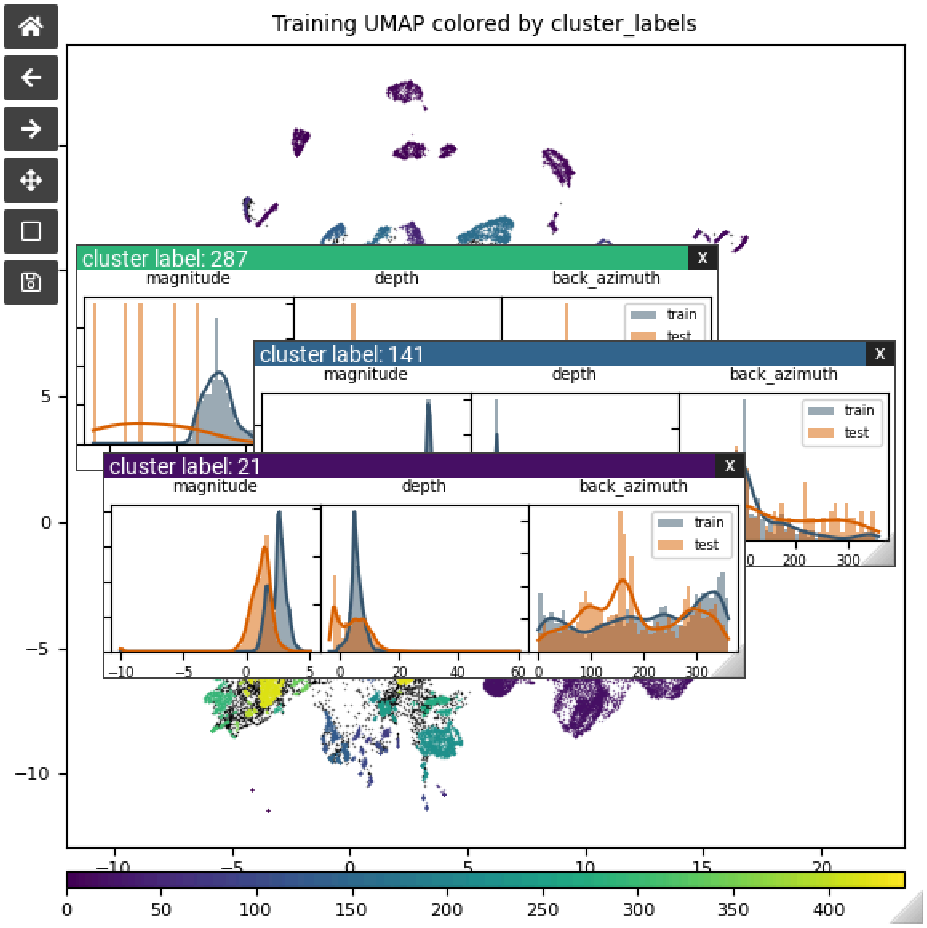


**Figure 8.** *Three* `DecoratedWidgetWrapper` *widgets, each wrapping an ipympl widget with more in-depth plots detailing distributions of attributes within selected clusters.*

move them, we provide a `WidgetWrapper` (clicking anywhere within begins the drag) and a `DecoratedWidgetWrapper` (a bar is rendered above the widget which can be clicked and dragged to move the entire widget.) An example is shown in Figure 8, where the background widget is an ipympl Matplotlib plot and three `DecoratedWidgetWrapper`'s that embed additional ipympl Matplotlib plots are floating freely above the background.

### *4.3. Connection Lines*

A potential concern with free-floating details-on-demand overlays is the dissociation from the underlying data: if a user opens more than a couple overlays that correspond to specific pieces of an underlying visualization, it is easy to lose track of the overlay widgets and the pieces they are associated with. In the example use case, clicking on a cluster within the UMAP opens distribution plots for data within that cluster. Because of the number of clusters, it can be difficult to determine which distribution plot belongs to which cluster. IPyOverlay adds the ability to render lines connecting overlay widgets to specific pieces of the background widget (Figure 9).

One end point of the line is always associated with a specified overlay widget and is automatically updated to the widget's center whenever it is moved. The other side of the line can be "attached" in a few different ways, including simply specifying pixel coordinates within the background, but a significant challenge is locations within dynamic content such as interactive plots that can be panned and scaled. The conceptual approach to solving

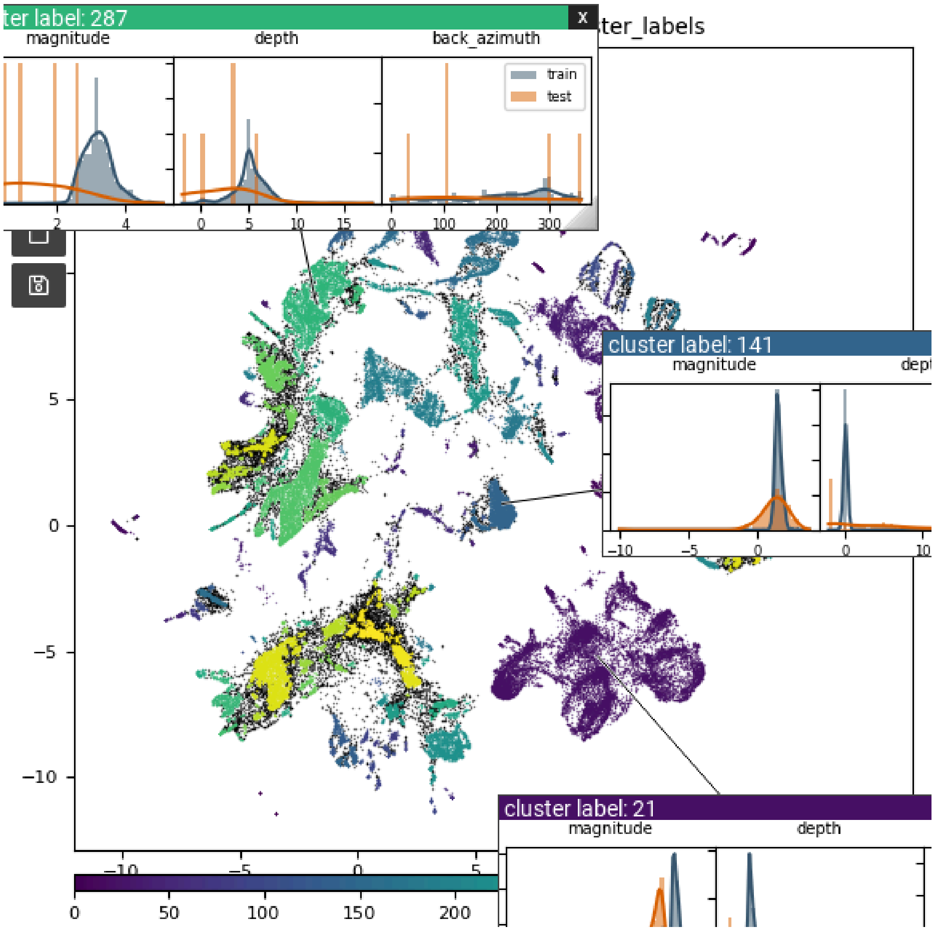


**Figure 9.** *The same detail plots as in Figure 8. The overlay widgets have been dragged further out to show the connection lines to the clusters in the UMAP each is based on.*

this for many plotting libraries is to store the x/y data point the connection should point to, the initial ranges of any plot axes, and current pixel boundaries of the plot within the container. Then, listening for any events that indicate an axis range has changed, handler functions can recalculate pixel locations based on the new relative position of the x/y data point within the new range.

Unfortunately, in practice, the functions necessary to do this for different plotting libraries can vary substantially as a result of implementation differences. At the time of this writing, IPyOverlay only supports autoupdating connections for dynamic content inside of ipympl (Matplotlib) and plotly plots. The authors intend to continue expanding the set of supported visualization libraries for this particular feature.

### *4.4. Context Menus*

A common form of details-on-demand feature in UI design across most platforms is the context menu. Generally initiated from a right-click in desktop operating systems and applications, or tapping and holding on a mobile device, context menus provide a localized and easy to access palette of possible commands to run at the current location. Complex dashboards can often have many different possible actions for the user to take, and simply displaying all of these as buttons may overwhelm or clutter the interface. Alternately, the actions may be too component/position specific to work well as buttons. Context menus fill this role by hiding these commands until the user requests them by right-clicking and

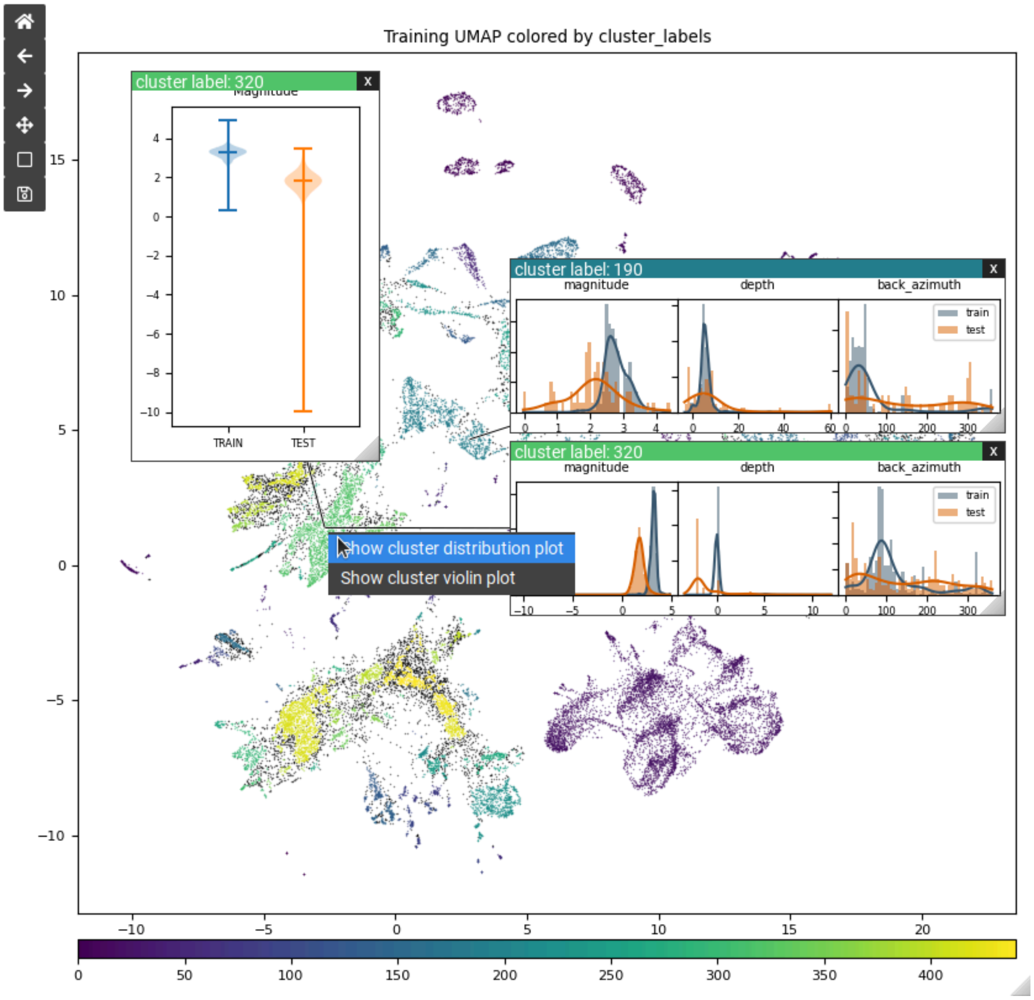


**Figure 10**. *An activated context menu providing a selection of commands, which can be expanded well beyond the single possible action that can be taken from a left-click.*

supporting different menus when over different components. Displaying a context menu similarly requires rendering some form of widget over another – IPyOverlay includes a `ContextMenuArea` component that can wrap any widget within an `OverlayContainer` and define a context menu with options and event handlers for that specific component. In the example use case, we add a context menu to the scatter plot to allow a user to optionally choose between displaying a violin plot with the distributions versus the density plots (Figure 10).

### *4.5. Example Usage*

A minimal code example using IPyOverlay's containers and connection lines is shown in Program 5. The example renders an interactive ipympl scatterplot of random data, and clicking on a point renders a connected overlay window with a bar chart of additional data associated with the point. A screenshot of this output is shown in Figure 11.

## 5. Conclusion

The prevalence and utility of Jupyter notebooks in data analysis and scientific workflows has prompted the development of a community of tools for interactive visualization. IPyWidgets, the backbone that supports much of this ecosystem, provides the mechanisms for the kernel to mediate communication between Python and JavaScript. A multitude of

```
import ipyoverlay as ui
import matplotlib.pyplot as plt
import numpy as np

%matplotlib widget

# make random data to plot - the first two dimensions are x and y coords
# and the latter 3 are associated data we want to see in inset plots
data = np.random.rand(100, 5)

# plot the scatter data dimensions
fig, ax = plt.subplots()
ax.scatter(x=data[:,0], y=data[:,1])

# make an IPyOverlay Container to wrap the ipympl figure to allow inset plots
container = ui.OverlayContainer(fig.canvas, height="auto", width="auto")

def on_point_click(point_index, event):
    # get the data associated with selected point
    data_point = data[point_index]
    category_bar_data = data_point[2:]

    # create bar chart for selected point
    inset_fig, inset_ax = plt.subplots(figsize=(2,2))
    inset_ax.bar(x=[0,1,2], height=category_bar_data)

    # make a floating IPyOverlay window with the plot and connect it
    # to the selected point's data location
    inset_window = ui.DecoratedWidgetWrapper(ui.display_output(inset_fig), title=str(point_index))
    container.add_child_at_mpl_point(inset_window, ax, data_point[0], data_point[1])

# attach the event handler to add the inset plot when any of the specified
# datapoints are clicked
handler = ui.mpl.event.on_mpl_point_click(ax, on_point_click, data[:,0], data[:,1], tolerance=.1)

container
```

**Program 5**. *This snippet wraps an `ipympl` figure with an `OverlayContainer` and responds to point clicks by adding a floating `DecoratedWidgetWrapper` with an additional plot, connected to the clicked point. Note that `display_output()` is a convenience function provided by IPyOverlay that can render a static plot into a default ipywidgets `Output` widget.*

libraries have built on top of these mechanisms and IPyVuetify in particular offers a collection of well designed and flexible components to use in dashboards and a straightforward abstraction for building custom components in projects based on Vue.js. A missing piece of this ecosystem is the ability to flexibly render localized pop-up windows in support of details on demand. We developed IPyOverlay to provide this capability through wrapper components that can arbitrarily position widgets on top of other widgets, opening the door to more types of visualization and dashboard layouts.

## Acknowledgements

Notice: This manuscript has been authored by UT-Battelle, LLC, under contract DE-AC05-00OR22725 with the US Department of Energy (DOE). 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 (https://www.energy.gov/doe-public-access-plan).

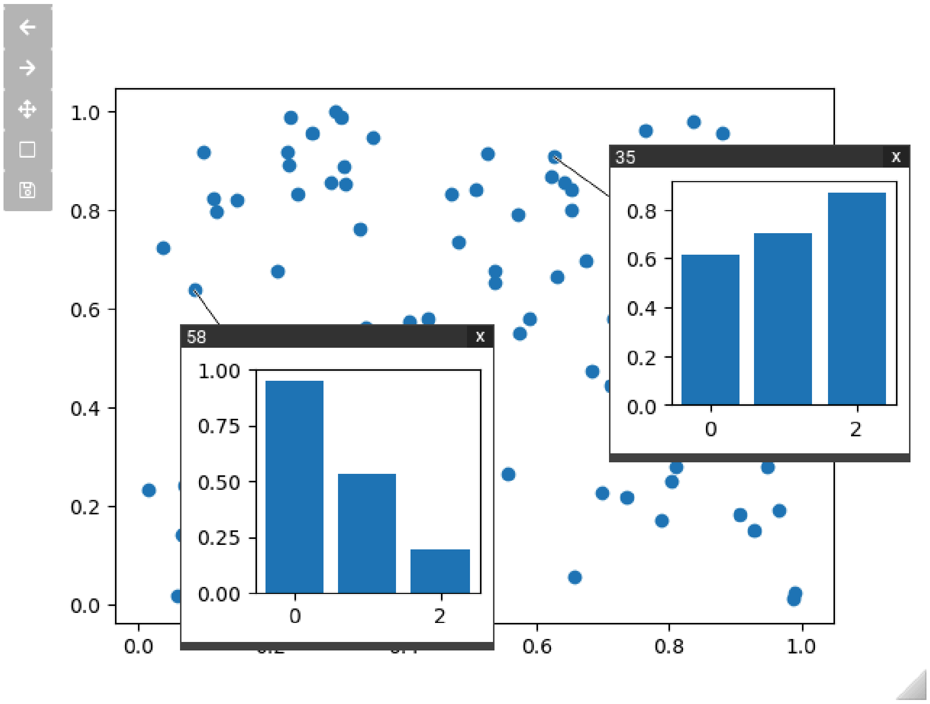


**Figure 11**. *An* `ipympl` *scatter plot of random data with two inset plots showing the additional data associated with each clicked point in a bar chart.*

## References

[1] T. Kluyver *et al.*, "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., Netherlands, 2016, pp. 87–90. [Online]. Available: https://eprints.soton.ac.uk/403913/

[2] Jupyter Widgets, "jupyter-widgets/ipywidgets." [Online]. Available: https://github.com/jupyter-widgets/ipywidgets

[3] Philipp Rudiger *et al.*, "holoviz/panel: Version 1.4.2." [Online]. Available: https://zenodo.org/doi/10.5281/zenodo.11046027

[4] M. A. Breddels, "widgetti/solara." [Online]. Available: https://github.com/widgetti/solara

[5] 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. doi: 10.1109/VL.1996.545307.

[6] Jupyter Widgets, "Low Level Widget Explanation — Jupyter Widgets 8.1.2 documentation." [Online]. Available: https://ipywidgets.readthedocs.io/en/latest/examples/Widget%20Low%20Level.html

[7] Jupyter Widgets, "jupyter-widgets/widget-cookiecutter: A cookiecutter template for creating a custom Jupyter widget project.." [Online]. Available: https://github.com/jupyter-widgets/widget-cookiecutter

[8] M. Renou *et al.*, "matplotlib/ipympl: Release 0.9.4." [Online]. Available: https://doi.org/10.5281/zenodo.10974323

[9] Plotly, Inc., "plotly/plotly.py: The interactive graphing library for Python." [Online]. Available: https://github.com/plotly/plotly.py

[10] J. VanderPlas, "mpld3/mpld3." [Online]. Available: https://github.com/mpld3/mpld3

[11] J. VanderPlas *et al.*, "Altair: Interactive Statistical Visualizations for Python," *Journal of Open Source Software*, vol. 3, no. 32, p. 1057, 2018, doi: 10.21105/joss.01057.

[12] The BQplot Project, "bqplot/bqplot." [Online]. Available: https://github.com/bqplot/bqplot

[13] Jupyter Widgets, "jupyter-widgets/pythreejs." [Online]. Available: https://github.com/jupyter-widgets/pythreejs

[14] M. Breddels *et al.*, "maartenbreddels/ipyvolume: ipyvolume v0.4.5." [Online]. Available: https://zenodo.org/record/1286976

[15] Jupyter Widgets, "jupyter-widgets/ipyleaflet." [Online]. Available: https://github.com/jupyter-widgets/ipyleaflet

[16] M. Meireles, "cytoscape/ipycytoscape." [Online]. Available: https://github.com/cytoscape/ipycytoscape

[17] T. Manz, "anywidget." [Online]. Available: https://zenodo.org/doi/10.5281/zenodo.11182005

[18] IPython Development Team, "ipython/traitlets." [Online]. Available: https://github.com/ipython/traitlets

[19] M. A. Breddels, “widgetti/ipyreact.” [Online]. Available: https://github.com/widgetti/ipyreact

[20] M. Buikhuizen, “widgetti/ipyvuetify: Jupyter widgets based on vuetify UI components.” [Online]. Available: https://github.com/widgetti/ipyvuetify

[21] Google, “Material Design.” [Online]. Available: https://material.io/

[22] J. Zbontar, L. Jing, I. Misra, Y. LeCun, and S. Deny, “Barlow Twins: Self-supervised learning via redundancy reduction,” in *Proceedings of the 38th International Conference on Machine Learning*, 2021, pp. 12310–12320. [Online]. Available: https://proceedings.mlr.press/v139/zbontar21a.html

[23] L. McInnes, J. Healy, and J. Melville, “UMAP: Uniform Manifold Approximation and Projection for Dimension Reduction.” [Online]. Available: https://arxiv.org/abs/1802.03426

[24] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu, “A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise,” in *Knowledge Discovery and Data Mining*, 1996. [Online]. Available: https://api.semanticscholar.org/CorpusID:355163

Open Access | https://doi.org/10.25080/DDJJ4932

# Orchestrating Bioinformatics Workflows Across a Heterogeneous Toolset with Flyte

**Pryce Turner**[1]

[1] Union AI

**SciPy 2024**

*July 8 - July 14, 2024*

Proceedings of the 23nd
Python in Science Conference
ISSN: 2575-9752

**Published** Jul 10, 2024

**Correspondence to**
Pryce Turner
pryce.turner@gmail.com

## Abstract

While Python excels at prototyping and iterating quickly, it's not always performant enough for whole-genome scale data processing. Flyte, an open-source Python-based workflow orchestrator, presents an excellent way to tie together the myriad tools required to run bioinformatics workflows. Flyte is a Kubernetes native orchestrator, meaning all dependencies are captured and versioned in container images. It also allows you to define custom types in Python representing genomic datasets, enabling a powerful way to enforce compatibility across tools. Finally, Flyte provides a number of different abstractions for wrapping these tools, enabling further standardization. Computational biologists, or any scientists processing data with a heterogeneous toolset, stand to benefit from a common orchestration layer that is opinionated yet flexible.

**Keywords** flyte, orchestration, bioinformatics

## 1. Introduction

Since the sequencing of the human genome [1], and as other wet lab processes have scaled in the last couple decades, computational approaches to understanding the living world have exploded. The firehose of data generated from all these experiments led to algorithms and heuristics developed in low-level high-performance languages such as C and C++. Later on, industry standard collections of tools like the Genome Analysis ToolKit (GATK) [2] were written in Java. A number of less performance intensive offerings such as MultiQC [3] are written in Python; and R is used extensively where it excels: visualization and statistical modeling. Finally, newer deep-learning models and Rust based components are entering the fray.

Different languages also come with different dependencies and approaches to dependency management, interpreted versus compiled languages for example handle this very differently. They also need to be installed correctly and available in the user's `PATH` for execution. Moreover, compatibility between different tools in bioinformatics often falls back on standard file types expected in specific locations on a traditional filesystem. In practice this means searching through datafiles or indices available at a particular directory and expecting a specific naming convention or filetype.

In short, bioinformatics suffers from the same reproducibility crisis [4] as the broader scientific landscape. Standardizing interfaces, as well as orchestrating and encapsulating these different tools in a flexible and future-proof way is of paramount importance on this unrelenting march towards larger and larger datasets.

## 2. Methods

Solving these problems using Flyte is accomplished by capturing dependencies flexibly with dynamically generated container images, defining custom types to enforce at the task boundary, and wrapping tools in Flyte tasks. Before diving into the finer points, a brief primer on Flyte is advised. While the introduction in the docs is a worthwhile read before continuing, here is a more concise *“hello world”* example:

```
from flytekit import task, workflow

@task
def greet() -> str:
  return "Hello"

@task
def say(greeting: str, name: str) -> str:
    return f"{greeting}, {name}!"

@workflow
def hello_world_wf(name: str = "world") -> str:
    greeting = greet()
    res = say(greeting=greeting, name=name)
    return res
```

Tasks are the most basic unit of work in Flyte. They are pure-Python functions and their interface is strongly typed in order to compose the workflow. The workflow itself is actually a domain-specific language that statically compiles a directed-acyclic graph (DAG) based on the dependencies between the different tasks. There are different flavors of tasks and workflows as we’ll see later, but this is the core concept.

The following example details a bioinformatics workflow built using Flyte. All code is drawn from the ever-evolving unionbio Github repository. There are many more datatypes, tasks and workflows defined there. Questions are always welcome and contributions are of course encouraged!

### *2.1. Images*

While it’s possible to run Flyte tasks and workflows locally in a Python virtual environment, production executions in Flyte run on a Kubernetes cluster. As a kubernetes native orchestrator, all tasks run in their own (typically single-container) pods using whatever image is specified in the `@task` decorator. Capturing dependencies in container images has been a standard for some time now, but this is taken a step further with ImageSpec. ImageSpec lets you easily define a base image and additional dependencies right alongside your task and workflow code. Additional dependencies from PyPI, Conda, or `apt` are supported out-of-box. Arbitrary `RUN` commands are also available for base images lacking Debian’s package manager, or to accomplish virtually anything not currently implemented. Finally, while envd is the default builder, other backends like a local Docker daemon or even remote builders are available should the need arise.

These ImageSpec definitions are loosely coupled to your workflow code and are built automatically when tasks are registered or run on a Flyte cluster. ImageSpec reduces the complexity inherent in manually authoring a Dockerfile and enables a more streamlined approach to building images without the need for an additional build step and configuration update to reference the latest image. This coupling and reduced complexity makes it easier to build single-purpose images instead of throwing everything into one monolithic image.

```python
main_img = ImageSpec(
    name="main",
    platform="linux/amd64",
    python_version="3.11",
    packages=["flytekit"],
    conda_channels=["bioconda"],
    conda_packages=[
        "samtools",
        "bcftools",
        "fastp",
        "bowtie2",
        "gatk4",
        "fastqc",
    ],
    registry="docker.io/unionbio",
)

folding_img = ImageSpec(
    name="protein",
    platform="linux/amd64",
    python_version="3.11",
    packages=["flytekit", "transformers", "torch"],
    conda_channels=["bioconda", "conda-forge"],
    conda_packages=[
        "prodigal",
        "biotite",
        "bioPython",
        "py3Dmol",
        "matplotlib",
    ],
    registry="docker.io/unionbio",
)
```

The `main` image has a lot of functionality and could arguably be pared down. It contains a number of very common low-level tools, along with GATK and a couple aligners [5]. The `protein` image on the other hand, only contains a handful of tools related to a very specific protein folding and visualization workflow. Unless using a remote builder, these images are built locally and then pushed to the registry specified. They will persist in the builder's local registry and leverage the builder's cache until cleaned-up. Once built, they are Open Container Initiative (OCI) compliant container images like any other, allowing you to compose them as you see fit. The main image could be used as the base for the folding image, for example. Another very simple but powerful use case would be to *Flytify* any off-the-shelf image by simply specifying a Python version and adding `flytekit` as a package.

Currently, a subset of the full Dockerfile functionality has been reimplemented in ImageSpec. A typical Dockerfile could include pulling Micromamba binaries, creating and activating an environment, before finally installing the relevant packages. ImageSpec's opinionated approach enables a simpler experience by handling this kind of boilerplate code behind-the-scenes. ImageSpec is also context aware in the same way `docker build` is, meaning a `source_root` containing a lock or env file can be specified and installed if you want to keep your local environment in sync. The images submodule of the `unionbio` repo puts this in practice, with different source roots used in test and production.

ImageSpecs can be specified in the task decorator alongside any infrastructure requirements in a very granular fashion:

```
@task(
    container_image=folding_img,
    requests=Resources(cpu="4", mem="32Gi", gpu="1"),
    accelerator=GPUAccelerator("nvidia-tesla-v100"),
    )
def predict_structure(seq: str):
  fold_protein(seq)
```

This image will be built and uploaded when your tasks and workflows are registered to a Flyte cluster.

### *2.2. Datatypes*

Having rich data types to enforce compatibility at the task boundary is essential to these wrapped tools working together. Flyte supports arbitrary data types through Python's dataclasses library. Genomics pipelines typically pass around one or many large text files related to the same sample. Data types capturing these files allow us to reason about them and their metadata more easily across tasks, as well as enforce naming conventions.

Importantly, Flyte abstracts the object store, allowing you to load these assets into pods wherever is most convenient for your tool. This not only makes it easier to work with these files, but also safer as you're working with ephemeral storage during execution instead of a shared production filesystem. In a shared filesystem, unintended side-effects could mutate artifacts unrelated to the current production run. This can be mitigated in a number of ways, such as setting up an empty directory for every experiment or restricting permissions to files after a run is complete. In an ephemeral setting however, inputs of interest are materialized at the beginning of the task and any relevant outputs are serialized to a unique prefix in the object store when the task completes. Any unintended modifications disappear when the pod is deleted.

Sequencers employ different strategies to produce millions of short strings representing DNA fragments called reads. One such strategy is single-read sequencing, which produces one sequencing read per DNA fragment. This is appropriate for simpler analyses such as profiling or for analyzing less complex genomes. Paired-end sequencing, on the other hand, produces two reads representing both sequencing directions of the DNA fragment. This is essential in applications such as "de novo" assembly which involves re-assembling these reads into a whole genome without a known reference as a guide. It also unlocks the detection of larger variations, typically longer insertions or deletions.

Regardless of the sequencing strategy, these reads are captured in one or a pair of FastQ files, an example of which is given below:

```
@SRR001666.1 071112_SLXA-EAS1_s_7:5:1:817:345 length=72
GGGTGATGGCCGCTGCCGATGGCGTCAAATCCCACCAAGTTACCCTTAACAACTTAAGGGTTTTCAAATAGA
+SRR001666.1 071112_SLXA-EAS1_s_7:5:1:817:345 length=72
IIIIIIIIIIIIIIIIIIIIIIIIIIIIII9IG9ICIIIIIIIIIIIIIIIIIIIIDIIIIIII>IIIIII/
@SRR001666.2 071112_SLXA-EAS1_s_7:5:1:801:338 length=72
GTTCAGGGATACGACGTTTGTATTTTAAGAATCTGAAGCAGAAGTCGATGATAATACGCGTCGTTTTATCAT
+SRR001666.2 071112_SLXA-EAS1_s_7:5:1:801:338 length=72
IIIIIIIIIIIIIIIIIIIIIIIIIIIIIIII6IBIIIIIIIIIIIIIIIIIIIIIIIGII>IIIII-I)8I
```

Two reads are present here, characterized by the nucleobases A, C, T, and G. The other lines contain metadata and quality information. Here is the dataclass that encapsulates these files and any salient information about them:

```
@dataclass
class Reads(DataClassJSONMixin):

    sample: str
    filtered: bool | None = None
    filt_report: FlyteFile | None = None
    uread: FlyteFile | None = None
    read1: FlyteFile | None = None
    read2: FlyteFile | None = None

    def get_read_fnames(self):
        filt = "filt." if self.filtered else ""
        return (
            f"{self.sample}_1.{filt}fastq.gz",
            f"{self.sample}_2.{filt}fastq.gz",
        )

    def get_report_fname(self):
        return f"{self.sample}_fastq-filter-report.json"

    @classmethod
    def make_all(cls, dir: Path):
      ...
```

We're capturing a few important aspects: whether the reads have been filtered and the results of that operation, as well as if they're paired-end reads or not. Paired-end reads will populate the `read1` and `read2` attributes. If they are unpaired then a single FastQ file representing a sample's reads is defined in the `uread` field. The presence or absence of these attributes implicitly disambiguates the sequencing strategy.

Additionally, the `make_all` function body has been omitted for brevity, but it accepts a directory and returns a list of these objects based on it's contents. In the other direction, a `get_read_fnames` method is defined to standardize naming conventions based on The 1000 Genomes Project guidelines.

FlyteFile, along with FlyteDirectory, represent a file or directory in a Flyte aware context. These types handle serialization and deserialization into and out of the object store. They re-implement a number of common filesystem operations like `open()`, which returns a streaming handle, for example. Simply returning a FlyteFile from a task will automatically upload it to whatever object store is defined. This unassuming piece of functionality is one of Flyte's key strengths: abstracting data management so researchers can focus on their task code. Since dataflow in Flyte is a first-class construct, having well defined inputs and outputs at the task boundary makes authoring workflows that much more reliable.

In order to accomplish sequencing in a sensible timeframe, reads generation is massively parallelized [6]. This dramatically improves the throughput, but removes crucial information regarding the location of those reads. In order to recover that information, the reads are aligned to a known reference genome, producing an Alignment file, which we also capture in a dataclass:

```
@dataclass
class Alignment(DataClassJSONMixin):

    sample: str
    aligner: str
    format: str | None = None
    alignment: FlyteFile | None = None
    alignment_idx: FlyteFile | None = None
    alignment_report: FlyteFile | None = None
    sorted: bool | None = None
    deduped: bool | None = None
    bqsr_report: FlyteFile | None = None

    def _get_state_str(self):
        state = f"{self.sample}_{self.aligner}"
        if self.sorted:
            state += "_sorted"
        if self.deduped:
            state += "_deduped"
        return state

    def get_alignment_fname(self):
        return f"{self._get_state_str()}_aligned.{self.format}"

    @classmethod
    def make_all(cls, dir: Path):
      ...
```

Compared to the Reads dataclass, the attributes captured here are of course only relevant to Alignments. However, the methods that interact with the local filesystem and enforce naming conventions remain. In the next section, we'll look at tasks that actually carry out this alignment.

### *2.3. Tasks*

While Flyte tasks are written in Python, there are a few ways to wrap arbitrary tools. ShellTasks are one such way, allowing you to define scripts as multi-line strings in Python. For added flexibility around packing and unpacking data types before and after execution, Flyte also ships with a `subproc_execute` function which can be used in vanilla Python tasks. Finally, arbitrary images can be used via a ContainerTask and avoid any `flytekit` dependency altogether.

Bowtie2 [7], a fast and memory efficient aligner, is used to carry out the aforementioned alignments. Before alignment can be carried out efficiently, an index must be generated from the reference genome. Indices are generated by pre-processing the reference into a data structure that enables rapid lookup of a match to a given read. Bowtie2 uses an FM-index which combines the Burrows-Wheeler Transform (BWT) with a suffix array. Broadly speaking, BWT enables compression of the data while the suffix array allows for efficient lookup of substrings. Here is a ShellTask creating a `bowtie2` index directory from a genome reference file:

```
bowtie2_index = ShellTask(
    name="bowtie2-index",
    debug=True,
    requests=Resources(cpu="4", mem="10Gi"),
    metadata=TaskMetadata(retries=3, cache=True, cache_version=ref_hash),
    container_image=main_img,
    script="""
    mkdir {outputs.idx}
    bowtie2-build {inputs.ref} {outputs.idx}/bt2_idx
    """,
    inputs=kwtypes(ref=FlyteFile),
    output_locs=[
        OutputLocation(var="idx", var_type=FlyteDirectory, location="/tmp/bt2_idx")
    ],
)
```

This task uses the `main_img` defined above; it also accepts a FlyteFile and outputs a FlyteDirectory. Another important feature to highlight here is caching, which saves us valuable compute for inputs that rarely change. Since the alignment index for a particular aligner only needs to be generated once for a given reference, we've set the `cache_version` to a hash of the reference's URI. As long as the reference exists at that URI, this bowtie indexing task will complete immediately and return that index. To perform the actual alignment, a regular Python task is used with a Flyte-aware subprocess function to call the bowtie CLI.

```
@task(container_image=main_img, requests=Resources(cpu="4", mem="10Gi"))
def bowtie2_align_paired_reads(idx: FlyteDirectory, fs: Reads) -> Alignment:
    idx.download()
    ldir = Path(current_context().working_directory)

    alignment = Alignment(fs.sample, "bowtie2", "sam", sorted=False, deduped=False)
    al = ldir.joinpath(alignment.get_alignment_fname())
    rep = ldir.joinpath(alignment.get_report_fname())

    cmd = [
        "bowtie2",
        "-x",
        f"{idx.path}/bt2_idx",
        "-1",
        fs.read1.path,
        "-2",
        fs.read2.path,
        "-S",
        al,
    ]

    result = subproc_execute(cmd)

    # Bowtie2 alignment writes stats to stderr
    with open(rep, "w") as f:
        f.write(result.error)

    alignment.alignment = FlyteFile(path=str(al))
    alignment.alignment_report = FlyteFile(path=str(rep))

    return alignment
```

Since Python tasks are the default task type, they're the most feature rich and stable. The main advantage to using one here is to unpack the inputs and construct the output type.

The resulting alignments are then deduplicated, sorted, recalibrated and reformatted. While important, these tasks are similarly implemented to the above alignment function and as such are omitted to focus on the next major step: variant calling. Variant calling has such a diversity of approaches that any meaningful exploration of the landscape is out of scope. However, the central purpose is to distill the aligned reads into a set of likely relevant loci that deviate from the reference. This is accomplished by aggregating information like quality scores and the number of reads covering a given location, called a pileup, to come to consensus around a most likely call.

```
@task(container_image=main_img_fqn)
def haplotype_caller(ref: Reference, al: Alignment) -> VCF:
    ref.aggregate()
    al.aggregate()
    vcf_out = VCF(sample=al.sample, caller="gatk-hc")
    vcf_fn = vcf_out.get_vcf_fname()
    vcf_idx_fn = vcf_out.get_vcf_idx_fname()

    hc_cmd = [
        "gatk",
        "HaplotypeCaller",
        "-R",
        ref.get_ref_path(),
        "-I",
        al.alignment.path,
        "-O",
        vcf_fn,
    ]

    subproc_execute(hc_cmd)

    vcf_out.vcf = FlyteFile(path=vcf_fn)
    vcf_out.vcf_idx = FlyteFile(path=vcf_idx_fn)

    return vcf_out
```

GATK's HaplotypeCaller is used to perform variant calling by accepting an Alignment and Reference object and producing a Variant Call Format (VCF) file. VCFs are another common tabular text file, with each row representing a variant present in the input sequences and its alternate in the reference, along with a quality score and extensible columns for adding additional information. Variants can range from single-nucleotide polymorphisms (SNPs) to much larger structural variations mentioned above. Once variants are called, they can be further filtered downstream for certain characteristics, or against one of the many curated databases, to arrive at a set of actionable insights.

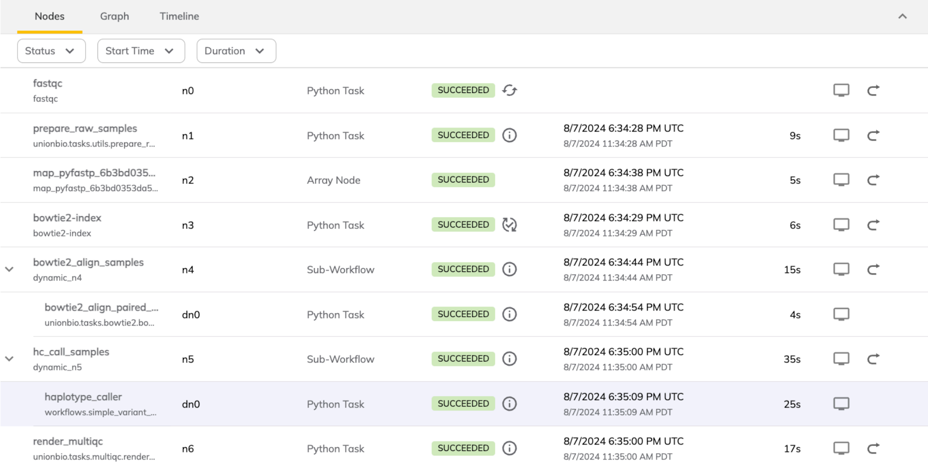
## 3. Results

A real world variant discovery workflow demonstrates how to tie all these disparate parts together. Starting with a directory containing raw FastQ files, we'll perform quality-control (QC), filtering, index generation, alignment, calling, and conclude with a final report of all the steps. Here's the code:

```python
from flytekit import workflow, dynamic, map_task
from flytekit.types.directory import FlyteDirectory
from flytekit.types.file import FlyteFile
from unionbio.tasks.utils import prepare_raw_samples
from unionbio.tasks.fastqc import fastqc
from unionbio.tasks.fastp import pyfastp
from unionbio.tasks.bowtie2 import bowtie2_idx, bowtie2_align_samples
from unionbio.tasks.callers import hc_call_variants
from unionbio.tasks.multiqc import render_multiqc


@workflow
def variant_discovery_wf(seq_dir: FlyteDirectory, ref_path: FlyteFile) -> FlyteFile:

    # Generate FastQC reports and check for failures
    fqc_out = fastqc(seq_dir=seq_dir)
    samples = prepare_raw_samples(seq_dir=seq_dir)

    # Map out filtering across all samples and generate indices
    filtered_samples = map_task(pyfastp)(rs=samples)

    # Explicitly define task dependencies
    fqc_out >> filtered_samples

    # Generate a bowtie2 index or load it from cache
    bowtie2_idx = bowtie2_index(ref=ref_path)

    # Generate alignments using bowtie2
    sams = bowtie2_align_samples(idx=bowtie2_idx, samples=filtered_samples)

    # Call variants
    vcfs = hc_call_variants(ref=ref_path, als=sams)

    # Generate final multiqc report with stats from all steps
    return render_multiqc(fqc=fqc_out, filt_reps=filtered_samples, sams=sams, vcfs=vcfs)
```

To help make sense of the flow of tasks, here is a screenshot from the Flyte UI that offers a visual representation of the different steps:

FastQC [8], an extremely common QC tool, is wrapped in a ShellTask and starts off the workflow by generating a report for all FastQ formatted reads files in a given directory. That directory is then turned into Reads objects via the `prepare_raw_samples` task. Those samples are passed to `fastp` for adapter removal and filtering of duplicate or low quality reads. FastP [9] is wrapped in a Python task which accepts a single Reads object. This task is then used in a map task to parallelize the processing of however many discrete samples were present in



| | | | | | |
|---|---|---|---|---|---|
| fastqc<br>fastqc | n0 | Python Task | SUCCEEDED | | |
| prepare_raw_samples<br>unionbio.tasks.utils.prepare_r... | n1 | Python Task | SUCCEEDED | 8/7/2024 6:34:28 PM UTC<br>8/7/2024 11:34:28 AM PDT | 9s |
| map_pyfastp_6b3bd035...<br>map_pyfastp_6b3bd0353da5... | n2 | Array Node | SUCCEEDED | 8/7/2024 6:34:38 PM UTC<br>8/7/2024 11:34:38 AM PDT | 5s |
| bowtie2-index<br>bowtie2-index | n3 | Python Task | SUCCEEDED | 8/7/2024 6:34:29 PM UTC<br>8/7/2024 11:34:29 AM PDT | 6s |
| bowtie2_align_samples<br>dynamic_n4 | n4 | Sub-Workflow | SUCCEEDED | 8/7/2024 6:34:44 PM UTC<br>8/7/2024 11:34:44 AM PDT | 15s |
| bowtie2_align_paired_...<br>unionbio.tasks.bowtie2.bo... | dn0 | Python Task | SUCCEEDED | 8/7/2024 6:34:54 PM UTC<br>8/7/2024 11:34:54 AM PDT | 4s |
| hc_call_samples<br>dynamic_n5 | n5 | Sub-Workflow | SUCCEEDED | 8/7/2024 6:35:00 PM UTC<br>8/7/2024 11:35:00 AM PDT | 35s |
| haplotype_caller<br>workflows.simple_variant_... | dn0 | Python Task | SUCCEEDED | 8/7/2024 6:35:09 PM UTC<br>8/7/2024 11:35:09 AM PDT | 25s |
| render_multiqc<br>unionbio.tasks.multiqc.render... | n6 | Python Task | SUCCEEDED | 8/7/2024 6:35:00 PM UTC<br>8/7/2024 11:35:00 AM PDT | 17s |

**Figure 1**. *Table listing the various tasks of the workflow alongside task type, status, completion time, and runtime*

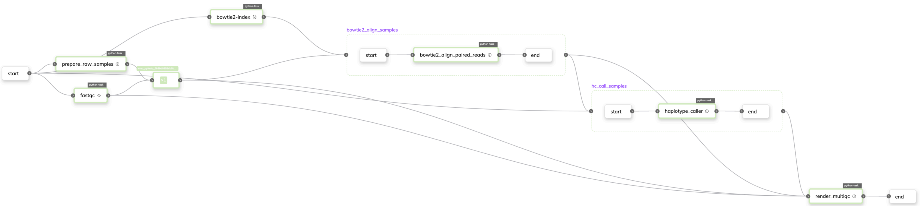


**Figure 2**. *Workflow DAG showing the tasks as color-coded nodes with connections between them representing dependencies*

the input directory. Flyte relies on the flow of strongly-typed inputs and outputs to assemble the workflow; since there is no implicit dependency between filtering and QC, we make this relationship explicit with the “>>” operator.

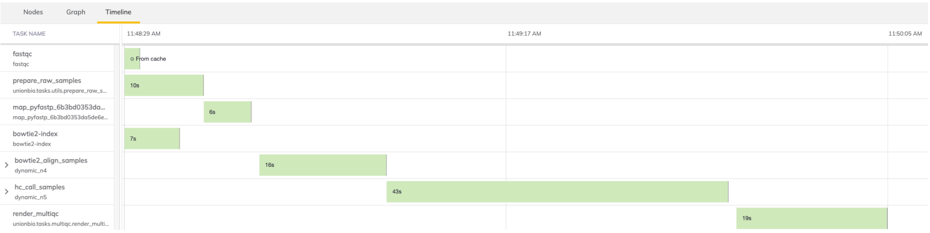
Once pre-processing is complete, alignment can take place. First, `bowtie2_index` generates an index if one is not already cached. Since the bowtie2 alignment task processes samples one at a time, it was wrapped in a dynamic workflow to process a list of inputs. Dynamics are another parallelism construct, similar to map tasks with some key differences: they are more flexible than map tasks at the expense of some efficiency. Similarly, variant calling is performed across all samples using HaplotypeCaller. Lastly, MultiQC [3], produces a final report of all the different steps in the workflow. Certain task definitions are omitted for the sake of cogency, they are all fully-defined in the `unionbio` repo.

Despite being a fairly parsimonious workflow, it’s important to highlight how many different languages are seamlessly integrated. The preprocessing tools are written in **Java** and **C/C++**. Alignment is carried out with a mix of **Perl** and **C++**. HaplotypeCaller is implemented purely in **Java**. Finally, the reporting tool is implemented in **Python**. While this simplicity affords easy understanding of task-flow from the code, the Flyte console provides excellent visualizations to best understand it’s structure:

Finally, it’s helpful to inspect a timeline of the execution which highlights a few things. Since this workflow was run several times over the course of capturing these figures, the `fastqc` task was cached in previous runs. It’s also clear from this figure which tasks were run in parallel in contrast to those which had dependencies on upstream outputs. Finally, the overall runtime is broken down into it’s separate parts. Since this was run on test data, everything executed fairly quickly.

## 4. Conclusion

Different steps in a bioinformatics pipeline often require tools with significantly different characteristics. As such, different languages are employed where their strengths are best leveraged. Regardless of which language or framework is used, ImageSpec captures those dependencies in an OCI-compliant image for use in Flyte workflows and beyond in a very ergonomic way. Defining dataclasses with FlyteFiles and additional metadata frees the data flow from the trappings of a traditional filesystem, while Flyte handles serialization so we



TASK NAME | 11:48:29 AM | 11:49:17 AM | 11:50:05 AM

fastqc
fastqc
From cache

prepare_raw_samples
unionbio.tasks.utils.prepare_raw_s...
10s

map_pyfastp_6b3bd0353da...
map_pyfastp_6b3bd0353da5de6e...
6s

bowtie2-index
bowtie2-index
7s

› bowtie2_align_samples
dynamic_n4
16s

› hc_call_samples
dynamic_n5
43s

render_multiqc
unionbio.tasks.multiqc.render_multi...
19s

**Figure 3**. *Execution timeline listing individual task runtimes in context of overall workflow runtime* can easily operate in a cloud native paradigm. With dependencies handled in a robust way and the data interface standardized, wrapping arbitrary tools in Flyte tasks produces reusable and composable components that behave predictably. Tying all of this into a common orchestration layer presents an enormous benefit to the developer experience and consequently the reproducibility and extensibility of the research project as a whole.

## References

[1] J. C. Venter *et al.*, “The Sequence of the Human Genome,” *Science*, vol. 291, no. 5507, pp. 1304–1351, 2001, doi: 10.1126/science.1058040.

[2] G. A. Van der Auwera *et al.*, “From FastQ Data to HighConfidence Variant Calls: The Genome Analysis Toolkit Best Practices Pipeline,” *Current Protocols in Bioinformatics*, vol. 43, no. 1, 2013, doi: 10.1002/0471250953.bi1110s43.

[3] P. Ewels, M̊. Magnusson, S. Lundin, and M. Käller, “MultiQC: summarize analysis results for multiple tools and samples in a single report,” *Bioinformatics*, vol. 32, no. 19, pp. 3047–3048, 2016, doi: 10.1093/bioinformatics/btw354.

[4] M. Baker, “1,500 scientists lift the lid on reproducibility,” *Nature*, vol. 533, no. 7604, pp. 452–454, 2016, doi: 10.1038/533452a.

[5] J. Armstrong, I. T. Fiddes, M. Diekhans, and B. Paten, “Whole-Genome Alignment and Comparative Annotation,” *Annual Review of Animal Biosciences*, vol. 7, no. 1, pp. 41–64, 2019, doi: 10.1146/annurev-animal-020518-115005.

[6] J. C. Venter, M. D. Adams, G. G. Sutton, A. R. Kerlavage, H. O. Smith, and M. Hunkapiller, “Shotgun Sequencing of the Human Genome,” *Science*, vol. 280, no. 5369, pp. 1540–1542, 1998, doi: 10.1126/science.280.5369.1540.

[7] B. Langmead, C. Trapnell, M. Pop, and S. L. Salzberg, “Ultrafast and memory-efficient alignment of short DNA sequences to the human genome,” *Genome Biology*, vol. 10, no. 3, 2009, doi: 10.1186/gb-2009-10-3-r25.

[8] S. Andrews, F. Krueger, A. Segonds-Pichon, L. Biggins, C. Krueger, and S. Wingett, “FastQC.” Babraham Institute, Babraham, UK, 2012.

[9] S. Chen, “Ultrafast onepass FASTQ data preprocessing, quality control, and deduplication using fastp,” *iMeta*, vol. 2, no. 2, 2023, doi: 10.1002/imt2.107.

Open Access | https://doi.org/10.25080/DXWY3560

SciPy 2024

July 8 - July 14, 2024

Proceedings of the 23nd
Python in Science Conference
ISSN: 2575-9752

# RoughPy

## Streaming data is rarely smooth

**Sam Morley**[1], and **Terry Lyons**[1]

[1]University of Oxford

### Abstract

Rough path theory is a branch of mathematics arising out of stochastic analysis. One of the main tools of rough path analysis is the signature, which captures the evolution of an unparametrised path including the order in which events occur. This turns out to be a useful tool in data science applications involving sequential data. RoughPy is our new Python package that aims change the way we think about sequential streamed data, by viewing it through the lens of rough paths. In RoughPy, data is wrapped in a stream object which can be composed and queried to obtain signatures that can be used in analysis. It also provides a platform for further exploration of the connections between rough path theory and data science.

**Keywords** sequential data, unparametrised paths, time series, rough paths, signatures, data science, machine learning, signature kernels, Log-ODE method

## 1. Introduction

Sequential data appears everywhere in the modern world: text, finance, health records, radio (and other electromagnetic spectra), sound (and speech), etc. Traditionally, these data are tricky to work with because of the exponential complexity and different scales of the underlying process. Until recently, with the development of transformers and large language models, it has been difficult to capture the long-term pattern whilst also capturing the short-term fine detail. Rough path theory gives us tools to work with sequential, ordered data in a mathematically rigorous way, which should provide a means to overcome some of the inherent complexity of the data. In this paper, we introduce a new package *RoughPy* for working with sequential data through the lens of rough path theory, where we can perform rigourous analyses and explore different ways to understand sequential data.

Rough paths arise in the study of *controlled differential equations* (CDEs), which generalise ordinary differential equations (ODEs) and stochastic differential equations [1], [2]. These are equations of the form $\mathrm{d}Y_t = f(Y_t, \mathrm{d}X_t)$, subject to an initial condition $Y_0 = y_0$, that model a non-linear system driven by a input path $X$. One simple CDE turns out to be critical to the theory:

$$\mathrm{d}S_t = S_t \otimes \mathrm{d}X_t \qquad S_0 = \mathbf{1}. \tag{1}$$

The solution of this equation is called the *signature* of $X$. It is analogous to the exponential function for ODEs, in that the solution of any CDE can be expressed in terms of the signature of the driving path. When the path $X$ is sufficiently regular, the signature can be computed directly as a sequence of iterated integrals. In other cases, we can still solve CDEs if we are given higher order data that can be used in place of the iterated integrals. A path equipped with this higher order data is called a *rough path*.

The signature turns out to be a useful summary of sequential data. It captures the order of events but not the necessarily the rate at which these events occur. The signature is robust to irregular sampling and provides a fixed-size view of the data, regardless of how many

**Published** Jul 10, 2024

**Correspondence to**
Sam Morley
sam.morley@maths.ox.ac.uk

observations are used to compute it. This means the signature can be a useful feature map to be used in machine learning for sequential data. There are numerous examples of using signatures of analysing sequential data outlined in Section 2.

Besides signatures, there are two other rough path-based methods that have found their way into data science in recent years. These are the signature kernel and neural CDEs. Both of these enjoy the same robustness of the signature, and expand the range of applications of rough path-based methods. We give a short overview of these methods in Section 1.2.

There are several Python packages for computing signatures of sequential data, including `esig` [3], `iisignature` [4], and `signatory` [5]. These packages provide functions for computing signatures from raw, structured data presented in an $n \times d$ array, where $d$ is the dimension of the stream and $n$ is the number of samples. This means the user is responsible for interpreting the data as a path and arranging the computations that need to be done.

RoughPy is a new package for working with sequential data and rough paths. The design philosophy for this package is to shift the emphasis from simply computing signatures on data to instead work with streams. A *stream* is a view of some data as if it were a rough path, that can be queried over intervals to obtain a signature. The actual form of the data is abstracted away in favour of stream objects that closely resemble the mathematics. The aim is to change the way that users think about sequential data and advance the understanding of path-like data analysis.

On top of the streams, RoughPy also provides concrete implementations for elements of the various algebras associated with rough path analysis. These include free tensor algebras, shuffle tensor algebras, and Lie algebras (Section 1.1). This allows the user to easily manipulate signatures, and other objects, in a more natural manner. This allows us to quickly develop methods by following the mathematics.

The paper is organised as follows. In the remainder of this section, we give a brief overview of the mathematics associated with rough path theory, and provide some additional detail for the signature kernel and neural CDEs. In Section 2 we list several recent applications of signatures and rough path-based methods in data science applications. These applications should serve to motivate the development of RoughPy. Finally, in Section 3 we give a more detailed overview of the RoughPy library, the types and functions it contains, and give an example of how it can be used.

RoughPy is open source (BSD 3-Clause) and available on GitHub https://github.com/datasig-ac-uk/roughpy.

### *1.1. Mathematical background*

In this section we give a very short introduction to signatures and rough path theory that should be sufficient to inform the discussion in the sequel. For a far more comprehensive and rigorous treatment, we refer the reader to the recent survey [6]. For the remainder of the paper, we write $V$ for the vector space $\mathbb{R}^d$, where $d \geq 1$.

A *path* in $V$ is a continuous function $X : [a, b] \to V$, where $a < b$ are real numbers. For the purposes of this discussion, we shall further impose the condition that all paths are of bounded variation. The value of a path $X$ at some parameter $t \in [a, b]$ is denoted $X_t$.

The signature of $X$ is an element of the *(free) tensor algebra*. For $n \geq 0$, the $n$th tensor power of $V$ is defined recursively by $V^{\otimes 0} = \mathbb{R}$, $V^{\otimes 1} = V$, and $V^{\otimes n+1} = V \otimes V^{\otimes n}$ for $n > 1$. For example, $V^{\otimes 2}$ is the space of $d \times d$ matrices, and $V^{\otimes 3}$ is the space of $d \times d \times d$ tensors. The *tensor algebra over* $V$ is the space

$$\mathrm{T}((V)) = \left\{\boldsymbol{x} = (x_0, x_1, \ldots) : x_j \in V^{\otimes j}\, \forall j \geq 0\right\} \tag{2}$$

equipped with the tensor product $\otimes$ as multiplication. The tensor algebra is a *Hopf algebra*, and comes equipped with an antipode operation $\alpha_V : \mathrm{T}((V)) \to \mathrm{T}((V))$. It contains a group $\mathrm{G}(V)$ of elements under tensor multiplication and the antipode. The members of $\mathrm{G}(V)$ are called *group-like* elements. For each $n \geq 0$, we write $\mathrm{T}^n(V)$ for the *truncated tensor algebra* of degree $n$, which is the space of all $\boldsymbol{x} = (x_0, x_1, \ldots)$ such that $x_j = 0$ whenever $j > n$. Similarly, we write $\mathrm{T}^{>n}((V))$ for the subspace of elements $\boldsymbol{x} = (x_0, x_1, \ldots)$ where $x_j = 0$ whenever $j \leq n$, which is an ideal in $\mathrm{T}((V))$ and $\mathrm{T}^n(V) = \mathrm{T}((V))/\mathrm{T}^{>n}((V))$. The truncated tensor algebra is an algebra, when given the *truncated tensor product*, obtained by truncating the full tensor product.

The signature $\mathrm{S}(X)_{s,t}$ of a path $X : [a,b] \to V$ over a subinterval $[s,t) \subseteq [a,b]$ is $\mathrm{S}(X)_{s,t} = \left(1, \mathrm{S}_1(X)_{s,t}, \ldots\right) \in \mathrm{G}(V)$ where for each $m \geq 1$, $\mathrm{S}_m(X)_{s,t}$ is given by the iterated (Riemann-Stieltjes) integral

$$\mathrm{S}_m(X)_{s,t} = \underset{s<u_1<u_2<\ldots<u_m<t}{\int \ldots \int} \mathrm{d}X_{u_1} \otimes \mathrm{d}X_{u_2} \otimes \ldots \otimes \mathrm{d}X_{u_m}. \tag{3}$$

The signature respects concatenation of paths, meaning $\mathrm{S}(X)_{s,t} = \mathrm{S}(X)_{s,u} \otimes \mathrm{S}(X)_{u,t}$ for any $s < u < t$. This property is usually called *Chen's relation*. Two paths have the same signature if and only if they differ by a *tree-like path* [7]. The signature is translation invariant, and it is invariant under reparametrisation.

The *dual* of $\mathrm{T}((V))$ is the *shuffle algebra* $\mathrm{Sh}(V)$. This is the space of linear functionals $\mathrm{T}((V)) \to \mathbb{R}$ and consists of sequences $(\lambda_0, \lambda_1, \ldots)$ with $\lambda_k \in (V^*)^{\otimes k}$ and where $\lambda_k = 0$ for all $k$ larger than some $N$. (Here $V^*$ denotes the dual space of $V$. In our notation $V^* \cong V$.) The multiplication on $\mathrm{Sh}(V)$ is the *shuffle product*, which corresponds to point-wise multiplication of functions on the path. Continuous functions on the path can be approximated (uniformly) by shuffle tensors acting on $\mathrm{G}(V)$ on the signature. This is a consequence of the Stone-Weierstrass theorem. This property is sometimes referred to as *universal nonlineararity*.

There are several *Lie algebras* associated to $\mathrm{T}((V))$. Define a *Lie bracket* on $\mathrm{T}((V))$ by the formula $[\boldsymbol{x}, \boldsymbol{y}] = \boldsymbol{x} \otimes \boldsymbol{y} - \boldsymbol{y} \otimes \boldsymbol{x}$, for $\boldsymbol{x}, \boldsymbol{y} \in \mathrm{T}((V))$. We define subspaces $L_m$ of $\mathrm{T}((V))$ for each $m \geq 0$ inductively as follows: $L_0 = \{\boldsymbol{0}\}$, $L_1 = V$, and, for $m \geq 1$,

$$L_{m+1} = \mathrm{span}\{[\boldsymbol{x}, \boldsymbol{y}] : \boldsymbol{x} \in V, \boldsymbol{y} \in L_m\}. \tag{4}$$

The space of formal Lie series $\mathcal{L}(V)$ over $V$ is the subspace of $\mathrm{T}((V))$ containing sequences of the form $(\ell_0, \ell_1, \cdots)$, where $\ell_j \in L_j$ for each $j \geq 0$. Note that $\mathcal{L}(V) \subseteq \mathrm{T}^{>0}(V)$. For any $\boldsymbol{x} \in \mathrm{T}(V)$ we define

$$\exp(\boldsymbol{x}) = \sum_{n=0}^{\infty} \frac{\boldsymbol{x}^{\otimes n}}{n!} \quad \text{and} \quad \log(\boldsymbol{1} + \boldsymbol{x}) = \sum_{n=1}^{\infty} \frac{(-1)^{n-1}}{n} \boldsymbol{x}^{\otimes n}. \tag{5}$$

For any path $X$, we have $\mathrm{LogSig}(X)_{s,t} := \log\left(\mathrm{S}(X)_{s,t}\right) \in \mathcal{L}(V)$, and we call this the *log-signature* of $X$ over $[s,t)$. This is an alternative representation of the path, but doesn't enjoy the same unievrsal non-linearity of the signature.

### *1.2. Rough paths in data science*

Now we turn to the applications of rough path theory to data science. Our first task is to form a bridge between sequential data and paths. Consider a finite, ordered sequence $\{(t_1, \boldsymbol{x}_1, \ldots, t_N, \boldsymbol{x}_N)\}$ of observations, where $t_j \in \mathbb{R}$, and $\boldsymbol{x}_j \in V$. (More generally, we might consider $\boldsymbol{x}_j \in \mathcal{L}(V)$ instead. That is, data that already contains higher-order information. In our language, it is a genuine rough path.) We can find numerous paths that interpolate these observations; a path $X : [t_0, t_N] \to V$ such that, for each $j$, $X_{t_j} = \boldsymbol{x}_j$. The simplest interpolation is to take the path that is linear between adjacent observations.

Once we have a path, we need to be able to compute signatures. For practical purposes, we truncate all signatures (and log-signatures) to a particular degree $M$, which we typically call the *depth*. The dimension of the ambient space $d$ is usually called the *width*. Using linear interpolation, we can compute the iterated integrals explicitly using a free tensor exponential of the difference of successive terms:

$$\mathrm{Sig}^M\big([t_j, t_{j+1})\big) = \exp_M\big(\boldsymbol{x}_{j+1} - \boldsymbol{x}_j\big) := \sum_{j=0}^{M} \frac{1}{j!}\big(\boldsymbol{x}_{j+1} - \boldsymbol{x}_j\big)^{\otimes j}. \tag{6}$$

Here, and in the remainder of the paper, we shall denote the empirical signature over an interval $I$ by $\mathrm{Sig}(I)$ and the log-signature as $\mathrm{LogSig}(I)$. We can compute the signature over arbitrary intervals by taking the product of the these terms, using the multiplicative property of the signature.

### 1.2.1. *The signature transform:*

Most of the early applications of rough paths in data science, the (truncated) signature was used as a feature map [8]. This provides a summary of the path that is independent of the parameterisation and the number of observations. Unfortunately, the signature grows geometrically with truncation depth. If $d > 1$, then the dimension of $\mathrm{T}^M(V)$ is

$$\sum_{m=0}^{M} d^m = \frac{d^{M+1} - 1}{d - 1} \tag{7}$$

The size of the signature is a reflection of the complexity of the data, where data with a higher complexity generally needs a higher truncation level and thus a larger signature. It is worth noting that this still represents a significant compression of stream information in many cases.

For some applications, it might be possible to replace the signature with the log-signature. The log-signature is smaller than the signature, but we lose the universal non-linearity property of the signature. Alternatively, we might turn to other techniques that don't require a full calculation of the signature (such as the signature kernel below). As the connection between rough paths and data science becomes more mathematically mature, we will likely find new ways to use the signature without requiring its full size.

### 1.2.2. *Signature kernels:*

Kernel methods are useful tools for learning with sequential data. Mathematically, a *kernel* on a set $W$ is a positive-definite function $k : W \times W \to \mathbb{R}$. Kernels are often quite easy to evaluate because of the kernel trick, which involves embedding the data in a inner product space, with a feature map, in which the kernel can be evaluated by simply taking an inner product. Informally, kernels measure the similarity between two points. They are used in a variety of machine learning tasks such as classification.

The *signature kernel* is a kernel induced on the space of paths by combining the signature with an inner product defined on the tensor algebra [9]. The theory surrounding the signature kernel has been expanded several times since their introduction [10], [11]. Typically, the inner product on $\mathrm{T}((V))$ will itself by derived from an inner product on $V$, extended to the tensor algebra.

Signatures are infinite objects, so we can't simply evaluate inner products on the tensor algebra. Fortunately, we can approximate the signature kernel by taking inner products of truncated signatures. Even better, it turns out that, in certain cases, the signature kernel can be realised as the solution to a partial differential equation (PDE) of Goursat type. This means the full signature kernel can be computed from raw data without needing to compute full signatures [12].

In fact, in recent preprint, it has been shown that there are higher order solvers for signature kernels by rewriting the kernel solution of a system of PDEs of Goursat type [13]. A critical part of their method involves the adjoint of both left and right free tensor multiplication, which are not available in any current package for computing signatures. These functions are provided by RoughPy.

#### 1.2.3. *Neural controlled differential equations:*

Neural CDEs are a method for modelling irregular time series. We consider CDEs of the form

$$\mathrm{d}Y_t = f_\theta(Y_t)\,\mathrm{d}X_t \tag{8}$$

where $f_\theta$ is a neural network. We can treat the path $Y$ as “hidden state” that we can tune using data to understand the relationship between the driving path $X_t$ and some response. Neural CDEs can be regarded as a continuous-time analogue of a recurrent neural network [14].

Neural CDEs initially showed some promising results on several benchmarks but now lag behind current state-of-the-art approaches to time series modelling. The latest iteration of neural CDEs are the recently introduced Log-neural controlled differential equations [15], which make use of the *Log-ODE* method for solving rough differential equations in order to boost the performance of neural CDEs.

## 2. Current applications of rough paths

In this section we enumerate several applications where rough paths have been used to develop or improve methods. This list presented here is certainly not exhaustive. In addition to the literature cited below, there are numerous additional references and worked examples, in the form of Jupyter notebooks, available on the DataSig website (https://datasig.ac.uk/examples).

### *2.1. Detecting interference in radio astronomy data*

Radio frequency interference (RFI) is a substantial problem in the field of radio astronomy. Even small amounts of RFI can obscure the faint signals generated by distant stellar objects and events. The problem of identifying RFI in a signal falls into a class of semi-supervised learning tasks called *novelty* (or *anomaly*) *detection*. Rough path methods have been applied to develop a novelty detection framework based on rough path methods to detect RFI in radio astronomy data from several radio telescopes [16]. Their result show that their framework is effective at detecting even faint RFI within the test data. This work is based on a general novelty detection framework [17].

Signatures kernels have also been used for a similar problem of detecting malware by inspecting the streaming tree of processes on a computer system T. Cochrane, P. Foster, V. Chhabra, M. Lemercier, T. Lyons, and C. Salvi [18]. Their method uses a support vector machine classifier to identify processes that are malicious compared to “normal” behaviour learned via training on a corpus of normality.

### *2.2. Tracking mood via natural language processing*

One application of rough paths in natural language processing has been in the domain of mental health [19], [20]. In this work, the authors present a model for identifying changes in a person’s mood based on their online textual content. Many mental health conditions have symptoms that manifest in the (textual) expression, so this could be a powerful tool for mental health professionals to identify changes in patients and intervene before the state develops. Their model achieves state-of-the-art performance vs existing models on two datasets.

### 2.3. *Predicting battery cell degradation*

Another recent application of signatures is to predict the degradation of lithium-ion cells [21]. They use signature features to train a model that can accurately predict the end of life of a cell using relatively low-frequency sampling compared to existing models. They also observed that the performance at higher frequency was comparable to other models.

### 2.4. *Prediction of sepsis in intensive care data*

One of the first effective demonstrations of the utility of signatures and rough paths based methods in healthcare was in the 2019 PhysioNet challenge [22]. In this contest, teams were invited to develop models to predict sepsis in patients from intensive care unit data. In this challenge, a team utilising signatures to enhance predictive power placed first in the official phase of the challenge. Since then, signatures and other rough path based approaches have been used in several other clinical contexts [20], [23], [24]. Clinical data is often irregularly sampled and often exhibits a high degree of missingness, but it can also be very high-frequency and dense. Rough path based methods can handle these data in an elegant way, and retain the structure of long and short term dependencies within the data.

### 2.5. *Human action recognition*

The task of identifying a specific action performed by a person from a short video clip is very challenging. Signatures derived from landmark data extracted from the video has been used to train classification models that achieved state-of-the-art performance compared with contemporary models [25], [26], [27]. (See also preprint papers [28], [29].) Also in the domain of computer vision, signatures have been used to produce lightweight models for image classification [30] and in handwriting recognition tasks [31].

## 3. RoughPy

RoughPy is a new library that aims to support the development of connections between rough path theory and data science. It represents a shift in philosophy from simple computations of signatures for sequential data, to a representation of these data as a rough path. The design objectives for RoughPy are as follows:

1. provide a class that presents a rough path view of some source of data as a rough path, exposing methods for querying the data over intervals to get a signature or log-signature;
2. provide classes and functions that allow the users to interact with the signatures and other algebraic objects in a natural, mathematical manner;
3. all operations should be differentiable and objects should be interoperable with objects from machine learning, such as TensorFlow (JAX) and PyTorch.

The first two objectives are simple design and implementation problems. The final objective presents the most difficulty, especially interoperability between RoughPy and common machine learning libraries. There are array interchange formats for NumPy-like arrays, such as the Python Array API standard [32] and the DLPack protocol [33]. These provide part of the picture, but in order for them to be fully supported, RoughPy must support a variety of compute backends such as CUDA (NVidia), ROCm/HIP (AMD), and Metal (Apple).

RoughPy is a substantial library with numerous components, mostly written in C++ with a Python interface defined using Pybind11 [34]. The original design of the library closely followed the C++ template libraries libRDE and libalgebra [35], although it has seen many iterations since.

In the remainder of this section, we discuss some of the core components of RoughPy, give an example of using RoughPy, and discuss the future of RoughPy.

### *3.1. Free tensors, shuffle tensors, and Lie objects*

In order to properly support rough path based methods and allow users to write code based on mathematical concepts, we provide realisations of several algebra types. The algebras provided in RoughPy are `FreeTensor`, `ShuffleTensor`, and `Lie`, which define elements of a particular free tensor algebra, shuffle tensor algebra, and Lie algebra respectively. Each of these algebras is initialized with a width, depth, and scalar coefficient type, encapsulated in a `Context` object.

In addition to the algebra classes, RoughPy provides a number of supporting functions, including antipodes and half-shuffle products for `FreeTensor`/`ShuffleTensor` objects, and adjoint operators for left free tensor multiplication. These are operations that are frequently used in the theory of rough paths, and will likely be necessary in developing new applications later (as in the signature kernels).

RoughPy algebras are designed around a flexible scalar ring system that allows users to perform calculations with different accuracy, or derive expressions by using polynomial coefficients. For most applications, single or double precision floating point numbers will provide a good balance between performance and accuracy. (Double precision floats are the default.) When more precision is required, rational coefficients can be used instead. These are backed by GMP rationals for fast, arbitrary precision rational arithmetic [36]. Polynomial coefficients can be used to derive formulae by performing calculations. This is a powerful technique for understanding the terms that appear in the result, particularly whilst testing and debugging.

### *3.2. Intervals*

RoughPy is very careful in the way it handles intervals. All intervals in RoughPy are half-open, meaning that they include one end point but not the other; they are either *clopen* $[a, b) := \{t : a \leq t < b\}$ or *opencl* $(a, b] := \{t : a < t \leq b\}$. Besides the type (clopen or opencl), all intervals must provide methods for retrieving the infimum ($a$ in the above notation) and the supremum ($b$ above) of the interval as double precision floats. This is enforced by means of an abstract base class `Interval`. The main concrete interval types are `RealInterval`, an interval with arbitrary real endpoints, and `DyadicInterval`, as described below. For brevity, we shall only consider clopen intervals.

A *dyadic interval* is an interval $D_k^n := [k/2^n, (k+1)/2^n)$, where $k, n$ are integers. The number $n$ is often described as the *resolution* of the interval. The family of dyadic intervals of a fixed resolution $n$ partition the real line so that every real number $t$ belongs to a unique dyadic interval $D_n^k$. Moreover, the family of all dyadic intervals have the property that two dyadic intervals are either disjoint or one contains the other (including the possibility that they are equal).

In many cases, RoughPy will granularise an interval into a dyadic intervals. The *dyadic granularisation* of $[a, b)$ with resolution $n$ is $[k_1/2^n, k_2/2^n)$ where $k_1 = \max\{k : k/2^n \leq a\}$ and $k_2 = \max\{k : k/2^n \leq b\}$. In effect, the dyadic granularisation is the result of “rounding” each end point to the included end of the unique dyadic interval that contain it.

### 3.3. Streams

Streams are central to RoughPy. A RoughPy `Stream` is a rough path view of some underlying data. It provides two key methods to query the object over intervals to retrieve either a signature or log-signature. Importantly, once constructed, the underlying data is inaccessible except by querying via these methods. `Stream`s are designed to be composed in various ways, such as by concatenation, in order to build up more complex streams. A `Stream` is actually a (type-erasing) wrapper around a more minimal `StreamInterface` abstract class.

We construct streams by a factory function associated with each different `StreamInterface`, which might perform some compression of the underlying data. For example, a basic `StreamInterface` is the `LieIncrementStream`, which can be constructed using the associated `from_increments` factory function (a static method of the class), which accepts an $n \times d$ array of *increment data*. These data will typically be the differences between successive values of the data (but could also include higher-order Lie terms). This is similar to the way that libraries such as `esig`, `iisignature`, and `signatory` consume data.

RoughPy streams cache the result of log-signature queries over dyadic intervals so they can be reused in later calculations. To compute the log-signature over any interval $I$, we granularise at a fixed stream resolution $n$ to obtain the interval $\tilde{I} = [k_1/2^n, k_2/2^n)$, and then compute

$$\mathrm{LogSig}(\tilde{I}) = \log\left(\prod_{k=k_1}^{k_2-1} \exp(\mathrm{LogSig}(D_k^n))\right). \tag{9}$$

The $\mathrm{LogSig}(D_k^n)$ terms on the right-hand-side are either retrieved from the cache, or computed from the underlying source. This is essentially the Campbell-Baker-Hausdorff formula applied to the log-signatures at the finest level. In practice, we can actually reduce the number of terms in the product, by merging complementary dyadic intervals that appear in the granularisation. We further optimise by using a fused multiply-exponential ($A\exp(B)$) operation.

Signatures are always computed by first computing the log-signature and then exponentiating. Directly computing the signature as a product of exponentials of (cached) log-signatures might accumulate enough numerical errors to drift slightly from a group-like tensor. That is, the result might not actually be a true signature. Taking the logarithm and then exponentiating back to obtain the signature has the effect of correcting this numerical drift from a true signature.

Aside from the basic `LieIncrementStream`, there are several other implementations of the `StreamInterface` currently available in RoughPy. The `BrownianStream` approximates Brownian motion by generating normal distributed increments over dyadic intervals of arbitrary resolution on demand, forming a reasonable approximation of true Brownian motion. The `ExternalDataStream` is an interface for loading data from various external sources, such as from a database or specialised data format. Currently, only sound files are supported but we plan to extend support for other sources as the need arises. This will certainly include "online" data sources such as computer peripheral devices (e.g. microphones).

The other main `StreamInterface` implementation is the `PiecewiseAbelianStream`, which is an important construction from CDE. A piecewise Abelian path, or log-linear path, is an example of a *smooth rough path*, which generalises piecewise linear approximations of an arbitrary stream. Formally, an *Abelian path* $Y$ is a pair $([a, b), \boldsymbol{y})$ where $a < b$ and $\boldsymbol{y} \in \mathcal{L}(V)$. The log-signature over an arbitrary interval $[u, v) \subseteq [a, b)$ is given by

$$\mathrm{LogSig}(Y)_{u,v} = \frac{v-u}{b-a}\boldsymbol{y}. \tag{10}$$

A *piecewise Abelian path* is the concatenation of finitely many Abelian paths with adjacent intervals. For any rough path $X$ and partition $\{a = t_0 < t_1 < ... < t_N = b\}$ there is a piecewise Abelian approximation for this path given by

$$\left\{\left(\left[t_{j-1}, t_j\right), \mathrm{LogSig}(X)_{t_{j-1},t_j}\right) : j = 1, ..., N\right\}. \tag{11}$$

This construction turns out to be vital for computing signature kernels [32] and for solving CDEs [2], [15]. In particular, this construction can be used to compress data at some degree, which can the be used in computations at a higher degree.

### *3.4. Example*

In this section we show a very simple example of how to use RoughPy to construct a stream and compute a signature. This example is similar to the first few steps of the tutorial found in the RoughPy documentation. RoughPy can be installed using `pip`, where prebuilt wheels are available for Windows, Linux, and MacOs:

```
pip install roughpy
```

We refer the reader to this documentation for much more detail. We will construct a stream in $\mathbb{R}^{26}$ by taking each letter in a word, “scipy” in this example, as the increments of a path:

```
import numpy as np

text = "scipy"
increments = np.zeros((5, 26), dtype="int8")
for i, c in enumerate(text):
    increments[i, ord(c) - 97] = 1
```

Now we import RoughPy and construct a `Stream` using the factory mentioned above. One other critical ingredient is the algebra `Context`, which is used to set up a consistent set of algebra objects with the desired width (26), truncation level (2), and coefficient type (`Rational`).

```
import roughpy as rp

ctx = rp.get_context(width=26, depth=2,
      coeffs=rp.Rational)
stream = rp.LieIncrementStream.from_increments(
         increments, ctx=ctx)
```

Now we can compute the signature of the stream over the whole domain of the stream $[0, 4]$ by omitting the interval argument:

```
sig = stream.signature()
print(sig)
# { 1() 1(3) 1(9) 1(16) 1(19) 1(25) 1/2(3,3)
#   1(3,9) 1(3,16) 1(3,25) 1/2(9,9) 1(9,16)
#   1(9,25) 1/2(16,16) 1(16,25) 1(19,3) 1(19,9)
#   1(19,16) 1/2(19,19) 1(19,25) 1/2(25,25) }
```

The first term of the signature is always 1, and the empty parentheses indicate the empty tensor word. The next five terms correspond to the counts of each unique letter that appears, the number in parentheses indicates the letter (with `a` being 1). The final terms indicate the order in which each pair of letters appear in the word. For instance, the term `1(3,9)` indicates that a `c` appears before an `i`.

**Remark 1**.

It turns out that most words in the English language can be distinguished using only their level 2 signatures. The first level signatures groups words into anagrams. The second level signature counts the occurrences of each ordered pair of letters. There are relatively few words that require level 3 data. From the standard Linux dictionary, containing around 80,000 words, there are two pairs of words that require the level 3 terms: “toot” and “otto”, and “naan” and “anna”. This is shown in the RoughPy documentation. Similar patterns can be observed in other languages too, including French, Spanish, German, Russian, and Lithuanian.

This is only the beginning of the story. From here, we can use the signatures to compute the similarity between streams, via the signature kernel for instance, or used as features in a variety of machine learning problems. More detailed examples of how to use signatures in data science are given on the DataSig website https://datasig.ac.uk/examples.

### *3.5. The future of RoughPy*

RoughPy is continuously evolving. At time of writing, the current version uses libalgebra and libalgebra-lite (libalgebra with fewer templates) for computations. Unfortunately, this made it difficult to achieve the differentiability and computation device support that we want. We are currently changing the way we implement vectors and algebras to provide the support for on-device computation that we want. Making the operations differentiable is crucial for machine learning, and will be the biggest challenge.

Long term, we need to expand support for signature kernels and CDEs. As applications of these tools grow in data science, we will need to devise new methods for computing kernels, or solving CDEs. We will also build a framework for constructing and working with linear maps, and homomorphisms. For example, one very useful linear map is the extension of the $\log$ function to the whole tensor algebra.

## 4. Conclusions

The use of rough path theory in data science is rapidly expanding and provides a different way to view sequential data. Signatures, and other methods arising from rough path theory, are already used in a wide variety of applications, with great effect. The next steps in overcoming the difficulty in modeling sequential data will require a change of perspective. Viewing these data through the lens of rough path theory might provide this change.

RoughPy is a new Python library for working with streamed data using rough path methods. It is designed to abstract away the form and source of data so that analysis can be performed by querying path-like objects. This approach is much closer to the mathematics. It also allows users to interact with the various algebras associated with rough paths (free tensor algebra, shuffle tensor algebra, Lie algebra) in a natural way. RoughPy is under active development, and a long list of improvements and extensions are planned.

## References

[1] T. J. Lyons, “Differential equations driven by rough signals.,” *Revista Matemática Iberoamericana*, vol. 14, no. 2, pp. 215–310, 1998, [Online]. Available: http://eudml.org/doc/39555

[2] T. J. Lyons, M. Caruana, and T. Lévy, *Differential Equations Driven by Rough Paths:´Ecole d’Eté de Probabilités de Saint-Flour XXXIV - 2004*. Springer Berlin Heidelberg, 2007. doi: 10.1007/978-3-540-71285-5.

[3] T. Lyons and D. Maxwell, “esig.” 2017.

[4] J. F. Reizenstein and B. Graham, “Algorithm 1004: The Iisignature Library: Efficient Calculation of Iterated-Integral Signatures and Log Signatures,” *ACM Transactions on Mathematical Software*, vol. 46, no. 1, pp. 1–21, 2020, doi: 10.1145/3371237.

[5] P. Kidger and T. Lyons, “Signatory: differentiable computations of the signature and logsignature transforms, on both CPU and GPU.” [Online]. Available: https://arxiv.org/abs/2001.00706

[6] T. Lyons and A. D. McLeod, “Signature Methods in Machine Learning.” [Online]. Available: https://arxiv.org/abs/2206.14674

[7] B. Hambly and T. Lyons, “Uniqueness for the signature of a path of bounded variation and the reduced path group,” *Annals of Mathematics*, vol. 171, no. 1, pp. 109–167, 2010, doi: 10.4007/annals.2010.171.109.

[8] P. Kidger, P. Bonnier, I. Perez Arribas, C. Salvi, and T. Lyons, “Deep Signature Transforms,” in *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché Buc, E. Fox, and R. Garnett, Eds., 2019. doi: 10.48550/arXiv.1905.08494.

[9] F. J. Kiraly and H. Oberhauser, “Kernels for Sequentially Ordered Data,” *Journal of Machine Learning Research*, vol. 20, no. 31, pp. 1–45, 2019, doi: 10.48550/arXiv.2102.03657.

[10] A. Fermanian, P. Marion, J.-P. Vert, and G. Biau, “Framing RNN as a kernel method: A neural ODE approach,” in *Advances in Neural Information Processing Systems*, M. Ranzato, A. Beygelzimer, Y. Dauphin, P. Liang, and J. W. Vaughan, Eds., 2021, pp. 3121–3134. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2021/file/18a9042b3fc5b02fe3d57fea87d6992f-Paper.pdf

[11] T. Cass, T. Lyons, and X. Xu, “Weighted signature kernels,” *The Annals of Applied Probability*, vol. 34, no. 1A, 2024, doi: 10.1214/23-aap1973.

[12] C. Salvi, T. Cass, J. Foster, T. Lyons, and W. Yang, “The Signature Kernel Is the Solution of a Goursat PDE,” *SIAM Journal on Mathematics of Data Science*, vol. 3, no. 3, pp. 873–899, 2021, doi: 10.1137/20m1366794.

[13] M. Lemercier and T. Lyons, “A High Order Solver for Signature Kernels.” [Online]. Available: https://arxiv.org/abs/2404.02926

[14] P. Kidger, J. Morrill, J. Foster, and T. Lyons, “Neural Controlled Differential Equations for Irregular Time Series,” in *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, Eds., 2020, pp. 6696–6707. doi: 10.48550/arXiv.1906.08215.

[15] B. Walker, A. D. McLeod, T. Qin, Y. Cheng, H. Li, and T. Lyons, “Log Neural Controlled Differential Equations: The Lie Brackets Make a Difference,” 2024, doi: 10.48550/ARXIV.2402.18512.

[16] P. Arrubarrena, M. Lemercier, B. Nikolic, T. Lyons, and T. Cass, “Novelty Detection on Radio Astronomy Data using Signatures.” [Online]. Available: https://arxiv.org/abs/2402.14892

[17] Z. Shao, R. S.-Y. Chan, T. Cochrane, P. Foster, and T. Lyons, “Dimensionless Anomaly Detection on Multivariate Streams with Variance Norm and Path Signature.” [Online]. Available: https://arxiv.org/abs/2006.03487

[18] T. Cochrane, P. Foster, V. Chhabra, M. Lemercier, T. Lyons, and C. Salvi, “SK-Tree: a systematic malware detection algorithm on streaming trees via the signature kernel,” in *2021 IEEE International Conference on Cyber Security and Resilience (CSR)*, 2021, pp. 35–40. doi: 10.1109/csr51186.2021.9527933.

[19] T. Tseriotou, A. Tsakalidis, P. Foster, T. Lyons, and M. Liakata, “Sequential Path Signature Networks for Personalised Longitudinal Language Modeling,” in *Findings of the Association for Computational Linguistics: ACL 2023*, 2023, pp. 5016–5031. doi: 10.18653/v1/2023.findings-acl.310.

[20] T. Tseriotou *et al.*, “Sig-Networks Toolkit: Signature Networks for Longitudinal Language Modelling,” in *Proceedings of the 18th Conference of the European Chapter of the Association for Computational Linguistics: System Demonstrations*, N. Aletras and O. De Clercq, Eds., St. Julians, Malta, 2024, pp. 223–237. doi: 10.48550/arXiv.2312.03523.

[21] R. Ibraheem, Y. Wu, T. Lyons, and G. dos Reis, “Early prediction of Lithium-ion cell degradation trajectories using signatures of voltage curves up to 4-minute sub-sampling rates,” *Applied Energy*, vol. 352, p. 121974, 2023, doi: 10.1016/j.apenergy.2023.121974.

[22] J. H. Morrill, A. Kormilitzin, A. J. Nevado-Holgado, S. Swaminathan, S. D. Howison, and T. J. Lyons, “Utilization of the Signature Method to Identify the Early Onset of Sepsis From Multivariate Physiological Time Series in Critical Care Monitoring,” *Critical Care Medicine*, vol. 48, no. 10, pp. e976–e981, 2020, doi: 10.1097/ccm.0000000000004510.

[23] S. N. Cohen *et al.*, “Subtle variation in sepsis-III definitions markedly influences predictive performance within and across methods,” *Scientific Reports*, vol. 14, no. 1, 2024, doi: 10.1038/s41598-024-51989-6.

[24] G. Falcioni, A. Georgescu, E. Molimpakis, L. Gottlieb, T. Kuhn, and S. Goria, “Path Signature Representation of Patient-Clinician Interactions as a Predictor for Neuropsychological Tests Outcomes in Children: A Proof of Concept,” in *2023 IEEE International Conference on Medical Artificial Intelligence (MedAI)*, 2023, pp. 1–9. doi: 10.1109/medai59581.2023.00008.

[25] W. Yang, T. Lyons, H. Ni, C. Schmid, and L. Jin, “Developing the Path Signature Methodology and Its Application to Landmark- Based Human Action Recognition,” in *Stochastic Analysis, Filtering, and Stochastic Optimization*, Springer International Publishing, 2022, pp. 431–464. doi: 10.1007/978-3-030-98519-6_18.

[26] J. Cheng *et al.*, “Skeleton-Based Gesture Recognition With Learnable Paths and Signature Features,” *IEEE Transactions on Multimedia*, vol. 26, pp. 3951–3961, 2024, doi: 10.1109/tmm.2023.3318242.

[27] S. Liao, T. Lyons, W. Yang, K. Schlegel, and H. Ni, “Logsig-RNN: a novel network for robust and efficient skeleton-based action recognition,” 2021. doi: 10.48550/arXiv.2110.13008.

[28] M. R. Ibrahim and T. Lyons, “FaceTouch: Detecting hand-to-face touch with supervised contrastive learning to assist in tracing infectious disease,” 2023, doi: 10.48550/ARXIV.2308.12840.

[29] L. Jiang, W. Yang, X. Zhang, and H. Ni, “GCN-DevLSTM: Path Development for Skeleton-Based Action Recognition.” [Online]. Available: https://arxiv.org/abs/2403.15212

[30] M. R. Ibrahim and T. Lyons, “ImageSig: A signature transform for ultra-lightweight image recognition,” in *2022 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, 2022, pp. 3648–3658. doi: 10.1109/cvprw56347.2022.00409.

[31] Z. Xie, Z. Sun, L. Jin, H. Ni, and T. Lyons, “Learning Spatial-Semantic Context with Fully Convolutional Recurrent Network for Online Handwritten Chinese Text Recognition,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 40, no. 8, pp. 1903–1917, 2018, doi: 10.1109/tpami.2017.2732978.

[32] A. Meurer *et al.*, “Python Array API Standard: Toward Array Interoperability in the Scientific Python Ecosystem,” in *Proceedings of the 22nd Python in Science Conference*, in SciPy. 2023, pp. 8–17. doi: 10.25080/gerudo-f2bc6f59-001.

[33] DLPack, “Open In Memory Tensor structure.” 2023.

[34] W. Jakob, J. Rhinelander, and D. Moldovan, “pybind11 – Seamless operability between C++11 and Python.” 2017.

[35] S. Buckley *et al.*, “CoRoPa – Computational Rough Paths project.” 2006.

[36] T. Granlund and the GMP development team, “GNU MP: The GNU Multiple Precision Arithmetic Library,” 2012.

Open Access | https://doi.org/10.25080/XHDR4700

SciPy 2024
July 8 - July 14, 2024

Proceedings of the 23nd
Python in Science Conference
ISSN: 2575-9752

# Mamba Models a possible replacement for Transformers?

## A Memory-Efficient Approach for Scientific Computing

**Suvrakamal Das**[1], **Rounak Sen**[1], and **Saikrishna Devendiran**[2]

[1]Maulana Abul Kalam Azad University Institute of Technology, West Bengal, [2]Amrita School of Computing, Amritapuri

### Abstract

The quest for more efficient and faster deep learning models has led to the development of various alternatives to Transformers, one of which is the Mamba model. This paper provides a comprehensive comparison between Mamba models and Transformers, focusing on their architectural differences, performance metrics, and underlying mechanisms. It analyzes and synthesizes findings from extensive research conducted by various authors on these models. The synergy between Mamba models and the SciPy ecosystem enhances their integration into science. By providing an in-depth comparison using Python and its scientific ecosystem, this paper aims to clarify the strengths and weaknesses of Mamba models relative to Transformers. It offers the results obtained along with some thoughts on the possible ramifications for future research and applications in a range of academic and professional fields.

**Keywords** Mamba Models, Solid State Models, Transformers, Flash Attention, LLMs

## 1. Introduction

The rapid advancements in deep learning have led to transformative breakthroughs across various domains, from natural language processing to computer vision. However, the quest for more efficient and scalable models remains a central challenge, especially when dealing with long sequences exhibiting long-range dependencies. Transformers, while achieving remarkable performance in numerous tasks, often suffer from high computational complexity and memory usage, particularly when handling long sequences.

This paper delves into the emerging field of State Space Models (SSMs) as a promising alternative to Transformers for efficiently capturing long-range dependencies in sequential data. We provide a comprehensive comparison between the recently developed Mamba model, based on SSMs, and the widely adopted Transformer architecture, highlighting their architectural differences, performance characteristics, and underlying mechanisms.

We begin by exploring the fundamental principles of SSMs, emphasizing their ability to represent and model continuous-time systems through a latent state vector. We then introduce the HiPPO framework, which extends SSMs to effectively handle long-range dependencies by leveraging the properties of orthogonal polynomials. This leads us to the discretization of continuous-time SSMs into discrete-time representations, enabling their implementation as recurrent models.

Building upon this foundation, we introduce the Structured State Space (S4) model, which addresses the computational limitations of traditional SSM implementations by employing a novel parameterization and efficient algorithms. S4's Normal Plus Low-Rank (NPLR) decomposition allows for stable and efficient diagonalization of the state matrix, leading to significant improvements in computational complexity.

**Published** Jul 10, 2024

**Correspondence to**
Suvrakamal Das
subhrokomol@gmail.com

We then discuss the Mamba model, which leverages the selective SSM approach to capture long-range dependencies in sequences. The Mamba architecture combines aspects of RNNs, CNNs, and classical state space models, offering a unique blend of efficiency and expressivity.

The paper then presents a detailed comparison of Mamba and Transformer architectures, highlighting their core components, computational characteristics, and performance implications. We demonstrate the advantages of Mamba in terms of computational efficiency, memory usage, and sequence handling, underscoring its potential for tackling complex scientific and industrial problems.

Finally, we explore the potential applications and future directions of Mamba models, particularly in the context of scientific computing and data analysis. We highlight the synergy between Mamba and the SciPy ecosystem, underscoring its ability to enhance the efficiency and scalability of scientific computing workflows and drive novel scientific discoveries.

## 2. State Space Models

The central goal of machine learning is to develop models capable of efficiently processing sequential data across a range of modalities and tasks [1]. This is particularly challenging when dealing with **long sequences**, especially those exhibiting **long-range dependencies (LRDs)** – where information from distant past time steps significantly influences the current state or future predictions. Examples of such sequences abound in real-world applications, including speech, video, medical, time series, and natural language. However, traditional models struggle to effectively handle such long sequences.

**Recurrent Neural Networks (RNNs)** [2], often considered the natural choice for sequential data, are inherently stateful and require only constant computation per time step. However, they are slow to train and suffer from the well-known “**vanishing gradient problem**”, which limits their ability to capture LRDs. **Convolutional Neural Networks (CNNs)** [3], while efficient for parallelizable training, are not inherently sequential and struggle with long context lengths, resulting in more expensive inference. **Transformers** [4], despite their recent success in various tasks, typically require specialized architectures and attention mechanisms to handle LRDs, which significantly increase computational complexity and memory usage.

A promising alternative for tackling LRDs in long sequences is **State Space Models (SSMs)** [5], a foundational mathematical framework deeply rooted in diverse scientific disciplines like control theory and computational neuroscience. SSMs provide a continuous-time representation of a system’s state and evolution, offering a powerful paradigm for capturing LRDs. While SSMs and S4s does not prevent the vanishing gradient problem but it reduces the impact with the help of HiPPO framework and NPLR Parametrization. They represent a system’s behavior in terms of its internal **state** and how this state evolves over time. SSMs are widely used in various fields, including control theory, signal processing, and computational neuroscience.

### *2.1. Continuous-time Representation*

The continuous-time SSM describes a system’s evolution using differential equations. It maps a continuous-time input signal $u(t)$ to an output signal $y(t)$ through a latent state $x(t)$. The state is an internal representation that captures the system’s history and influences its future behavior.

The core equations of the continuous-time SSM are:

- **State Evolution:**

$$x'(t) = Ax(t) + Bu(t) \tag{1}$$

- **Output Generation:**

$$y(t) = Cx(t) + Du(t) \tag{2}$$

where:

- $x(t)$ is the state vector at time $t$, belonging to a $N$-dimensional space.
- $u(t)$ is the input signal at time $t$.
- $y(t)$ is the output signal at time $t$.
- $A$ is the state matrix, controlling the evolution of the state vector $x(t)$.
- $B$ is the control matrix, mapping the input signal $u(t)$ to the state space.
- $C$ is the output matrix, projecting the state vector $x(t)$ onto the output space.
- $D$ is the command matrix, directly mapping the input signal $u(t)$ to the output. (For simplicity, we often assume $D$ = 0, as $Du(t)$ can be viewed as a skip connection.)

This system of equations defines a continuous-time mapping from input $u(t)$ to output $y(t)$ through a latent state $x(t)$. The state matrix $A$ plays a crucial role in determining the dynamics of the system and its ability to capture long-range dependencies.

### *2.2. HiPPO Framework for Long-Range Dependencies*

Despite their theoretical elegance, naive applications of SSMs often struggle with long sequences. This is due to the inherent limitations of simple linear differential equations in capturing long-range dependencies (LRDs). To overcome this, the **High-Order Polynomial Projection Operator (HiPPO)** [6] framework provides a principled approach for designing SSMs specifically suited for LRDs.

HiPPO focuses on finding specific state matrices $A$ that allow the state vector $x(t)$ to effectively memorize the history of the input signal $u(t)$. It achieves this by leveraging the properties of orthogonal polynomials. The HiPPO framework derives several structured state matrices, including:

- **HiPPO-LegT (Translated Legendre):** Based on Legendre polynomials, this matrix enables the state to capture the history of the input within sliding windows of a fixed size.
- **HiPPO-LagT (Translated Laguerre):** Based on Laguerre polynomials, this matrix allows the state to capture a weighted history of the input, where older information decays exponentially.
- **HiPPO-LegS (Scaled Legendre):** Based on Legendre polynomials, this matrix captures the history of the input with respect to a linearly decaying weight.

### *2.3. Discrete-time SSM: Recurrent Representation*

To apply SSMs on discrete-time data sequences ($u_0$, $u_1$, ...), it's necessary to discretize the continuous-time model. This involves converting the differential equations into difference equations, where the state and input are defined at discrete time steps. One common discretization method is the **bilinear transform**, also known as the **Tustin method**. This transform approximates the derivative $x'(t)$ by a weighted average of the state values at two consecutive time steps, introducing a **step size** $\delta$ that represents the time interval between samples.

SSMs Figure 1 typically require integration within a broader neural network architecture due to their limited inherent capabilities. From a high-level perspective, SSMs exhibit functional similarities to linear Recurrent Neural Networks (RNNs). Both architectures process sequential input tokens, transforming and combining the previous hidden state representation with the embedding of the current input. This iterative processing characteristic aligns SSMs with the sequential nature of RNNs.

SSMs have 4 sets of matrices and parameters to process the input namely

$$\Delta, A, B, C \tag{3}$$

where:

- $\Delta$ acts as a gating factor, selectively weighting the contribution of matrices $A$ and $B$ at each step. This allows the model to dynamically adjust the influence of past hidden states and current inputs.
- $A$ represents the state transition matrix. When modulated by $\Delta$, it governs the propagation of information from the previous hidden state to the current hidden state.
- $B$ denotes the input matrix. After modulation by $\Delta$, it determines how the current input is integrated into the hidden state.
- $C$ serves as the output matrix. It maps the hidden state to the model's output, effectively transforming the internal representations into a desired output space.

The discretization technique facilitates the transformation of continuous differential equations into discrete time-step representations, leveraging the $\Delta$ matrix to decompose the infinitely continuous process into a discrete time-stepped process, thereby reducing computational complexity. In this approach, the $A$ and $B$ steps undergo discretization through the following equations:

$$\overline{A} = \exp(\Delta A) \tag{4}$$

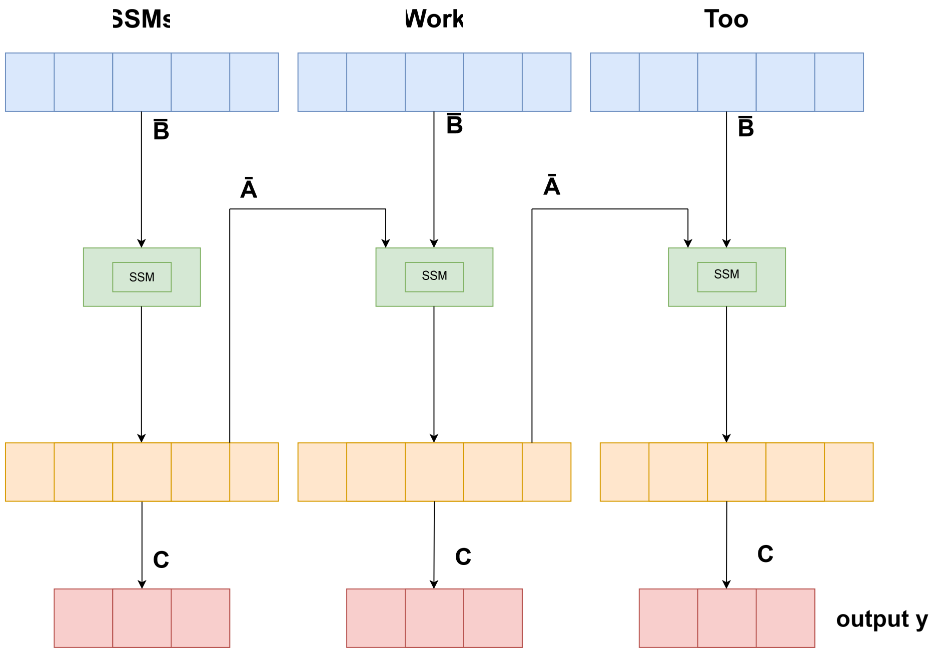


**Figure 1**. *This diagram illustrates the architecture of the Selective State Space Model (SSM). Each input sequence (blue boxes) is processed by an SSM layer (green) after being multiplied by matrix $\overline{B}$. The state space module (SSM) handles sequential data and captures long-range dependencies, after which the output (yellow boxes) undergoes a transformation by matrix $\overline{C}$. The final outputs (red boxes) are combined to produce the model's output $y$. This setup emphasizes the modular and repetitive nature of SSM layers, highlighting their role in sequence modeling.*

$$\overline{B} = (\Delta A)^{-1}(\exp(\Delta A) - I) \cdot \Delta B \tag{5}$$

This discretization scheme effectively reduces the continuous differential equation to a series of discrete time steps, enabling numerical approximations to be computed iteratively. By segmenting the continuous process into finite increments, the computational burden is alleviated, rendering the problem more tractable for numerical analysis and simulation.

### *2.4. Training SSMs*

While the recurrent representation provides a computationally efficient way to perform inference with an SSM, it is not optimal for training due to its sequential nature. To overcome this, SSM leverages the connections between linear time-invariant (LTI) SSMs and convolution. The convolutional representation allows for efficient parallel training using Fast Fourier Transform (FFT) algorithms. However, the main challenge lies in computing the SSM convolution kernel $K$. Computing it naively with $L$ successive matrix multiplications by $A$ results in $O(N^2 * L)$ operations and $O(NL)$ memory – a significant computational bottleneck for long sequences.

The state-space models (SSMs) compute the output using a linear recurrent neural network (RNN) architecture, which operates on a hidden state $\Delta$. In this formulation, the hidden state propagates through a linear equation of the following form:

$$h_t = \overline{A}h_{t-1} + \overline{B}x_t \tag{6}$$

where

- $h_t$ is hidden state matrix at time step t
- $x_t$ is input vector at time t

The initial hidden state $h_0$ is computed as:

$$h_0 = \overline{A}h_{-1} + \overline{B}x_0 = \overline{B}x_0 \tag{7}$$

Subsequently, the hidden state at the next time step, $h_1$, is obtained through the recursion:

$$h_1 = \overline{A}h_0 + \overline{B}x_1 = \overline{AB} \tag{8}$$

The output $Y_t$ is then calculated from the hidden state $h_t$ using the following linear transformation:

$$y_t = Ch_t \tag{9}$$

- C is the output control matrix
- $y_t$ is output vector at time t
- $h_t$ is the Internal hidden state at time t

$$\begin{aligned} y_0 &= Ch_0 = C\overline{B}x_0 \\ y_1 &= Ch_1 = C\overline{AB}x_0 + C\overline{B}x_1 \\ y_2 &= C\overline{A}^2\overline{B}x_0 + C\overline{AB}x_1 + C\overline{B}x_2 \\ &\vdots \\ y_t &= C\overline{A}^t\overline{B}x_0 + C\overline{A}^{t-1}\overline{B}x_1 + \ldots + C\overline{AB}x_{t-1} + C\overline{B}x_t \end{aligned} \tag{10}$$

$$Y = K \cdot X \tag{11}$$

where :

- $X$ is the input matrix ***i.e.*** $[x_0, x_1, \ldots, x_L]$
- $K = \left(C\overline{B}, C\overline{AB}, \ldots, C\overline{A}^{L-1}\overline{B}\right)$

This linear RNN architecture effectively captures the temporal relationships present in sequential data and thus enabling the model to learn and ensure the propagation of the key-relevant information through the recurrent connections. This linear formulation has led to scalable implementations, which in turn take use of matrix operations' computing efficiency.

## 3. S4: A Structured State Space Model

The theoretical advantages of State Space Models (SSMs) [5] for handling long sequences, particularly their ability to capture long-range dependencies, make them a promising alternative to traditional sequence models. However, the computational limitations of existing SSM implementations, such as the LSSL, hinder their widespread adoption. The Structured State Space (S4) model aims to overcome these limitations by introducing novel parameterization [7] and efficient algorithms that preserve the theoretical strengths of SSMs [8] .

## 4. Diagonalization Problem

The core computational bottleneck in SSMs stems from repeated matrix multiplication by the state matrix $A$ when calculating the convolution kernel $K$. If $A$ were a diagonal matrix, this computation would become significantly more tractable. Diagonal matrices allow for efficient power calculations as well as multiplication by a vector, resulting in a time complexity of $O(N)$ for $N$ dimensions.

Diagonalization involves finding a change of basis that transforms $A$ into a diagonal form. However, this approach faces significant challenges when $A$ is **non-normal**. Non-normal matrices have complex eigenstructures, which can lead to several problems:

- **Numerically unstable diagonalization:** Diagonalizing non-normal matrices can be numerically unstable, especially for large matrices. This is because the eigenvectors may be highly sensitive to small errors in the matrix, leading to large errors in the computed eigenvalues and eigenvectors.
- **Exponentially large entries:** The diagonalization of some non-normal matrices, including the HiPPO matrices, can involve matrices with entries that grow exponentially with the dimension $N$. This can lead to overflow issues during computation and render the diagonalization infeasible in practice.

Therefore, naive diagonalization of non-normal matrices in SSMs is not a viable solution for efficient computation.

## 5. The S4 Parameterization: Normal Plus Low-Rank (NPLR)

S4 overcomes the challenges of directly diagonalizing non-normal matrices by introducing a novel parameterization [7]. It decomposes the state matrix $A$ into a sum of a **normal matrix** and a **low-rank term**. This decomposition allows for efficient computation while preserving the structure necessary to handle long-range dependencies. The S4 parameterization is expressed as follows:

- SSM convolution kernel

$$\overline{K} = \kappa_L\left(\overline{A}, \overline{B}, \overline{C}\right) \text{ med med med for med med med} A = V\Lambda V^* - PQ^T \tag{12}$$

where:

- $V$ is a unitary matrix that diagonalizes the normal matrix.
- $\Lambda$ is a diagonal matrix containing the eigenvalues of the normal matrix.
- $P$ and $Q$ are low-rank matrices that capture the non-normal component.
- These matrices HiPPO - $LegS, LegT, LagT$ all satisfy $r$ = 1 or $r$ = 2.

This decomposition allows for efficient computation because:

- **Normal matrices are efficiently diagonalizable:** Normal matrices can be diagonalized stably and efficiently using unitary transformations.
- **Low-rank corrections are tractable:** The low-rank term can be corrected using the Woodbury identity, a powerful tool for inverting matrices perturbed by low-rank terms.

## 6. S4 Algorithms and Complexity

S4 leverages its NPLR parameterization to develop efficient algorithms for computing both the recurrent representation ($A$) and the convolutional kernel ($K$).

### *6.1. S4 Recurrence*

The bilinear transform is used to discretize the state matrix in order to construct the S4 recurrent representation. The important thing to note is that, because of the Woodbury identity, the inverse of a DPLR matrix does not result in any change in the matrix. Therefore, the discretized state matrix is the product of two DPLR matrices, allowing for efficient matrix-vector multiplication in O(N) time.

### *6.2. Parallel Associative Scan*

The linear recurrent behavior inherent in the previous formulation is not efficiently implementable on GPU architectures, which favor parallel computing paradigms. This limitation renders convolutions inefficient in such environments. To address this challenge, the parallel associative scan technique is employed, which introduces a prefix sum-like operation to scan for all prefix sums. Although inherently sequential, this approach leverages an efficient parallel algorithm model to parallelize the SSM convolution, resulting in a significant performance boost. The parallel associative scan method exhibits linear time and space complexity, making it a computationally efficient solution.

SSM

$$y_t = C\overline{A}^t\overline{B}x_0 + C\overline{A}^{t-1}\overline{B}x_1 + ... + C\overline{A}\overline{B}x_{t-1} + C\overline{B}x_t \tag{13}$$

Selective SSM

$$\begin{gathered} y_t = C_0\overline{A}^t\overline{B}_0x_0 + C_1\overline{A}^{t-1}\overline{B}_1x_1 + ... \\ \text{input -dependent} B \text{ and} C \text{ matrix} \end{gathered} \tag{14}$$

By leveraging the parallel associative scan technique [9], the selective SSM formulation can be efficiently implemented on parallel architectures, such as GPUs. This approach enables the exploitation of the inherent parallelism in the computation, leading to significant performance gains, particularly for large-scale applications and time-series data processing tasks.

### *6.3. S4 Convolution*

S4's convolutional representation is computed through a series of steps:

1. **SSM Generating Function:** Instead of directly computing the convolution kernel $K$, S4 calculates its spectrum by evaluating its truncated generating function. The generating function allows for efficiently expressing powers of $A$ as a single matrix inverse.
2. **Woodbury Correction:** The Woodbury identity is used to correct the low-rank term in the generating function, reducing the problem to evaluating the generating function for a diagonal matrix.

3. **Cauchy Kernel:** The generating function for a diagonal matrix is equivalent to computing a Cauchy kernel, which is a well-studied problem with efficient, numerically stable algorithms.

This process reduces the complexity of computing the convolution kernel $K$ to $O(N+L)$ operations and $O(N+L)$ memory, significantly improving upon the LSSL's complexity.

### *6.4. S4 Architecture Details*

The S4 layer, as defined by its NPLR parameterization, implements a mapping from a 1-D input sequence to a 1-D output sequence. To handle multiple features, the S4 architecture utilizes $H$ independent copies of the S4 layer, each processing one feature dimension. These outputs are then mixed using a position-wise linear layer, similar to a depthwise-separable convolution. This architecture allows for efficient computation while preserving the ability to capture relationships between different features.

Non-linear activation functions are typically added between S4 layers to enhance the model's expressivity, further paralleling the structure of CNNs. Thus, the overall deep S4 model resembles a depthwise-separable CNN, but with global convolution kernels that effectively capture long-range dependencies.

In summary, S4 offers a structured and efficient approach to SSMs, overcoming the limitations of previous implementations while preserving their theoretical strengths. Its NPLR parameterization allows for stable and efficient computation, while its efficient algorithms significantly reduce computational complexity. S4's ability to handle multiple features and its resemblance to CNNs further contribute to its versatility and potential as a powerful general sequence modeling solution.

## 7. Mamba Model Architecture

One Mamba Layer [10] Figure 2 is composed of a selective state-space module and several auxiliary layers. Initially, a linear layer doubles the dimensionality of the input token embedding, increasing the dimensionality from 64 to 128. This higher dimensionality provides the network with an expanded representational space, potentially enabling the separation of previously inseparable classes. Subsequently, a canonical 1D convolution layer processes the output of the previous layer, manipulating the dimensions within the linearly upscaled 128-dimensional vector. This convolution layer employs the **SiLU (Sigmoid-weighted Linear Unit)** activation function [11]. The output of the convolution is then processed by the selective state-space module, which operates akin to a linear recurrent neural network (RNN).

Mamba then performs a gated multiplication operation. The input is passed through another linear layer and an activation function, and the resulting output is multiplied element-wise with the output of the S4 module. The authors' intuition behind this operation is that the multiplication serves as a measure of similarity between the output of the SSM module, which contains information from previous tokens, and the embedding of the current token. Finally, a linear layer reduces the dimensionality from 128 back to 64. To construct the complete Mamba architecture, multiple layers are stacked on top of one another, similar to the Transformer architecture, where Transformer layers are stacked sequentially.

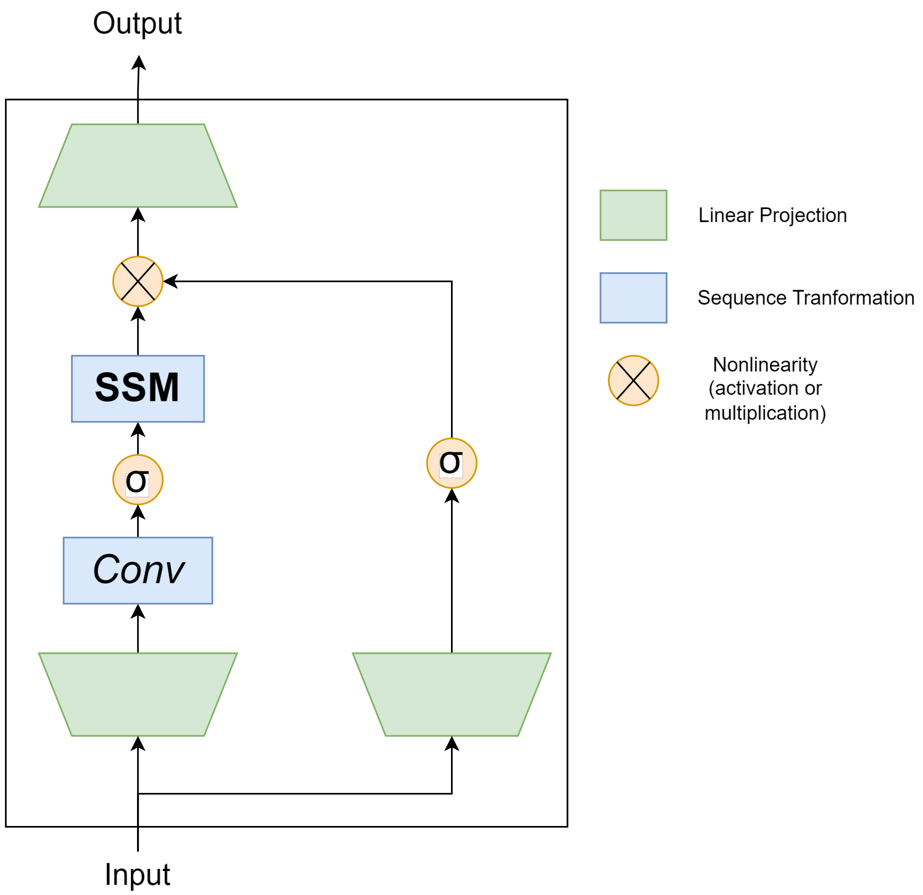


**Figure 2.** *This diagram represents the Mamba architecture, illustrating the flow from input through convolutional and sequence transformation layers, with nonlinear activation functions, to produce the final output via linear projection.*

## 8. Key Differences Between Mamba and Transformer Architectures

In this section, we present a detailed comparison of the Mamba and Transformer architectures. We focus on their core components, computational characteristics, and performance implications. Visualizations and equations are provided to illustrate these differences clearly.

Self attention, feed forward Neural Networks, normalization, residual layers and so on.

### *8.1. Architecture Overview*

#### 8.1.1. *Transformer Architecture:*

Transformers Figure 3 rely heavily on attention mechanisms to model dependencies between input and output sequences. A better understanding of the code will be of great help [12].

The core components include:

- **Multi-Head Self-Attention**: Allows the model to focus on different parts of the input sequence.
- **Position-wise Feed-Forward Networks**: Applied to each position separately.

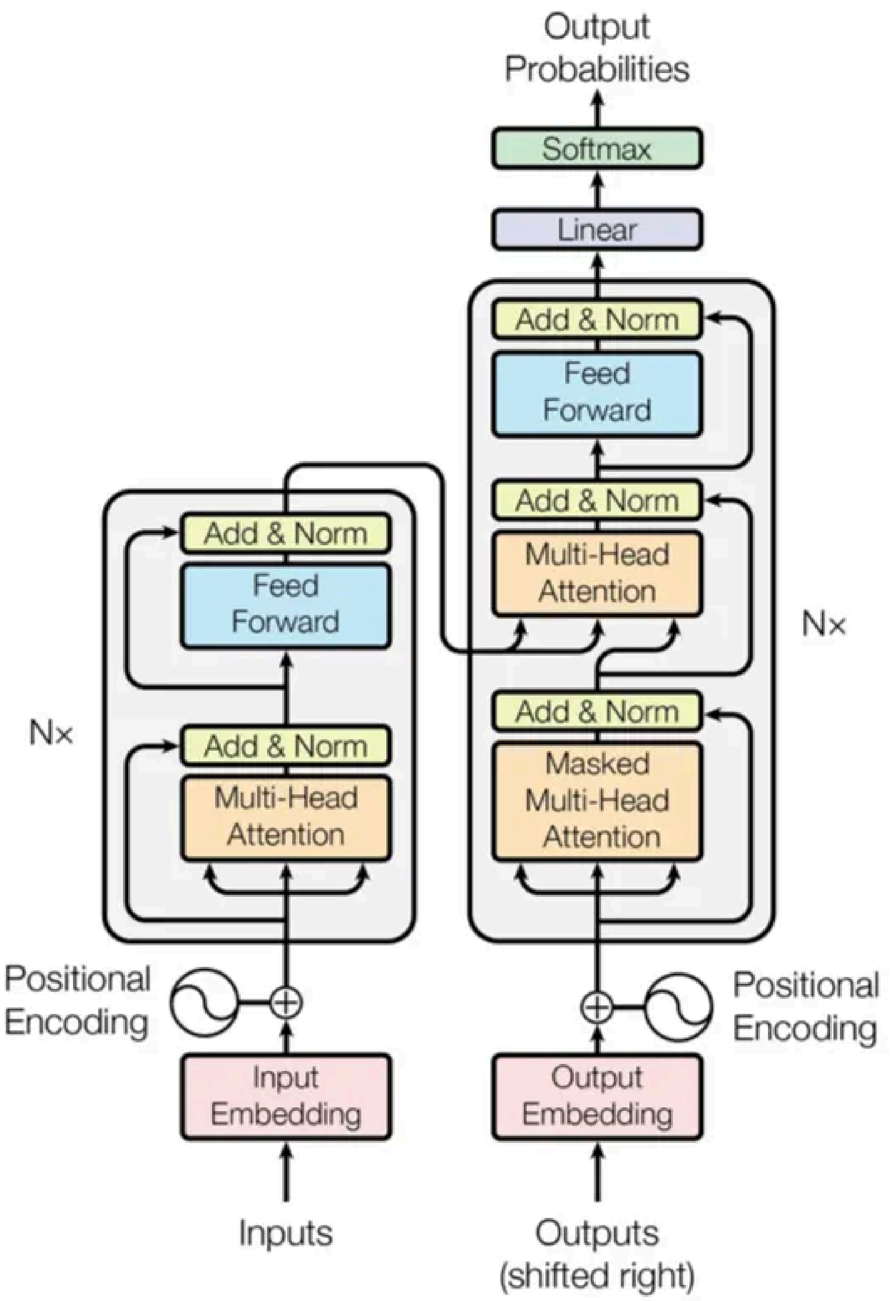


**Figure 3**. *This diagram illustrates the transformer model architecture, featuring encoder and decoder layers with multi-head attention mechanisms, positional encoding, and feed-forward networks, culminating in output probabilities via a softmax layer.*

- **Positional Encoding**: Adds information about the position of each token in the sequence, as Transformers lack inherent sequential information due to the parallel nature of their processing.

### 8.1.2. *Mamba Architecture:*

Mamba models Figure 2 are based on Selective State Space Models (SSMs), combining aspects of RNNs, CNNs, and classical state space models. Key features include:

- **Selective State Space Models**: Allow input-dependent parameterization to selectively propagate or forget information.
- **Recurrent Mode**: Efficient recurrent computations with linear scaling.
- **Hardware-aware Algorithm**: Optimized for modern hardware to avoid inefficiencies from the Flash Attention 2 Paper.

### *8.2. Key Differences*

#### 8.2.1. *1. Attention Mechanisms vs. Selective State Space Models:*

**Transformers** use multi-head self-attention to capture dependencies within the sequence:

$$\mathrm{Attention}(Q, K, V) = \mathrm{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{15}$$

Where $Q$, $K$, and $V$ are the query, key, and value matrices, respectively, and $d$ is the dimension of the key vectors.

**Mamba Models** replace attention with selective state space parameters that change based on the input:

$$h'(t) = Ah(t) + Bx(t) \tag{16}$$

$$y(t) = Ch(t) \tag{17}$$

Here, $A$, $B$, and $C$ are state space parameters that vary with the input, allowing for efficient handling of long sequences without the quadratic complexity of attention mechanisms.

#### 8.2.2. *2. Computational Complexity:*

| **Feature** | **Architecture** | **Complexity** | **Inference Speed** | **Training Speed** |
|---|---|---|---|---|
| Transformer | Attention-based | High | O(n) | O(n²) |
| Mamba | SSM-based | Lower | O(1) | O(n) |

#### 8.2.3. *3. Sequence Handling and Memory Efficiency:*

**Transformers** require a cache of previous elements to handle long-range dependencies, leading to high memory usage.

**Mamba Models** utilize selective state spaces to maintain relevant information over long sequences without the need for extensive memory caches, providing a more memory-efficient solution.

Mamba integrates selective state spaces directly into the neural network architecture. The selective mechanism allows the model to focus on relevant parts of the input dynamically.

There are other competing architectures that aim to replace or complement Transformers, such as Retentive Network [13], Griffin [14], Hyena [15], and RWKV [16]. These architectures propose alternative approaches to modeling sequential data, leveraging techniques like gated linear recurrences, local attention, and reinventing recurrent neural networks (RNNs) for the Transformer era.

## 9. Mamba's Synergy with Scipy

Scipy [17] provides a robust ecosystem for scientific computing in Python, offering a wide range of tools and libraries for numerical analysis, signal processing, optimization, and more. This ecosystem serves as a fertile ground for the development and integration of Mamba, facilitating its training, evaluation, and deployment in scientific applications.

Leveraging Scipy's powerful data manipulation and visualization capabilities, Mamba models can be seamlessly integrated into scientific workflows, enabling in-depth analysis, rigorous statistical testing, and clear visualization of results.

The combination of Mamba's language understanding capabilities and Scipy's scientific computing tools opens up new avenues for exploring large-scale scientific datasets commonly encountered in scientific research domains such as astronomy, medicine, and beyond, extracting insights, and advancing scientific discoveries.

### 9.1. Potential Applications and Future Directions:

- **Efficient Processing of Large Scientific Datasets:** Mamba's ability to handle long-range dependencies makes it well-suited for analyzing and summarizing vast amounts of scientific data, such as astronomical observations, medical records, or experimental results, thereby reducing the complexity and enabling more efficient analysis.
- **Enhancing Model Efficiency and Scalability:** Integrating Mamba with Scipy's optimization and parallelization techniques can potentially improve the efficiency and scalability of language models, enabling them to handle increasingly larger datasets and more complex scientific problems.
- **Advancing Scientific Computing through Interdisciplinary Collaboration:** The synergy between Mamba and Scipy fosters interdisciplinary collaboration between natural language processing researchers, scientific computing experts, and domain-specific scientists, paving the way for novel applications and pushing the boundaries of scientific computing.

The diverse range of models as U-Mamba [18], Vision Mamba[19], VMamba [20], MambaByte [21]and Jamba [22], highlights the versatility and adaptability of the Mamba architecture. These variants have been designed to enhance efficiency, improve long-range dependency modeling, incorporate visual representations, explore token-free approaches, integrate Fourier learning, and hybridize with Transformer components.

## 10. Conclusion

Mamba models present a compelling alternative to Transformers for processing long sequences, particularly in scientific computing. Their use of selective state spaces delivers linear time complexity and superior memory efficiency, making them faster and less resource-intensive than Transformers for lengthy data. Mamba's flexible architecture enables easy integration with scientific workflows and scalability. However, their complexity demands further research to streamline implementation and encourage wider adoption. While not yet a complete replacement for Transformers, Mamba models offer a powerful tool for analyzing complex scientific data where efficiency and integration with scientific tools are paramount, making their continued development crucial.

## References

[1] Albert Gu, Tri Dao, "Mamba: Linear-Time Sequence Modeling with Selective State Spaces." [Online]. Available: https://github.com/state-spaces/mamba

[2] A. Sherstinsky, "Fundamentals of Recurrent Neural Network (RNN) and Long Short-Term Memory (LSTM) network," *Physica D: Nonlinear Phenomena*, vol. 404, p. 132306, 2020, doi: 10.1016/j.physd.2019.132306.

[3] K. O'Shea and R. Nash, "An Introduction to Convolutional Neural Networks." 2015. doi: https://doi.org/10.48550/arXiv.1511.08458.

[4] A. Vaswani *et al.*, "Attention Is All You Need." 2023. doi: https://doi.org/10.48550/arXiv.1706.03762.

[5] A. Gu, K. Goel, and C. Ré, "Efficiently Modeling Long Sequences with Structured State Spaces." 2022. doi: https://doi.org/10.48550/arXiv.2111.00396.

[6] A. Gu, T. Dao, S. Ermon, A. Rudra, and C. Re, “HiPPO: Recurrent Memory with Optimal Polynomial Projections.” 2020. doi: https://doi.org/10.48550/arXiv.2008.07669.

[7] A. Gu, A. Gupta, K. Goel, and C. Ré, “On the Parameterization and Initialization of Diagonal State Space Models.” 2022. doi: https://doi.org/10.48550/arXiv.2206.11893.

[8] Albert Gu, Karan Goel, Christopher Ré, “Structured State Spaces for Sequence Modeling.” [Online]. Available: https://github.com/state-spaces/s4

[9] Y. H. Lim, Q. Zhu, J. Selfridge, and M. F. Kasim, “Parallelizing non-linear sequential models over the sequence length.” 2024. doi: https://doi.org/10.48550/arXiv.2309.12252.

[10] A. Gu and T. Dao, “Mamba: Linear-Time Sequence Modeling with Selective State Spaces.” 2023. doi: https://doi.org/10.48550/arXiv.2312.00752.

[11] S. Elfwing, E. Uchibe, and K. Doya, “Sigmoid-Weighted Linear Units for Neural Network Function Approximation in Reinforcement Learning.” 2017. doi: https://doi.org/10.48550/arXiv.1702.03118.

[12] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, Illia Polosukhin, “Transformer model from Attention Is All You Need.” [Online]. Available: https://github.com/tensorflow/tensor2tensor

[13] Y. Sun *et al.*, “Retentive Network: A Successor to Transformer for Large Language Models.” 2023. doi: https://doi.org/10.48550/arXiv.2307.08621.

[14] S. De *et al.*, “Griffin: Mixing Gated Linear Recurrences with Local Attention for Efficient Language Models.” 2024. doi: https://doi.org/10.48550/arXiv.2402.19427.

[15] M. Poli *et al.*, “Hyena Hierarchy: Towards Larger Convolutional Language Models.” 2023. doi: https://doi.org/10.48550/arXiv.2302.10866.

[16] B. Peng *et al.*, “RWKV: Reinventing RNNs for the Transformer Era.” 2023. doi: https://doi.org/10.48550/arXiv.2305.13048.

[17] P. Virtanen *et al.*, “SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python,” *Nature Methods*, vol. 17, pp. 261–272, 2020, doi: https://doi.org/10.1038/s41592-019-0686-2.

[18] J. Ma, F. Li, and B. Wang, “U-Mamba: Enhancing Long-range Dependency for Biomedical Image Segmentation.” 2024. doi: https://doi.org/10.48550/arXiv.2401.04722.

[19] L. Zhu, B. Liao, Q. Zhang, X. Wang, W. Liu, and X. Wang, “Vision Mamba: Efficient Visual Representation Learning with Bidirectional State Space Model.” 2024. doi: https://doi.org/10.48550/arXiv.2401.09417.

[20] Y. Liu *et al.*, “VMamba: Visual State Space Model.” 2024. doi: https://doi.org/10.48550/arXiv.2401.10166.

[21] J. Wang, T. Gangavarapu, J. N. Yan, and A. M. Rush, “MambaByte: Token-free Selective State Space Model.” 2024. doi: https://doi.org/10.48550/arXiv.2401.13660.

[22] O. Lieber *et al.*, “Jamba: A Hybrid Transformer-Mamba Language Model.” 2024. doi: https://doi.org/10.48550/arXiv.2403.19887.

Open Access | https://doi.org/10.25080/RVRR7774

SciPy 2024
July 8 - July 14, 2024

Proceedings of the 23nd
Python in Science Conference
ISSN: 2575-9752

# THEIA: An Offline Tool for Tradespace Visualization

**Samuel Williams**[1,2], **Scott Christensen**[1,2], and **Marvin Brown**[1,2]

[1]U.S. Army Engineer Research & Development Center, [2]U.S. Army Corps of Engineers

## Abstract

Within the Army Corps of Engineers (USACE), there is a need to evaluate tradespaces. Tradespace datasets are the result of large parameter sweeps run over numerous design options and can consist of thousands or even millions of design configurations and the corresponding performance metrics. Because of the immense size of these datasets, the ability to effectively visualize the data is essential for proper evaluation. At the USACE Engineer Research & Development Center (ERDC), an easy-to-use plotting tool known as the Tradespace Holistic Exploration & Insight Application (THEIA) has been developed for visualizing this complex tradespace data related to the acquisitions process. THEIA was developed using Python libraries including Panel, Param, Holoviews, Bokeh, and Plotly. When combined, these libraries offer a wide range of widgets and plots that allow the user to visualize their data in multiple ways. Additionally, users can easily save plots, export findings, and utilize multiple data files at once. THEIA is also capable of importing tabular data while presenting options to customize visualizations and help the end-user make informed decisions.

**Keywords** python, panel, holoviews, holoviz, param, plotly, bokeh, tradespace

## 1. Background/Motivation

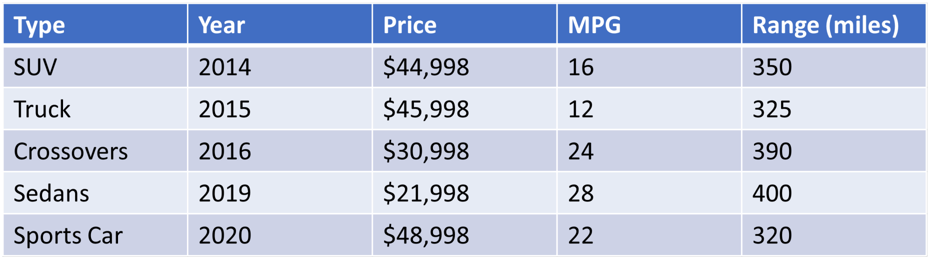
Within the United States Army Corps of Engineers (USACE) the evaluation of tradespaces has become an important capability. Tradespaces, in essence, are extensive datasets generated from broad parameter sweeps across a multitude of design options. These datasets can encompass thousands, if not millions of design configurations, each accompanied by its respective performance metrics. This process results in an immense amount of data, which, while rich in information, presents a challenge in terms of analysis and interpretation. Because of the colossal size of these datasets, the ability to visualize them effectively is not just beneficial, but essential for comprehensive evaluation. This visualization is particularly crucial when it come to the acquisitions process, where understanding the implications of each design choice can have far-reaching consequences. To address this need, the USACE Engineer Research & Development Center (ERDC) has developed a user-friendly plotting tool, named the Tradespace Holistic Exploration & Insight Application (THEIA).

## 2. What is a tradespace?

In the context of this paper, the term tradespace is a combination of the terms “parameter space” and “tradeoff”. A tradespace dataset is a dataset that shows tradeoffs associated with certain design choices in a parameter space. Oftentimes, these datasets are generated by high performance computing (HPC) jobs called parameter sweeps.

**Published** Jul 10, 2024

**Correspondence to**
Samuel Williams
Samuel.C.Williams@usace.army.mil



| Type | Year | Price | MPG | Range (miles) |
|---|---|---|---|---|
| SUV | 2014 | $44,998 | 16 | 350 |
| Truck | 2015 | $45,998 | 12 | 325 |
| Crossovers | 2016 | $30,998 | 24 | 390 |
| Sedans | 2019 | $21,998 | 28 | 400 |
| Sports Car | 2020 | $48,998 | 22 | 320 |

**Figure 1.** *This is a simple example of a tradespace in the context of car shopping. When shopping for a new vehicle, buyers often compile a list of pros and cons based on certain vehicle types. The dataset the buyer creates is, in essence, a tradespace. For example, a buyer may want to maximize potential range, while minimizing vehicle cost. By analyzing this tradespace, the buyer would be able to see that opting for a sedan provides both the best price and the best potential range.*

## 3. Methods

THEIA's primary function is to provide a means of visualizing intricate tradespace data, thereby facilitating a more thorough understanding of the various design configurations and their corresponding performance metrics. It's a robust tool built upon several Python libraries, including Panel, Bokeh, Param, HoloViews, and Plotly. The integration of these libraries equips THEIA with a diverse array of widgets and plots, enabling users to visualize their data in various ways simultaneously. Panel, Param, and HoloViews are all a part of the HoloViz suite of Python packages, which allows for excellent synchronization between them.

### *3.1. Panel*

Panel [1] is an open-source library for easily building tools, dashboards, and complex applications, all within Python. Its role in THEIA is to define the layout of the application, in an easy, extensible way. This is done by utilizing Panel's built-in visual components. Components are the building blocks of apps built on Panel. The primary components utilized by THEIA are Widgets, Panes, Layouts, and Templates.

- Widgets are generally small components like sliders, text fields, or checkboxes.
- Panes are data wrappers that enable rendering of that data in multiple ways.
- Layouts allow for the coupling of various components like widgets or panes so that they can be rendered together using the Bokeh library.
- Templates are components that allow the rendering of Panel objects in an HTML document.

### *3.2. Param*

Param [2] is an open-source library built for handling user-modifiable parameters, arguments, and attributes that control your code. These capabilities are used in THEIA to monitor for user inputs and changes while maintaining a far simpler codebase. This is done by leveraging Parameters in special Parameterized classes, which provide features like type-checking and default values. THEIA leverages these classes to define acceptable input types and values as well as handle errors without large amounts of boilerplate code.

### *3.3. Holoviews*

HoloViews [3] is an open-source library built for data analysis and visualization with small amounts of code. This library is the backend that provides many of the plots in THEIA. HoloViews takes a different approach to plotting than many other popular libraries. Instead of focusing purely on visualization, HoloViews plots are defined in a way that supports analysis in addition to visualization. This is done by coupling raw data with metadata in a way that allows for immediate visualization rendering as data changes. HoloViews also leverages some of Bokeh's capabilities when rendering specific plots.

### *3.4. Plotly Express*

Plotly Express [4] is a high-level interface to Plotly. It's built to produce easily styleable figures with minimal code. HoloViews leverages this library within THEIA to render supplementary plots like the parallel plot. Plotly Express is an optimal choice because it often provides highly interactive plots by default.

## 4. Results

Together, these scientific Python libraries allowed the Rapid Application Development team at ERDC to develop a robust visualization tool capable of running in the browser without being reliant on an internet connection. Additionally, using THEIA requires no coding. Users simply upload a CSV file using the built-in navigator, and THEIA does the rest. The following screenshots show THEIA's capabilities using the iris dataset.

THEIA also allows user to select from various plot types and color themes, which is a vital feature when working with tradespace data because it allows the user to choose the theme which best distinguishes the complex data.

THEIA also allows users to easily save plots for future reference, export findings for further analysis, and utilize multiple data files simultaneously.

Ultimately, THEIA serves as an excellent example of how open-source libraries can be leveraged to develop robust solutions to emerging problems.

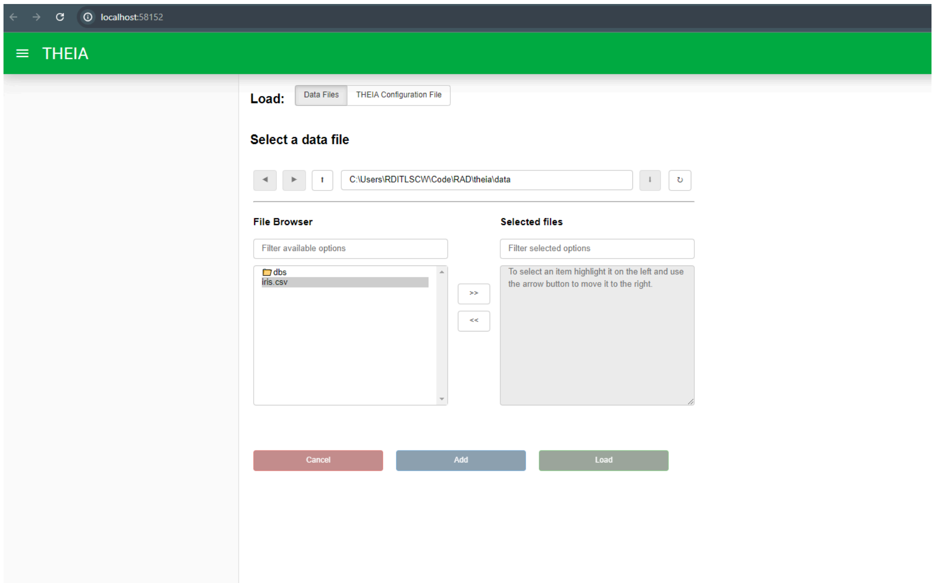
## References

[1] Philipp Rudiger, "holoviz/panel: Panel." [Online]. Available: https://doi.org/10.5281/zenodo.3706648

[2] Philipp Rudiger, "holoviz/param: Param." [Online]. Available: https://doi.org/10.5281/zenodo.11046027

[3] Philipp Rudiger, "holoviz/holoviews: holoviews." [Online]. Available: https://doi.org/10.5281/zenodo.10653612

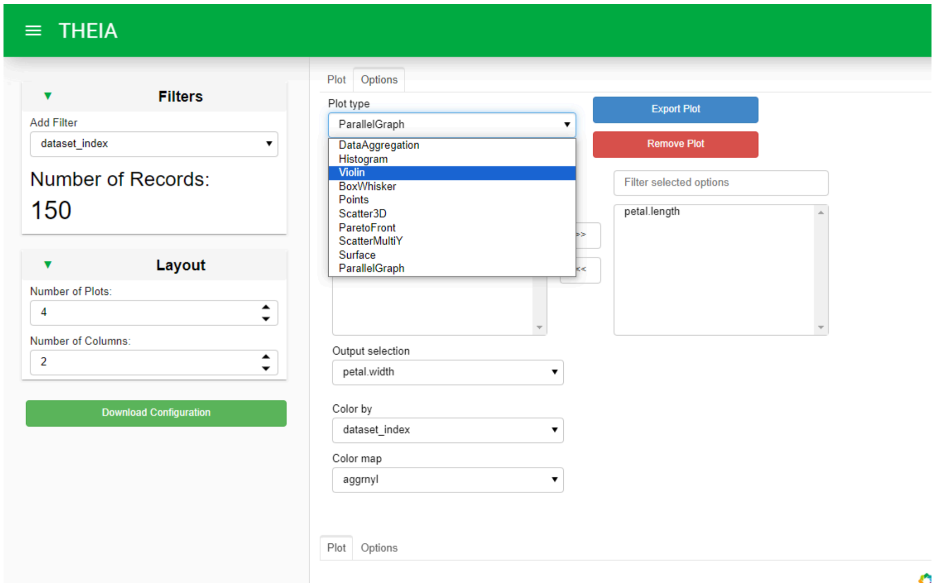
[4] Plotly Technologies Inc., "Collaborative data science." [Online]. Available: https://plot.ly/



C:\Users\RDITLSCW\Code\RAD\theia\data

To select an item highlight it on the left and use the arrow button to move it to the right.

**Figure 2.** *View of THEIA's file browser.*



Add Filter: dataset_index

Number of Records: 150

Number of Plots: 4

Number of Columns: 2

Plot type: ParallelGraph

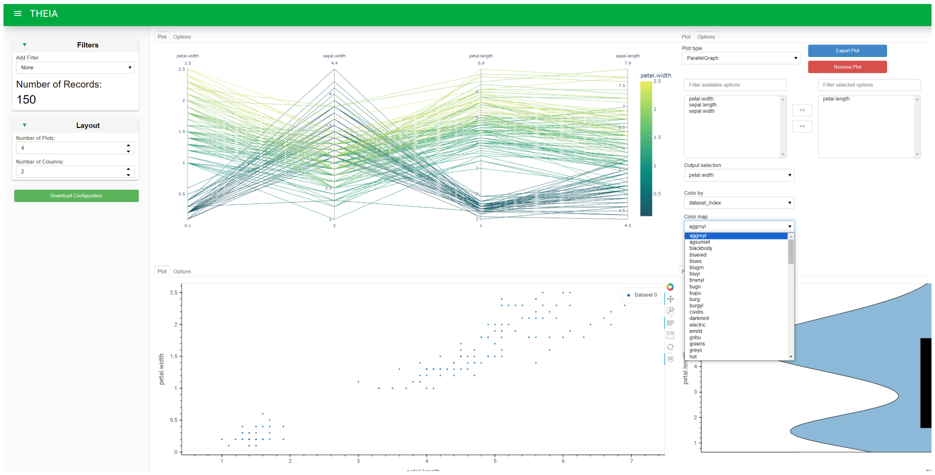
petal.length

Output selection: petal.width

Color by: dataset_index

Color map: aggrnyl

**Figure 3.** *View of THEIA's plot type options.*



Add Filter: None

Number of Records: 150

Number of Plots: 4

Number of Columns: 2

Plot type: ParallelGraph

petal.width
sepal.length
sepal.width

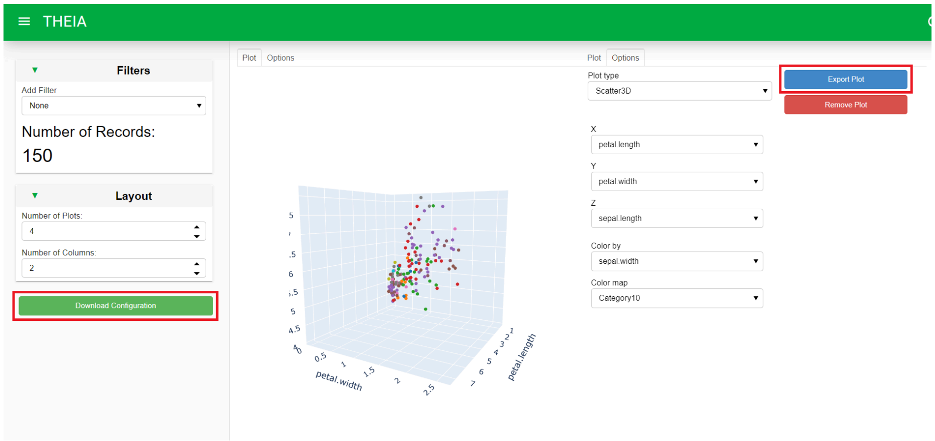
petal.length

Output selection: petal.width

Color by: dataset_index

Color map: aggrnyl

**Figure 4.** *View of THEIA's colormap options.*



Add Filter: None

Number of Records: 150

Number of Plots: 4

Number of Columns: 2

Plot type: Scatter3D

X: petal.length

Y: petal.width

Z: sepal.length

Color by: sepal.width

Color map: Category10

**Figure 5.** *A view of the 'Download Configuration' and 'Export Plot' options. "Download Configuration" can be used to save a YAML file that THEIA can leverage to return to a given state. "Export Plot" is used to save an HTML version of a specific plot.*

Open Access | https://doi.org/10.25080/JXDK4427

SciPy 2024

July 8 - July 14, 2024

Proceedings of the 23nd Python in Science Conference

ISSN: 2575-9752

# Echodataflow: Recipe-based Fisheries Acoustics Workflow Orchestration

**Valentina Staneva**[1], **Soham Butala**[1], **Landung (Don) Setiawan**[2], and **Wu-Jung Lee**[3]

[1]eScience Institute, University of Washington, [2]Scientific Software Engineering Center, University of Washington, [3]Applied Physics Laboratory, University of Washington

## Abstract

With the influx of large data from multiple instruments and experiments, scientists are wrangling complex data pipelines that are context-dependent and non-reproducible. We demonstrate how we leverage Prefect [1], a modern orchestration framework, to facilitate fisheries acoustics data processing. We built a Python package Echodataflow [2] which 1) allows users to specify workflows and their parameters through editing text “recipes” which provide transparency and reproducibility of the pipelines; 2) supports scaling of the workflows while abstracting the computational infrastructure; 3) provides monitoring and logging of the workflow progress. Under the hood, Echodataflow uses Prefect to execute the workflows while providing a domain-friendly interface to facilitate diverse fisheries acoustics use cases. We demonstrate the features through a typical ship survey data processing pipeline.

**Keywords** prefect, workflow orchestration, dask, zarr, fisheries acoustics

## 1. Motivation

Acoustic fisheries surveys and ocean observing systems collect terabytes of echosounder (water column sonar) data that require custom processing pipelines to obtain the distributions and abundance of fish and zooplankton in the ocean [3]. The data are collected by sending an acoustic signal into the ocean which scatters from objects in the water column and the returning “echo” is recorded. Although data usually have similar dimensions: range, time, location, and frequency, and can be stored into multi-dimensional arrays, the exact format varies based on the data collection scheme and the exact instrument used. Fisheries ship surveys, for example, follow pre-defined paths (transects) and can span several months (Figure 1 left). Ocean moorings, on the other hand, have instruments at fixed locations and can collect data continuously at specified intervals for months (Figure 1 right). Uncrewed Surface Vessels (USVs) (e.g. Saildrone [4], DriX [5], Figure 1 middle) can autonomously collect echosounder data over large spatial regions. In all these scenarios, data are usually collected with similar instruments, and there is an overlap between the initial processing procedures. However, there are always variations associated with the specific data collection format, end research needs, data volume, and available computational infrastructure. For example, ship surveys may require grouping data along individual transects and exluding other data; they may also have varying range/depth resulting into data arrays of different dimensions. Mooring data are more regular, but their volume is large, and studies may require organizing data into daily patterns to analyze long term trends. USVs collect data at varying speeds thus requiring converting the time dimension to distance in order to have consistent echo patterns. The time when the data needs to be processed also affects the workflows: on premise/realtime applications usually require processing small data subsets at a time with limited computing resources; historical

**Published** Jul 10, 2024

**Correspondence to**
Valentina Staneva
vms16@uw.edu

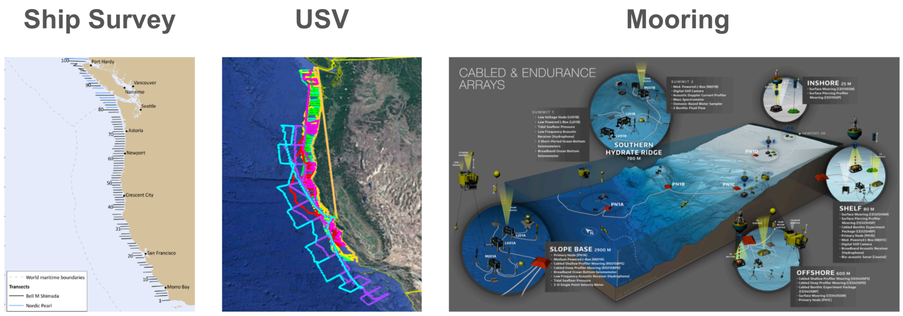


**Figure 1**. ***Data Collection Schemes:*** *left, ship survey transect map for the Joint U.S.-Canada Integrated Ecosystem and Pacific Hake Acoustic Trawl Survey [6] middle, USV path map for Saildrone west coast survey [7] right, map and instrument diagram for a stationary ocean observing system (Ocean Observatories Initiative Cabled and Endurance Arrays [8], Image Credit: Center for Environmental Visualization, University of Washington)*

analyses require processing large datasets, and can benefit from cluster/cloud computing. The various scenarios demand different data workflows, and adapting from one setting to another is not trivial.

## 2. Fisheries Acoustics Workflows

Fisheries acoustics scientists traditionally have had go-to tools and procedures for their data processing and analysis, mostly relying on computation on a local computer. However, as the diversity of computing and data storage resources grows and the field becomes more interdisciplinary (it involves scientists with backgrounds in physics, biology, oceanography, acoustics, signal processing, machine learning, software engineering, etc.), it is becoming more challenging to make decisions on the best arrangement to accomplish the work. For example, Figure 2 shows the many variations of workflows that can be defined based on the use cases and the options for data storage and computing infrastructure.

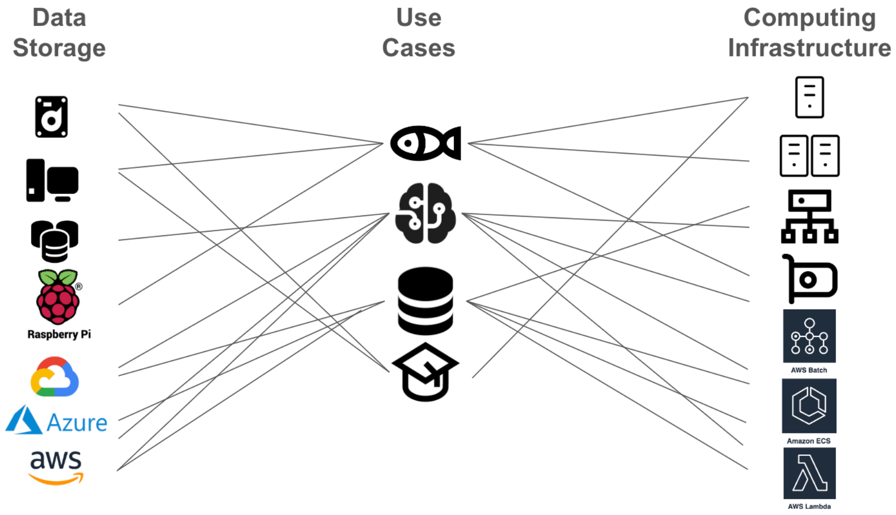


**Figure 2**. ***Fisheries Acoustics Workflow Variations:*** *Various use cases (fisheries, data management, machine learning, education) drive different needs for data storage and computing infrastructure. Options are abundant but adopting new technology and adapting workflows across use cases is not trivial.*

### *2.1. User Stories*

To demonstrate the software requirements of the fisheries acoustics community, below we describe several example user stories.

A **fisheries scientist** needs to process all data after a 2-month ship survey to obtain fish biomass estimates. They have previously used a commercial software package and are open to exploring open-source tools to achieve the same goal. They are familiar with basic scripting in Python.

A **machine learning engineer** is developing an ML algorithm to automatically detect fish on a USV. They need to prepare a large dataset for training but do not know all the necessary preprocessing steps. They are very familiar with ML libraries but do not have the domain-specific knowledge for acoustic data processing. They are also not familiar with distributed computing libraries.

A **data manager** wants to process several terabytes of mooring data and serve them to the scientific community. They have a few Python scripts to do this for a small set of files at a time, but want to scale the processing for many deployments using a cloud infrastructure.

An **acoustics graduate student** obtained echosounder data analysis scripts from a retired scientist but does not have all the parameters needed to reproduce the results in order to proceed with their dissertation research.

We draw attention to the different levels of experience of these users: each user has expertise in a subdomain, however, to accomplish their specific goal(s), they need to learn new tools or obtain knowledge from others. We outline several requirements that stem from these stories:

- The system should run both on a local computer and within a cloud environment.
- The system should allow processing to be scaled to large datasets, but should not be overly complicated. For example, users with Python scripting experience can run it locally with pre-defined stages and parameters.
- The system should provide visibility into the operations that are applied to the data, and the procedures should be interpretable to users without acoustics expertise.
- The system should preferably be free and open source so that it is accessible to members of different institutions.
- The system should adapt to rapid changes of cloud and distributed computing libraries, and preferably should leverage existing developments within the technical communities.

### *2.2. Software Landscape*

Traditionally echosounder data processing pipelines are executed within a GUI-based software (e.g. Echoview [9], LSSS [10], ESP3 [11], Matecho [12]). These software packages have been invaluable for onboard real-time visualization, as well as post-survey data screening and annotation. Some of them also support integration with scripting tools which facilitates the reproducible execution of the pipelines. For example, the Echoview software provides the option to automate pipelines through an Automation Module and to visualize the processing stages in a Dataflow Toolbox. Further, one can script operations through the `echoviewR` package [13]. However, since Echoview is neither free nor open source, these pipelines cannot be shared with researchers who do not have a license. In general, the GUI tools are usually designed to be used on a desktop computer and require downloading the data first, which is becoming challenging with the growing volume of the datasets. There has been also growth in development of new methods to detect the species of interest from the echosounder data, with the goal of substituting for the manual annotation procedures and making analysis of large datasets more efficient and objective. However, the new methods are typically developed independently from the existing software packages. Over the last several years there has been substantial development of open source Python packages (PyEchoLab [14], `echopype` [15], `echopy` [16]), each providing common echosounder processing functionalities, but differing in the data structure organization and processing. Since echosounder instruments store the data in binary, instrument-specific formats, the first stage requires parsing the raw data into a more common format. PyEcholab converts the data into `numpy` [17] arrays. `echopy` expects data are already parsed into `numpy` arrays and all methods operate on them. Echopype converts raw data files into a standardized Python `EchoData` object, which can be stored in a `zarr` [18] format and supports distributed computing by utilizing `dask` [19] and `xarray` [20]. The use of open source packages and well-established formats allow further integration with other open source libraries such as those for machine learning (e.g. classification, clustering) or visualization. In addition, if custom modification is required for a specific application scenario, researchers can adapt the code and contribute the modification back to the packages, which is likely to benefit other researchers.

### 2.2.1. *Challenges:*

Despite the availability of methods and tools to process echosounder data, it is not trivial to orchestrate all function calls in an end-to-end pipeline. While a well-documented Jupyter [21] notebook can show the sequence of processing stages, a considerable amount of path and parameter configuration is required to execute these stages on a large dataset, store the intermediate data products, and log the process comprehensively. Although automation can be achieved through a combination of Python and bash scripts that provision the environment, execute the stages, and manage inputs/outputs, the configuration process can be tedious, prone to error, and specific to the use case and the computing platform. Adapting an existing procedure to a new setting is usually not straightforward, and sometimes even reproducing previous results can pose a challenge. Below we discuss in more detail the different choices of data storage and computational infrastructure and the associated challenges of building workflows across them.

#### *Data Storage:*

Researchers are faced with decisions of where to store the data from experiments, intermediate products, and final results. Initially, data are usually stored on local hard drive storage associated with the instrument (which on some platforms may have limited capacity), but eventually, these data may be transferred to a data archive if one is maintained within the community. Some agencies (e.g. NOAA National Centers for Environmental Information (NCEI) [22]) have adopted cloud storage, and have publicly shared their data, which greatly facilitates data access and reuse. However, those repositories are usually not where researchers can store processed products. Funding models and organizational structure can result in short-term availability of resources and the need to change providers. Certain applications may need to access the data before they are archived and unreliable internet connection may require storing the data on-premise or at temporary locations. *To be agile to those frequent changes and allow to easily switch between different platforms, workflows will benefit from a level of abstraction from storage systems.*

#### *Computing Infrastructure:*

With the growth of the echosounder datasets, researchers face challenges processing the data on their personal machines: both in terms of memory usage and computational time. A typical first attempt for resolution would be to amend the workflow to process smaller chunks of the data and parallelize operations across multiple cores if available.

However, today researchers are also presented with a multitude of options for distributed computing: high-performance computing clusters at local or national institutions, cloud provider services: batch computing (e.g. Azure Batch, AWS Batch, Google Cloud Batch), container services (e.g. Amazon Elastic Container Services, Azure Container Apps, Google Kubernetes Engine), serverless functions (e.g. AWS Lamdba Functions, Google Cloud Functions, Microsoft Azure Functions). The choice may be driven by the storage system: its usage fees and retrieval speeds. Data, code and workflow organization usually has to be adapted based on the computing infrastructure. The knowledge required to configure these systems to achieve efficient processing is quite in-depth, and distributed libraries can be hard to debug and can have unexpected performance bottlenecks. *Abstracting the computing infrastructure and the execution of the tasks can allow researchers to focus on the scientific analysis of these large and rich datasets.*

## 3. Echodataflow Overview

At the center of `echodataflow`'s design [2] is the notion that a workflow can be configured through a set of recipes (`.yaml` files) that specify the pipeline, data storage, and logging structure. The idea draws inspiration from the Pangeo-Forge Project which facilitates the Extraction, Transformation, Loading (ETL) of earth science geospatial datasets from traditional repositories to analysis-ready, cloud-optimized (ARCO) data stores [23]. The pangeo-forge recipes (which themselves are inspired by the conda-forge recipes [24]) provide a model of how the data should be accessed and transformed, and the project has garnered numerous recipes from the community.

While Pangeo-Forge's focus is on transformation from `netcdf` [25] and `hdf5` [26] formats to `zarr`, `echodataflow`'s aim is to support full echosounder data processing and analysis pipelines: from instrument-generated raw data files to data products which contain acoustically-derived biological estimates, such as abundance and biomass. `echodataflow` leverages Prefect [1] to abstract data and computation management. In Figure 3 we provide an overview of `echodataflow`'s framework. At the center we see several steps of an echosounder data processing pipeline: `open_raw`, `combine_echodata`, `compute_Sv`, `compute_MVBS`, `frequency_differencing`, which produce echo classificaton results using a simple threshold-based criterion. All these functions exist in the `echopype` package, and are wrapped by `echodataflow` into pre-defined stages. Prefect executes the stages on a Dask cluster which can be started locally or can be externally set up. These `echopype` functions already support distributed operations with Dask, and thus the integration with Prefect within `echodataflow` is natural. Dask clusters can be set up on a variety of platforms: local computers, cloud virtual machines, kubernetes [27] clusters, or HPC clusters (via `dask-jobqueue` [28]), etc. and allow abstraction from the computing infrastructure. The input datasets, intermediate data products, and final data products can live in different storage systems (local/cloud) and Prefect's block feature provides seamless, provider-agnostic, and secure integration with them. Workflows can be executed and monitored through Prefect's dashboard, while logging of each function is handled by `echodataflow`.

### 3.1. *Why Prefect?*

We chose Prefect among other Python workflow orchestration frameworks such as Apache Airflow [29], Dagster [30], Argo [31], Luigi [32]. While most of these tools provide flexibily and level of abstraction suitable for executing fisheries acoustics pipelines, we selected Prefect for the following reasons:

- Prefect accepts dynamic workflows which are specified at runtime and do not require to follow a Directed Acyclic Graph, which can be restricting and difficult to implement.

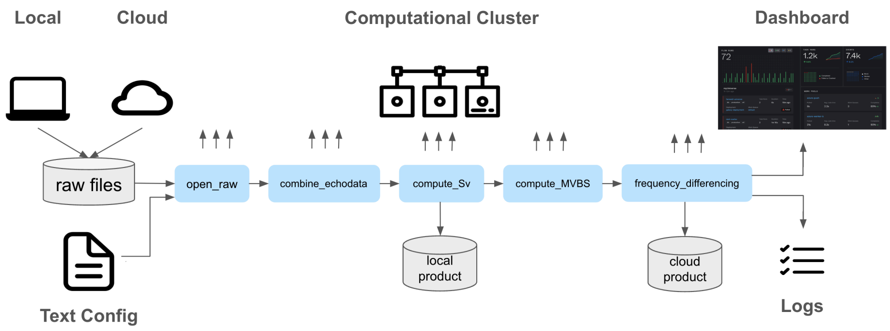


**Figure 3**. ***Echodataflow Framework:*** *The above diagram provides an overview of the `echodataflow` framework: the objective is to fetch raw files from a local filesystem/cloud archive, process them through several stages of an echosounder data workflow using a cluster infrastructure, and store both intermediate and final data products. In `echodataflow` the workflow is executed based on text configurations, and logs are generated for the individual processing stages. Prefect handles the execution of the tasks on the cluster and provides tools for monitoring the workflow runs.*

- In Prefect, Python functions are first class citizens, thus building a Prefect workflow does not deviate substantially from traditional science workflows composed of functions.
- Prefect integrates with a `dask` cluster, and `echopype` processing functions are already using `dask` to scale operations.
- Prefect's code runs similarly locally as well as on cloud services.
- Prefect's monitoring dashboard is open source, can be run locally, and is intuitive to use.

We next describe in more detail the components of the workflow lifecycle.

## 4. Workflow Configuration

The main goal of `echodataflow` is to allow users to configure an echosounder data processing pipeline through editing configuration "recipe" templates. `echodataflow` can be configured through three templates: `datastore.yaml` which handles the data storage locations, `pipeline.yml` which specifies the processing stages, and `logging.yaml` which sets the logging format.

### *4.1. Data Storage Configuration*

Below we show an example file `datastore.yaml` with a data storage configuration for a ship survey. In this scenario the goal is to process data from the Joint U.S.-Canada Integrated Ecosystem and Pacific Hake Acoustic Trawl Survey [6] which are publicly available on an AWS S3 bucket hosted by NOAA National Centers for Environmental Information Acoustics (NCEA) Archive [22]. The archive contains data from many surveys dating back to 1991 ($\sim$ 280TB). The configuration allows to pass parameters specifying the ship, survey, and sonar model names and select the subset of files belonging only to the survey of interest. The output destination is set to a private S3 bucket belonging to the user (within an AWS account different from the input one), and the credentials are passed through a `block_name`. The survey contains $\sim$4000 files, and one can set the group option to combine the files into survey-specific groups: based on the transect information provided in the `transect_group.txt` file. One can further use regular expressions to subselect other subgroups based on needs.

```yaml
# datastore.yaml

name: Bell_M._Shimada-SH1707-EK60
sonar_model: EK60
raw_regex: (.*)-?D(?P<date>\w{1,8})-T(?P<time>\w{1,6})
args:
  urlpath: s3://ncei-wcsd-archive/data/raw/{{ ship_name }}/{{ survey_name }}/{{ sonar_model }}/*.raw
  parameters:
    ship_name: Bell_M._Shimada
    survey_name: SH1707
    sonar_model: EK60
  storage_options:
    anon: true
  group:
    file: ./transect_group.txt
    storage_options:
      block_name: echodataflow-aws-credentials
      type: AWS
  group_name: default_group
  json_export: true
  raw_json_path: s3://echodataflow-workground/combined_files/raw_json
output:
  urlpath: <YOUR-S3-BUCKET>
  overwrite: true
  retention: false
  storage_options:
    block_name: echodataflow-aws-credentials
    type: AWS
```

### *4.2. Pipeline Configuration*

The pipeline configuration file's purpose is to list the stages of the processing pipeline and the computational set-up for their execution. Below we show an example `pipeline.yaml` file which cofigures a pipeline with several stages: `open_raw`, `combine_echodata`, `compute_Sv`, `compute_MVBS`. Each stage is executed as a separate Prefect subflow (a component of a Prefect workflow), and one can specify additional options on whether to store the raw files. `echodataflow` requires access to a Dask cluster: it can be either created on the fly by setting the `use_local_dask` to `true`, or an IP address of an already running cluster can be provided. Individual stages may require different cluster configurations to efficiently execute the tasks. Those can be specified with the additional `prefect_config` option through which the user can set a specific Dask task runner or the number of retries. Managing retries is essential for handling transient failures, such as connectivity issues, ensuring the stages can be re-executed without any manual interference if a failure occurs.

```yaml
# pipeline.yaml

active_recipe: standard
use_local_dask: true
n_workers: 4
scheduler_address: tcp://127.0.0.1:61918
pipeline:
  - recipe_name: standard
    stages:
      - name: echodataflow_open_raw
        module: echodataflow.stages.subflows.open_raw
        options:
          save_raw_file: true
          use_raw_offline: true
          use_offline: true
        prefect_config:
          retries: 3
      - name: echodataflow_combine_echodata
        module: echodataflow.stages.subflows.combine_echodata
        options:
          use_offline: true
      - name: echodataflow_compute_Sv
        module: echodataflow.stages.subflows.compute_Sv
        options:
          use_offline: true
      - name: echodataflow_compute_MVBS
        module: echodataflow.stages.subflows.compute_MVBS
        options:
          use_offline: true
        external_params:
          range_meter_bin: 20
          ping_time_bin: 20S
```

### 4.3. *Logging Configuration*

By default, the outcomes of each stage are logged. The logs can be stored in `.json` or plain text files, and the format of the entries can be specified in the configuration file as displayed below. The `json` format allows searching through the logs for a specific key.

```yaml
# logging.yaml

version: 1
disable_existing_loggers: False
formatters:
  json:
    format: '[%(asctime)s] %(process)d %(levelname)s %(mod_name)s:%(func_name)s:%(lineno)s -
%(message)s'
  plaintext:
    format: '[%(asctime)s] %(process)d %(levelname)s %(mod_name)s:%(func_name)s:%(lineno)s -
%(message)s'
handlers:
  logfile:
    class: logging.handlers.RotatingFileHandler
    formatter: plaintext
    level: DEBUG
    filename: echodataflow.log
    maxBytes: 1000000
    backupCount: 3

loggers:
  echodataflow:
    level: DEBUG
    propagate: False
    handlers: [logfile]
```

In this case the logs are stored in the plain text file `echodataflow.log`. Below we show an example of output logs.

```
[2024-06-06 17:32:08,945] 51493 ERROR apply_mask.py:EK60_SH1707_Shimada2_applymask.zarr:147 - Computing
apply_mask
[2024-06-06 17:32:08,946] 51493 ERROR file_utils.py:file_utils:147 - Encountered Some Error in
EK60_SH1707_Shimada0
[2024-06-06 17:32:08,946] 51493 ERROR file_utils.py:file_utils:147 - 'source_ds' must have coordinates
'ping_time' and 'range_sample'!
[2024-06-06 17:32:08,946] 51493 ERROR file_utils.py:file_utils:147 - Encountered Some Error in
EK60_SH1707_Shimada1
[2024-06-06 17:32:08,946] 51493 ERROR file_utils.py:file_utils:147 - 'source_ds' must have coordinates
'ping_time' and 'range_sample'!
[2024-06-06 17:32:08,946] 51493 ERROR file_utils.py:file_utils:147 - Encountered Some Error in
EK60_SH1707_Shimada2
[2024-06-06 17:32:08,946] 51493 ERROR file_utils.py:file_utils:147 - 'source_ds' must have coordinates
'ping_time' and 'range_sample'!
```

In Section 7, we provide more information on logging options.

## 5. Workflow Execution

To convert a scientific pipeline into an executable Prefect workflow, one needs to organize its components into flows, sublfows, and tasks (the key objects of Prefect's execution logic). Usually, the stages of a pipeline are organized into flows and subflows, while the individual pieces of work within the stage are organized into tasks. In practice, flows, subflows, and tasks are all Python functions, and they differ in how we want to execute them (e.g. concurrently/sequentially, w/o retries), and what we want to track during execution (e.g. input/outputs, state logging, etc.). In `echodataflow` we organize the typical echosounder processing stages into subflows (flows within the main workflow), while the operations on different files (or groups of them) are individual tasks. We describe how functions are organized in the `open_raw` stage, which reads the files from raw format, parses the data, and writes them into a `zarr` format. The `echodataflow_open_raw` function is decorated as a flow, and is one of many subflows of the full workflow. This function processes all files.

```python
@flow
@echodataflow(processing_stage="Open-Raw", type="FLOW")
def echodataflow_open_raw(
    groups: Dict[str, Group], config: Dataset, stage: Stage, prev_stage: Optional[Stage]
):
    """
    Process raw sonar data files and convert them to zarr format.

    Args:
        config (Dataset): Configuration for the dataset being processed.
        stage (Stage): Configuration for the current processing stage.
        prev_stage (Stage): Configuration for the previous processing stage.

    Returns:
        List[Output]: List of processed outputs organized based on transects.
```

`echodataflow_open_raw` contains a loop which iterates through all file groups and applies the `process_raw` function which operates on a single group and is decorated as a task. All tasks will be executed on the Dask cluster.

```
for name, gr in groups.items():
        for raw in gr.data:
            new_processed_raw = process_raw.with_options(
                task_run_name=raw.file_path, name=raw.file_path, retries=3
            )
            future = new_processed_raw.submit(raw, gr, working_dir, config, stage)
            futures[name].append(future)
```

```
@task()
@echodataflow()
def process_raw(
    raw: EchodataflowObject, group: Group, working_dir: str, config: Dataset, stage: Stage
):
    """
    Process a single group of raw sonar data files.
```

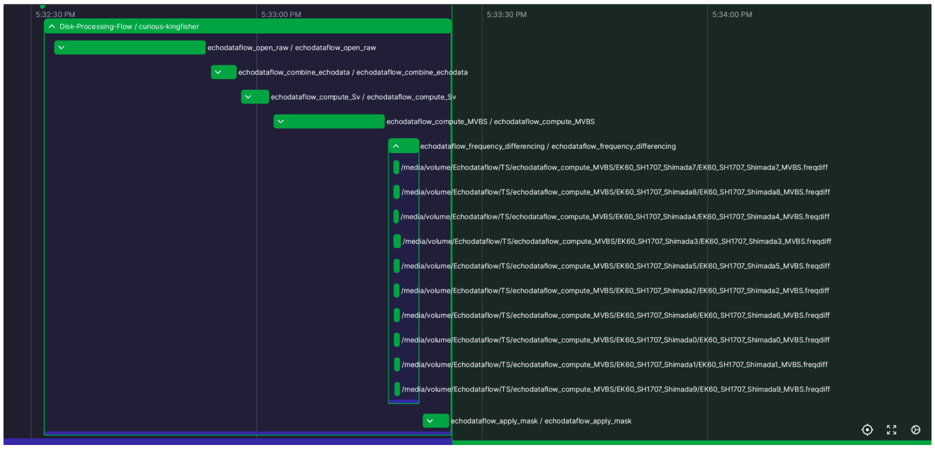
## 6. Workflow Monitoring

One of the main advantages of using orchestration frameworks is that they usually provide tools to monitor the workflow execution. The integration with Prefect allows leveraging Prefect's dashboard (Prefect UI) for monitoring the execution of the flows. The dashboard can be run locally and within Prefect's online managed system (Prefect Cloud). The local version provides an entirely open source framework for running and monitoring workflows. Figure 4 shows the view of completed runs within the dashboard. The progress can be monitored while the flows are in progress.

Further, one can also view the progress of the execution of the tasks on the Dask cluster.

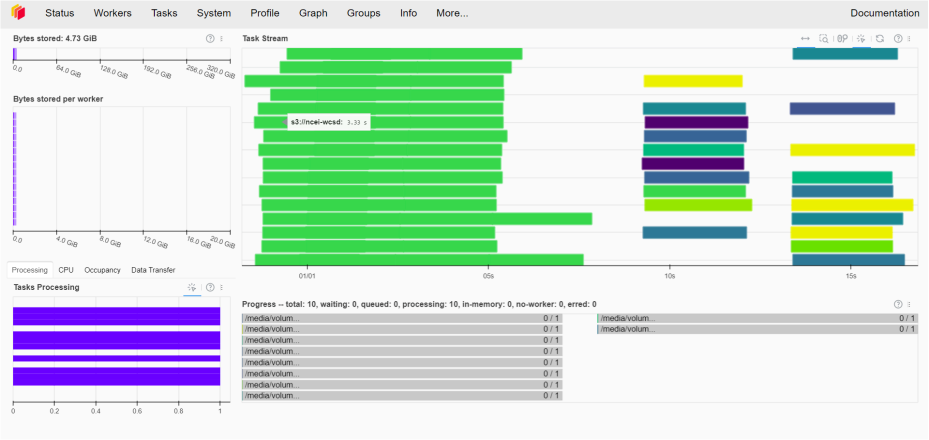
## 7. Workflow Logging

Processing large data archives requires a robust logging system to identify at which step and for which files the processing has failed. Locating the issues allows to set a path forward to resolve them: either through improving the robustness of the individual libraries performing the processing steps, or through identifying the artifacts of the data which are incompatible with the existing pipeline. To address this, we provide several approaches:



```
Disk-Processing-Flow / curious-kingfisher
echodataflow_open_raw / echodataflow_open_raw
echodataflow_combine_echodata / echodataflow_combine_echodata
echodataflow_compute_Sv / echodataflow_compute_Sv
echodataflow_compute_MVBS / echodataflow_compute_MVBS
echodataflow_frequency_differencing / echodataflow_frequency_differencing
/media/volume/Echodataflow/TS/echodataflow_compute_MVBS/EK60_SH1707_Shimada7/EK60_SH1707_Shimada7_MVBS.freqdiff
/media/volume/Echodataflow/TS/echodataflow_compute_MVBS/EK60_SH1707_Shimada8/EK60_SH1707_Shimada8_MVBS.freqdiff
/media/volume/Echodataflow/TS/echodataflow_compute_MVBS/EK60_SH1707_Shimada4/EK60_SH1707_Shimada4_MVBS.freqdiff
/media/volume/Echodataflow/TS/echodataflow_compute_MVBS/EK60_SH1707_Shimada3/EK60_SH1707_Shimada3_MVBS.freqdiff
/media/volume/Echodataflow/TS/echodataflow_compute_MVBS/EK60_SH1707_Shimada5/EK60_SH1707_Shimada5_MVBS.freqdiff
/media/volume/Echodataflow/TS/echodataflow_compute_MVBS/EK60_SH1707_Shimada2/EK60_SH1707_Shimada2_MVBS.freqdiff
/media/volume/Echodataflow/TS/echodataflow_compute_MVBS/EK60_SH1707_Shimada6/EK60_SH1707_Shimada6_MVBS.freqdiff
/media/volume/Echodataflow/TS/echodataflow_compute_MVBS/EK60_SH1707_Shimada0/EK60_SH1707_Shimada0_MVBS.freqdiff
/media/volume/Echodataflow/TS/echodataflow_compute_MVBS/EK60_SH1707_Shimada1/EK60_SH1707_Shimada1_MVBS.freqdiff
/media/volume/Echodataflow/TS/echodataflow_compute_MVBS/EK60_SH1707_Shimada9/EK60_SH1707_Shimada9_MVBS.freqdiff
echodataflow_apply_mask / echodataflow_apply_mask
```

**Figure 4.** ***Flow Runs:*** *Log of completed runs in Prefect UI. The stages (subflows) are executed sequentially. One can expand the view of an individual flow and see the tasks computed (asynchronously) within it.*



Progress -- total: 10, waiting: 0, queued: 0, processing: 10, in-memory: 0, no-worker: 0, erred: 0

```
/media/volum...   0 / 1      /media/volum...   0 / 1
/media/volum...   0 / 1      /media/volum...   0 / 1
/media/volum...   0 / 1
/media/volum...   0 / 1
/media/volum...   0 / 1
/media/volum...   0 / 1
/media/volum...   0 / 1
/media/volum...   0 / 1
```

**Figure 5**. ***Dask Dashboard:*** *The execution of the tasks on the Dask cluster can also be monitored through the Dask dashboard.*

- Basic Logging with Dask Worker Streams: this approach configures Dask worker streams to handle `echodataflow` logs, which is straightforward if exact log order is not crucial.
- Centralized Logging with Amazon CloudWatch [33]: this approach centralizes all logs for easy access and analysis. It can be useful when users are already utilizing AWS.
- Advanced Logging with Apache Kafka [34] and Elastic Stack [35] (Elasticsearch, Kibana, Beats, Logstash): this approach leverages Kafka for log aggregation and Elastic Stack for log analysis and visualization, offering a robust solution for those who can maintain the infrastructure, for example data center managers.

By default if logging is not configured, all the worker messages are directed to the application console. The order of logs may not be preserved since logs are written once control returns from the Dask workers to the main application.

## 8. Workflow Deployment

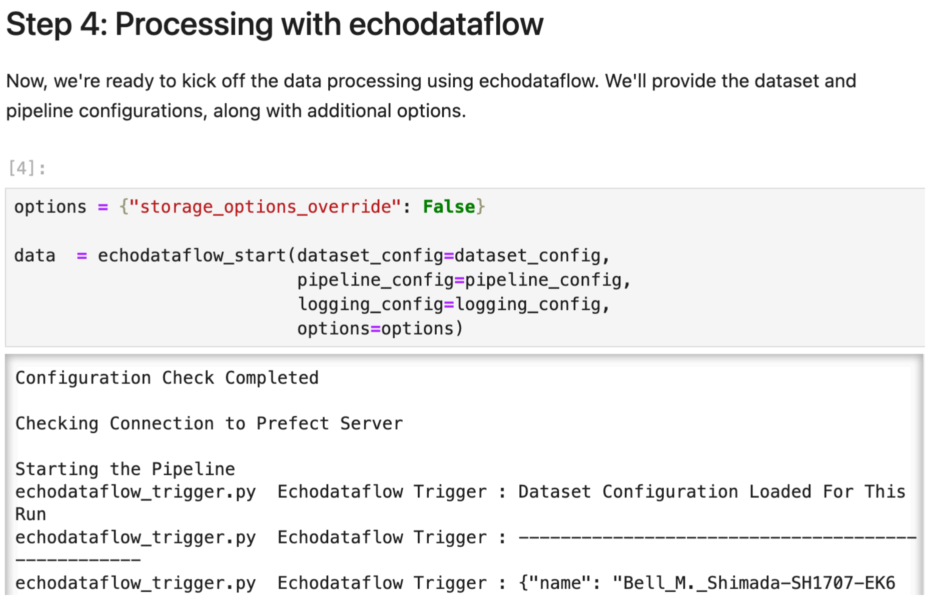
### 8.1. *Notebook*

`echodataflow` can be directly initiated within a Jupyter notebook, which makes development interactive and provides a work environment familiar to researchers. One can see how the workflow is initiated within the Jupyter cell in Figure 6.

We provide two demo notebooks: one for execution on a local machine and another one for execution on AWS.

### 8.2. *Docker*

We facilitate the deployment of `echodataflow` on various platforms by building a Docker image from which one can launch a container with all required components and the user can access the workflow dashboard on the corresponding port.



## Step 4: Processing with echodataflow

Now, we're ready to kick off the data processing using echodataflow. We'll provide the dataset and pipeline configurations, along with additional options.

[4]:

```
options = {"storage_options_override": False}

data  = echodataflow_start(dataset_config=dataset_config,
                           pipeline_config=pipeline_config,
                           logging_config=logging_config,
                           options=options)
```

```
Configuration Check Completed

Checking Connection to Prefect Server

Starting the Pipeline
echodataflow_trigger.py  Echodataflow Trigger : Dataset Configuration Loaded For This
Run
echodataflow_trigger.py  Echodataflow Trigger : ----------------------------------------
------------
echodataflow_trigger.py  Echodataflow Trigger : {"name": "Bell_M._Shimada-SH1707-EK6
```

**Figure 6**. ***Initiating `echodataflow` in a Jupyter Notebook:*** *Once one has a set of "recipe" configuration files, they can initiate the workflow in a notebook cell with the `echodataflow_start` command.*

```
docker pull blackdranzer/echodataflow

prefect server start

docker run --network="host" -e PREFECT_API_URL=http://host.docker.internal:4200/api blackdranzer/
echodataflow
```

Upon execution, the user can readily access the Prefect UI dashboard and run workflows from there.

We also provide a Docker image for initiating logging with Kafka and Elastic Stack, thus streamlining the configuration of several tools.

## 9. Command Line Interface

We provide a command line interface which supports credential handling, and several additional features for managing workflows: stage addition and rule validation.

### *9.1. Adding Stages*

Currently, most major functionalities in the echopype package are wrapped into stages: `open_raw`, `add_depth`, `add_location`, `compute_Sv`, `compute_TS`, `compute_MVBS`, `combine_echodata`, `frequency_differencing`, `apply_mask`.

We provide tools to generate boilerplate template configuration based on the existing stages. Here is an example to add a stage:

```
echodataflow gs <stage_name>
```

For instance, to generate a boilerplate configuration for the `compute_Sv` stage, one would use:

```
echodataflow gs compute_Sv
```

This command creates a template configuration file for the specified stage, allowing to customize and integrate it into a workflow. The generated file includes:

- a flow: it orchestrates the execution of all files that need to be processed, either concurrently or in parallel, based on the configuration.
- a task (helper function): it assists the flow by processing individual files.

### 9.2. *Rule Validation*

Scientific workflows often have stages that cannot be executed until other stages have completed. Those conditions can be set through `echodataflow` client during the initialization process and are stored in a `echodataflow_rules.txt` file:

```
echodataflow_open_raw:echodataflow_compute_Sv
echodataflow_open_raw:echodataflow_combine_echodata
echodataflow_open_raw:echodataflow_compute_TS
echodataflow_combine_echodata:echodataflow_compute_Sv
echodataflow_compute_Sv:echodataflow_compute_MVBS
```

These rules dictate the sequence in which stages should be executed, ensuring that each stage waits for its dependencies to complete. They can be set through the `echodataflow rules -add ...` command.

#### 9.2.1. *Aspect-Oriented Programming (AOP) for Rule Validation:*

In `echodataflow`, we adopt an aspect-oriented programming [36] approach for rule validation. This is achieved using a decorator that can be applied to functions to enforce rules and log function execution details. The echodataflow decorator logs the entry and exit of a decorated function and modifies the function's arguments based on the execution context. This supports two types of execution: "TASK" and "FLOW".

Example Usage:

```
@echodataflow(processing_stage="StageA", type="FLOW")
def my_function(arg1, arg2):
    # Function code here
    pass
```

In the example, the echodataflow decorator ensures that the function `my_function` is executed within the context of "StageA" as a "FLOW", checking for dependencies and logging relevant information.

## 10. Example Use Case: Processing Ship Survey Data from an Archive

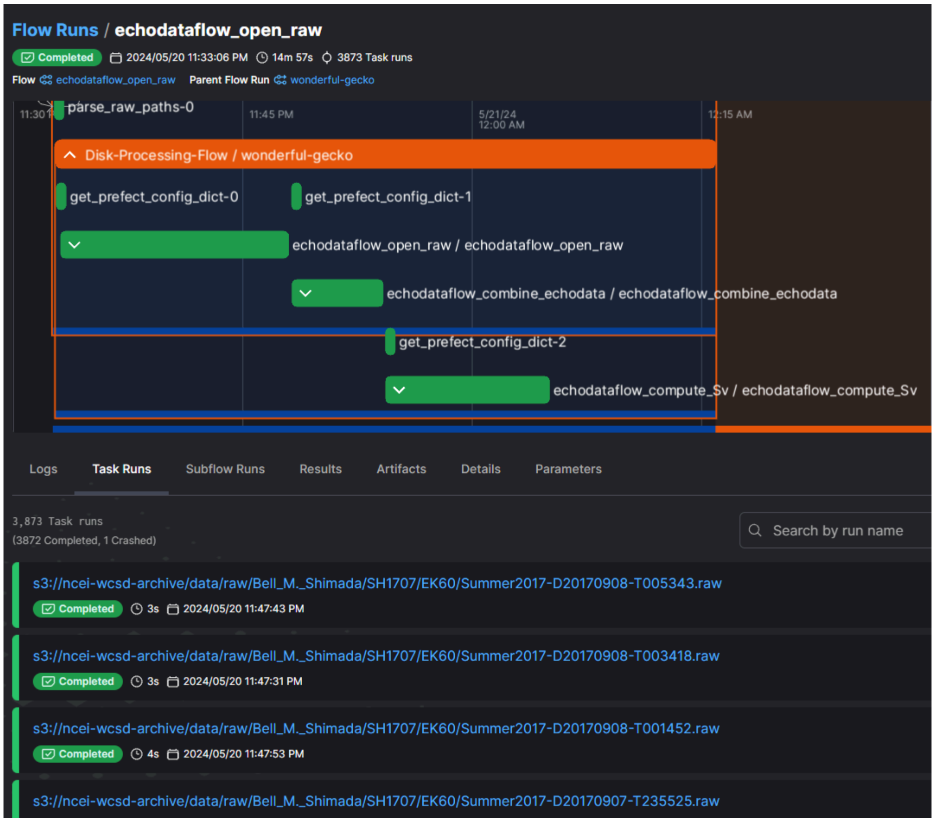
We demonstrate a workflow processing all acoustic data for the 2017 Joint U.S.-Canada Integrated Ecosystem and Pacific Hake Acoustic Trawl Survey through a few routine processing stages. The survey spans a period of 06/15/2017 - 09/13/2017, covering the entire west coast of the US and Canada. Figure 1(a) shows a map of a typical transect schedule of the survey. Raw acoustic data are collected continuously while the ship is in motion, resulting in a total of 3873 files collected with a total size of 93 GB. The raw files are archived by the NOAA NCEI Water Column Sonar Data Archive and are publicly accessible on their Amazon

Web Services S3 bucket (https://registry.opendata.aws/ncei-wcsd-archive/). The processing pipeline involves several steps:

- Convert raw files to cloud-native `zarr` format following closely a community convention [15], [37]
- Combine multiple individual `zarr` files within a continuous transect segment into a single `zarr` file
- Compute Sv: calibrate the measured acoustic backscatter data to volume backscattering strength (Sv, unit: dB re 1 $m^{-1}$)

Once data are converted to Sv, they are easy to manipulate, as the data are stored in an `xarray` data array and are smaller than that of the original data. The final dataset can be served as an analysis-ready data product to the community. It can be beneficial to store also intermediate datasets at different processing stages: for example, preserving the converted raw files in the standardized `zarr` format allows users to regenerate any of the following stages with different groupings or resolution, without having to fetch and convert raw data again.

The execution of the workflow with `echodataflow` allowed us to monitor the progress of all files Figure 7: 3872 files were successfully processed, and 1 failed. Most importantly, the failure did not block the execution of the other files, and a log was generated for the stage and the filename for which the error occurred. This experiment serves as a confirmation that the transition from local development to a full production pipeline with `echodataflow` can indeed be smooth.



Flow Runs / echodataflow_open_raw

☑ Completed 2024/05/20 11:33:06 PM 14m 57s 3873 Task runs

Flow echodataflow_open_raw Parent Flow Run wonderful-gecko

11:30 PM
parse_raw_paths-0
11:45 PM
5/21/24
12:00 AM
12:15 AM
Disk-Processing-Flow / wonderful-gecko
get_prefect_config_dict-0
get_prefect_config_dict-1
echodataflow_open_raw / echodataflow_open_raw
echodataflow_combine_echodata / echodataflow_combine_echodata
get_prefect_config_dict-2
echodataflow_compute_Sv / echodataflow_compute_Sv
Logs
Task Runs
Subflow Runs
Results
Artifacts
Details
Parameters
Search by run name

3,873 Task runs
(3872 Completed, 1 Crashed)

s3://ncei-wcsd-archive/data/raw/Bell_M._Shimada/SH1707/EK60/Summer2017-D20170908-T005343.raw
☑ Completed 3s 2024/05/20 11:47:43 PM

s3://ncei-wcsd-archive/data/raw/Bell_M._Shimada/SH1707/EK60/Summer2017-D20170908-T003418.raw
☑ Completed 3s 2024/05/20 11:47:31 PM

s3://ncei-wcsd-archive/data/raw/Bell_M._Shimada/SH1707/EK60/Summer2017-D20170908-T001452.raw
☑ Completed 4s 2024/05/20 11:47:53 PM

s3://ncei-wcsd-archive/data/raw/Bell_M._Shimada/SH1707/EK60/Summer2017-D20170907-T235525.raw

**Figure 7.** ***Processing full 2017 Survey Data:*** *1/3873 files failed at the `open_raw` stage, but this did not impact the entire pipeline. As shown, other files were processed successfully through all stages.*

## 11. Future Development

Our immediate goal is to provide more example workflow recipes integrating other stages of echosounder data processing, such as machine learning prediction, training dataset generation, biomass estimation, interactive visualization, etc. We will demonstrate utilizing functionalities from a suite of open source Python packages (`echoregions` [38] for reading region annotations and creating corresponding masks, `echopop` [39] for combining acoustic data with biological "ground truth" into biomass estimation, `echoshader` [40] for echogram and map dashboard visualization) in building workflows for the Pacific Hake Survey: both in a historical and near-realtime on-ship data processing context. We aim to streamline the stage addition process. We will further investigate how to improve memory management and caching between and within stages to optimize for different scenarios. There is growing interest in the fisheries acoustics community to share global, accessible, and interoperable datasets [41], and to agree on community data standards and definitions of processing levels [37], [42]. As those mature we will align them with existing stages in `echodataflow`, which will support building interoperable datasets whose integration will push us to study bigger and more challenging questions in fisheries acoustics.

## 12. Beyond Fisheries Acoustics

Echodataflow was designed to facilitate fisheries acoustics workflows, but the structure can be adapted to data processing pipelines in other scientific communities. The key aspects are to identify the potential stages of the workflows and associated Python packages/functions that implement them, and to design the structure of the configuration files. The other aspects such as logging, deployment, monitoring, new-stage integration are domain-agnostic. Processing pipelines that require manipulation of large labeled arrays can directly benefit from the Dask cluster integration and are prevalent in the research community. Our use case of regrouping data based on time segments is a common need within scientific settings in which the file unit level of the instrument is not aligned with the unit level of analysis, and requires further reorganization and potential resampling and regridding along certain coordinates. We hope it can serve as a guide on how to build configurable, reproducible, and scalable workflows in new scientific areas.

### Acknowledgements

We thank the Fisheries Engineering and Acoustic Technologies team at the NOAA Northwest Fisheries Science Center: Julia Clemons, Alicia Billings, Rebecca Thomas, Elizabeth Phillips for introducing us to the Pacific Hake Survey operations and the hake biomass estimation workflow.

This work used cpu compute and storage resources at Jetstream2 through allocation AGR230002 from the Advanced cyberinfrastructure Coordination Ecosystem: Services & Support (ACCESS) program [43], [44], which is supported by National Science Foundation grants #2138259, #2138286, #2138307, #2137603, and #2138296.

**Funding**

NOAA Award No. NA21OAR0110201, NOAA Award No. NA20OAR4320271 AM43, eScience Institute

### References

[1] "Prefect." [Online]. Available: https://www.prefect.io/

[2] "Echodataflow." [Online]. Available: https://github.com/OSOceanAcoustics/echodataflow

[3] NOAA National Center for Environmental Information, “Understanding Our Ocean with Water-Column Sonar Data.” [Online]. Available: https://storymaps.arcgis.com/stories/e245977def474bdba60952f30576908f

[4] “Saildrone.” [Online]. Available: https://www.saildrone.com/

[5] “DriX.” [Online]. Available: https://www.ixblue.com/north-america/maritime/maritime-autonomy/uncrewed-surface-vehicles/

[6] Northwest Fisheries Science Center, Fishery Resource Analysis and Monitoring Division, “The 2021 Joint U.S.-Canada Integrated Ecosystem and Pacific Hake Acoustic Trawl Survey: Cruise Report SH-21-06,” 2022, doi: 10.25923/0979-6D84.

[7] Saildrone, “US/Canada West Coast Fisheries,” 2019. [Online]. Available: https://www.saildrone.com/technology/data-sets/west-coast-fisheries-2019

[8] J. Trowbridge *et al.*, “The Ocean Observatories Initiative,” *Frontiers in Marine Science*, vol. 6, 2019, doi: 10.3389/fmars.2019.00074.

[9] Echoview Software Pty Ltd, “Echoview - Sound Knowledge.” [Online]. Available: https://www.echoview.com/

[10] R. Korneliussen *et al.*, “The Large Scale Survey System - LSSS,” in *Proceedings of the 29th Scandinavian Symposium on Physical Acoustics*, Ustaoset, Norway, 2006. [Online]. Available: https://api.semanticscholar.org/CorpusID:204802910

[11] Y. Ladroit, P. C. Escobar-Flores, A. C. G. Schimel, and R. L. O’Driscoll, “ESP3: An open-source software for the quantitative processing of hydro-acoustic data,” *SoftwareX*, vol. 12, p. 100581, 2020, doi: 10.1016/j.softx.2020.100581.

[12] Y. Perrot *et al.*, “Matecho: An Open-Source Tool for Processing Fisheries Acoustics Data,” *Acoustics Australia*, vol. 46, no. 2, pp. 241–248, 2018, doi: 10.1007/s40857-018-0135-x.

[13] L.-M. K. Harrison, M. J. Cox, G. Skaret, and R. Harcourt, “The R package EchoviewR for automated processing of active acoustic data using Echoview,” *Frontiers in Marine Science*, vol. 2, 2015, doi: 10.3389/fmars.2015.00015.

[14] C. C. Wall, R. Towler, C. Anderson, R. Cutter, and J. M. Jech, “PyEcholab: An open-source, python-based toolkit to analyze water-column echosounder data,” *The Journal of the Acoustical Society of America*, vol. 144, no. 3, p. 1778, 2018, doi: 10.1121/1.5067860.

[15] W.-J. Lee, E. Mayorga, L. Setiawan, I. Majeed, K. Nguyen, and V. Staneva, “Echopype: A Python library for interoperable and scalable processing of water column sonar data for biological information,” *arXiv:2111.00187 [eess]*, 2021, doi: 10.48550/arXiv.2111.00187.

[16] “open-ocean-sounding/echopy.” [Online]. Available: https://github.com/open-ocean-sounding/echopy

[17] C. R. Harris *et al.*, “Array programming with NumPy,” *Nature*, vol. 585, no. 7825, pp. 357–362, 2020, doi: https://doi.org/10.1038/s41586-020-2649-2.

[18] A. Miles *et al.*, “zarr-developers/zarr-python: v2.18.2.” [Online]. Available: https://doi.org/10.5281/zenodo.11320255

[19] Dask Development Team, “Dask: Library for dynamic task scheduling,” 2016. [Online]. Available: http://dask.pydata.org/

[20] S. Hoyer and J. Hamman, “xarray: N-D labeled arrays and datasets in Python,” *Journal of Open Research Software*, vol. 5, no. 1, 2017, doi: 10.5334/jors.148.

[21] T. Kluyver *et al.*, “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., 2016, pp. 87–90.

[22] C. Wall, “Building an Accessible Archive for Water Column Sonar Data.” [Online]. Available: http://dx.doi.org/10.1029/2016EO057595

[23] C. Stern *et al.*, “Pangeo Forge: Crowdsourcing Analysis-Ready, Cloud Optimized Data Production,” *Frontiers in Climate*, vol. 3, 2022, doi: 10.3389/fclim.2021.782909.

[24] conda-forge community, “The conda-forge Project: Community-based Software Distribution Built on the conda Package Format and Ecosystem.” [Online]. Available: https://doi.org/10.5281/zenodo.4774216

[25] R. Rew and G. Davis, “NetCDF: an interface for scientific data access,” *IEEE Computer Graphics and Applications*, vol. 10, no. 4, pp. 76–82, 1990, doi: 10.1109/38.56302.

[26] The HDF Group, “Hierarchical Data Format, version 5.” [Online]. Available: https://github.com/HDFGroup/hdf5

[27] “Kubernetes.” [Online]. Available: https://kubernetes.io/

[28] “Dask JobQueue.” [Online]. Available: https://jobqueue.dask.org/en/latest/

[29] “Apache Airflow.” [Online]. Available: https://airflow.apache.org/

[30] “Dagster.” [Online]. Available: https://dagster.io/

[31] “Argo Workflows,” 2024. [Online]. Available: https://github.com/argoproj/argo-workflows

[32] “Luigi.” [Online]. Available: https://luigi.readthedocs.io/en/stable/running_luigi.html

[33] “Amazon Cloudwatch.” [Online]. Available: https://aws.amazon.com/pm/cloudwatch/

[34] "Apache Kafka." [Online]. Available: https://kafka.apache.org/

[35] "Elastic Stack." [Online]. Available: https://www.elastic.co/

[36] G. Kiczales *et al.*, "Aspect-oriented programming," in *ECOOP'97 — Object-Oriented Programming*, M. Aḳsit and S. Matsuoka, Eds., Berlin, Heidelberg, 1997, pp. 220–242.

[37] G. Macaulay and H. Peña, "The SONAR-netCDF4 convention for sonar data, Version 1.0," 2018, doi: 10.17895/ices.pub.4392.

[38] K. Nguyen, C. Tuguinay, V. Staneva, and W.-J. Lee, "OSOceanAcoustics/echoregions: v0.1.0 (Initial Release of Echoregions)." [Online]. Available: https://doi.org/10.5281/zenodo.8400850

[39] B. Lucca, E. Mayorga, b reyes, and W.-J. Lee, "OSOceanAcoustics/echopop: v0.4.0." [Online]. Available: https://doi.org/10.5281/zenodo.11454149

[40] D. Lei, D. Setiawan, B. Reyes, E. Mayorga, W.-J. Lee, and V. Staneva, "OSOceanAcoustics/echoshader: v0.1.0." [Online]. Available: https://doi.org/10.5281/zenodo.10856784

[41] "ICES Working Group on Global Acoustic Interoperable Network (GAIN)." [Online]. Available: https://github.com/ices-eg/wk_WKGAIN

[42] "Echosounder Data Processing Levels." [Online]. Available: https://github.com/OSOceanAcoustics/echolevels

[43] D. Y. Hancock *et al.*, "Jetstream2: Accelerating cloud computing via Jetstream," in *Practice and Experience in Advanced Research Computing 2021: Evolution Across All Dimensions*, in PEARC '21. Boston, MA, USA, 2021. doi: 10.1145/3437359.3465565.

[44] T. J. Boerner, S. Deems, T. R. Furlani, S. L. Knuth, and J. Towns, "ACCESS: Advancing Innovation: NSF's Advanced Cyberinfrastructure Coordination Ecosystem: Services & Support," in *Practice and Experience in Advanced Research Computing 2023: Computing for the Common Good*, in PEARC '23. Portland, OR, USA, 2023, pp. 173–176. doi: 10.1145/3569951.3597559.

Open Access | https://doi.org/10.25080/YADT7194

# Python-Based GeoImagery Dataset Development for Deep Learning-Driven Forest Wildfire Detection

**Valeria Martin**[1], **Derek Morgan**[1], and **K. Brent Venable**[1,2]

[1]University of West Florida, [2]Florida Institute of Human Machine Cognition

*SciPy 2024*

*July 8 - July 14, 2024*

Proceedings of the 23rd
Python in Science Conference
ISSN: 2575-9752

## Abstract

In recent years, leveraging satellite imagery with deep learning (DL) architectures has become an effective approach for environmental monitoring tasks, including forest wildfire detection. Nevertheless, this integration requires substantial high-quality labeled data to train the DL models accurately. Leveraging the capabilities of multiple Python libraries, such as rasterio and GeoPandas, and Google Earth Engine's Python API, this study introduces a streamlined methodology to efficiently gather, label, augment, process, and evaluate a large-scale bi-temporal high-resolution satellite imagery dataset for DL-driven forest wildfire detection. Known as the California Wildfire GeoImaging Dataset (CWGID), this dataset comprises over 100,000 labeled 'before' and 'after' wildfire image pairs, created from pre-existing satellite imagery. An analysis of the dataset using pre-trained and adapted Convolutional Neural Network (CNN) architectures, such as VGG16 and EfficientNet, achieved accuracies of respectively 76% and 93%. The pipeline outlined in this paper demonstrates how Python can be used to gather and process high-resolution satellite imagery datasets, leading to accurate wildfire detection and providing a tool for broader environmental monitoring.

**Keywords** Wildfire Detection, Satellite Imagery, Deep Learning, Convolutional Neural Networks (CNN), California Wildfire GeoImaging Dataset (CWGID), Google Earth Engine (GEE), Sentinel-2, Machine Learning, Environmental Monitoring, Geospatial Data Analysis

## 1. Introduction

This paper presents a Python-based methodology for gathering and using a labeled high-resolution satellite imagery dataset for forest wildfire detection.

Forests are important ecosystems found globally. They are made up of trees, plants, and other various types of vegetation. Forests host many species and are crucial for maintaining environmental health, as they support biodiversity, climate regulation, and oxygen production. Moreover, they bring economic and social benefits, including energy production, job opportunities, and spaces for leisure and tourism. Protecting forests and tackling forest loss is a current global priority [1].

With the development of Earth Observation (EO) systems, remote sensing became a time-efficient and cost-effective method for monitoring and detecting forest change [2]. Moreover, recent advancements in satellite technology have significantly enhanced forest monitoring capabilities by providing high-resolution imagery and increasing the frequency of observations.

Satellite imagery-based change detection and forest monitoring have traditionally relied on manually identifying specific features and using predefined algorithms and models, such as differential analysis, thresholding techniques, and clustering and classification algorithms.

**Published** Jul 10, 2024

**Correspondence to**
Valeria Martin
vm58@students.uwf.edu

This approach requires considerable domain expertise and such algorithms and models may not capture the full complexity of the studied data [3].

With the emergence of deep learning (DL) algorithms, specifically computer vision methods, such as Convolutional Neural Networks (CNNs) [4] and Fully Convolutional Neural Networks (e.g., U-Nets [5]), there is a significant opportunity to enhance and facilitate forest change detection efforts. These advanced computational methods can rapidly identify complex patterns within vast datasets. Furthermore, when integrated with EO systems they can facilitate near real-time monitoring and detection of multiple forest loss causes, assess their extent, or even predict and evaluate their spread [6], [7]. Thus, integrating DL methods with satellite imagery offers a more dynamic and precise approach, capable of handling the patterns and variability associated with imagery data. For instance, DL models can automatically learn complex patterns related to wildfire spread from labeled examples, whereas traditional methods might miss subtle but important indicators.

However, DL algorithms require a substantial amount of labeled data to effectively learn and identify change [8]. Therefore, the development of labeled high-resolution satellite imagery datasets is important and relevant for addressing environmental problems. Currently, the availability of labeled high-quality satellite imagery datasets is an obstacle to developing DL models for environmental change detection [9].

Generally, building a satellite imagery dataset is a time-intensive process. However, Google Earth Engine (GEE) [10] has recently revolutionized this process by providing an extensive, cloud-based platform for the efficient collection, processing, and analysis of satellite imagery. GEE's Python API allows its users to programmatically query their platform and download cloud-free large-scale satellite imagery datasets from multiple satellite collections, such as Sentinel-2.

For example, while traditional methods might require manual search and download of images, GEE can automate this process, significantly reducing the time needed to find suitable satellite images. This technology makes data collection and processing faster and easier, facilitating environmental monitoring by providing reliable and easily accessible high-quality satellite imagery.

Furthermore, Python facilitates the use of DL in environmental monitoring by providing a rich ecosystem of libraries and tools, such as TensorFlow [11], which contains multiple existing DL architectures that can be adapted and used with satellite imagery. Nevertheless, integrating DL and remotely sensed images requires multiple processing steps, such as having smaller imagery tiles and adapting the models to use GeoTIFF data, among others. These steps include:

1. Data Acquisition: Collect satellite imagery from sources such as Google Earth Engine or other satellite image providers.
2. Image Tiling: Divide large satellite images into smaller tiles to fit the input image size used in DL models.
3. Data Annotation: Label the images to create a ground truth dataset for training DL models.
4. Data Augmentation: Applying transformations such as rotations and flips to increase the diversity of the training dataset.

Python's tools help manage these steps. For example, libraries like rasterio, Tifffile and GeoPandas, can be used to process and transform satellite imagery data into formats suitable for DL models.

This paper presents a methodology, implemented in Python, to streamline the creation and evaluation, via DL, of satellite imagery datasets. The methodology covers the entire workflow: from data acquisition, labelling, and preprocessing to model adaptation, training, and evaluation. Specifically, this approach is applied to gather and validate a high-resolution dataset for forest wildfire detection, the California Wildfire GeoImaging Dataset (CWGID). Additionally, this methodology can be adapted for various environmental monitoring tasks, showing its versatility in studying and responding to different environmental changes.

## 2. Building a Sentinel-2 Satellite Imagery Dataset

To construct the CWGID, a multi-step process is needed.

### 2.1. *Gathering and Refining Historic Wildfire Polygon Data from California*

The initial step is to gather georeferenced forest wildfire polygon data from California, sourced from the Fire and Resource Assessment Program (FRAP) maintained by the California Department of Forestry and Fire Protection [12]. This FRAP data includes perimeters of past wildfires and serves as the geographic reference needed to select satellite imagery with GEE. Figure 1 illustrates the polygons from the FRAP. The polygons delineated in purple represent areas affected by wildfires in forested regions. These delineated polygons are used to create the CWGID.

Then, in Python, the Pandas library [13] is used to organize the forest wildfire attribute data into a Pandas DataFrame, which is then filtered to align with the launch date and operational phase of the Sentinel-2 satellites, selected for their open-source, high-resolution imagery capabilities [14]. Additionally, the dates are adjusted to fall within the green-up period, avoiding the winter and fall seasons where snow cover could interfere with identifying burnt areas.

Next, the data is formatted to meet GEE's querying specifications:

- A 15-day range for pre- and post-wildfire dates is generated and added to the DataFrame.
- Using Pandas, the date ranges are formatted to meet GEE's requirements.

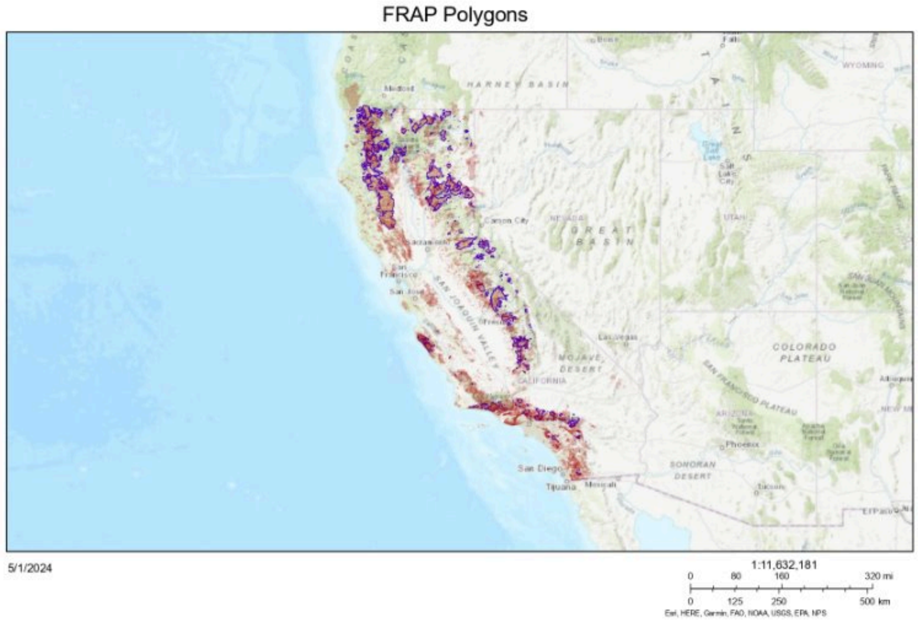


**Figure 1**. *Representation of the Polygon Data from the FRAP. Polygons delineated in purple represent wildfires that occurred in forested areas, used for the California Wildfire GeoImaging Dataset (CWGID).*

- Using the pyproj library [15], the recorded point coordinates are converted from NAD83 to WGS84 to facilitate the querying process.
- With the geopy [16] library, the coordinates of the squared region of interest are calculated, featuring a side length of 15 miles.

### *2.2. Downloading the Imagery Data Using GEE's Python API*

GEE is a cloud-based platform for global environmental data analysis. It combines an extensive archive of satellite imagery and geospatial datasets with powerful computational resources to enable researchers to detect and quantify changes on the Earth's surface. GEE's Python API offers an accessible interface for automating the process of satellite imagery downloads, making it a popular tool for environmental monitoring and research projects.

Multiple steps are needed to set up the GEE's Python API. First, a project is created in Google Cloud Console and the Earth Engine API is enabled. Authentication and Google Drive editing rights are configured to effectively manage and store the downloaded imagery. Following the setup, the Earth Engine Python API is installed on a local machine, and the necessary authentications are performed to initialize the API.

Then, a Python script is developed to automate the download of images depicting the pre- or post-wildfire data using GEE's Python API (see Program 1). To download three-channel RGB GeoTIFF imagery, the bands B4 (red), B3(green), and B2(blue) need to be specified (different band compositions can be selected in this step). In satellite imagery, bands refer to specific wavelength ranges captured by the satellite sensors, and they are used to create composite images that highlight different features of the Earth's surface. These bands correspond to the visible spectrum, which is useful for visual interpretation and analysis. The script to download the satellite imagery needed using GEE (see Program 1) is configured with a for loop to iterate through each entry in the DataFrame, extracting necessary parameters such as date ranges, region of interest (ROI) coordinates, and center coordinates of each wildfire polygon. Also, the script is designed to specify parameters such as the desired image collection and a threshold for cloud coverage. Tiles exhibiting more than 10% cloud coverage are automatically excluded to maintain data quality. Finally, the images are downloaded and exported to Google Drive in a GeoTIFF format.

Figure 2 presents an example of a pre-and post-wildfire imagery pair downloaded from GEE to Google Drive using Program 1 .

### *2.3. Creating the Ground Truth Wildfire Labels*

Ground truth masks are essential in forest wildfire detection and general land cover classifications [17]. In this project, these type of masks are generated to label the data.

First, Python is used to rasterize the combined geometry of the forest wildfire polygon data and the downloaded post-wildfire RGB satellite imagery. Specifically, the forest wildfire polygons are accessed in Python using GeoPandas [18] and reprojected to match the coordinate system of the satellite imagery (EPSG:4326). Then, each post-wildfire RGB image is locally and temporarily downloaded from Google Drive, with essential properties such as width, height, transform, and bounds extracted using the rasterio library [19]. Next, the geometry column from the forest wildfire polygon data is extracted and intersected with each image bound using Python's shapely [20] library. Finally, binary masks are created by rasterizing the combined geometries. These masks match the dimensions of the satellite images, ensuring that each pixel labeled as wildfire damage corresponds directly to the polygon data (see Program 2). The binary masks are saved temporarily in GeoTIFF format and are uploaded to a dedicated Google Drive folder. All the temporary local files were deleted to clear space and maintain system efficiency.

```
# Authenticate into EE
ee.Authenticate()

# Initialize EE
ee.Initialize()

# Initialize a list for iteration
indices = []
for i in range(0, x):  # modify depending on your number of data points
    indices.append(i)

# Define the bands you want to download
bands_rgb = ['B4', 'B3', 'B2']  # Red, Green, Blue

event_type = 'before'  # Toggle this to 'after' as needed

for i in indices:
    # Set the center point of the wildfire polygon
    center_point = ee.Geometry.Point(round(data[i][1], 2), round(data[i][2], 2))

    # Select an image from the specified satellite collection for the center point
    tile = ee.ImageCollection('COPERNICUS/S2') \  # Define the satellite imagery you will be working
with, in this case Sentinel 2
        .filterBounds(center_point) \
        .filterDate(data[i][3], data[i][4]) \  # Define a date range for the satellite image
        .sort('CLOUDY_PIXEL_PERCENTAGE') \  # Organize the images based on the % cloud cover
        .first()  # Get the image with the lowest cloud cover

    # Check the properties of the image
    image_properties = tile.getInfo()
    cloudy_percentage = image_properties.get(
         'properties', {}).get('CLOUDY_PIXEL_PERCENTAGE', 0)  # Obtain the exact percentage of cloud
cover for each image

    # Proceed if the image has less than 10% cloud coverage
    if cloudy_percentage <= 10:
        # Define the Region of Interest (ROI) using the provided coordinates
        roi = ee.Geometry.Polygon([[
            [data[i][9], data[i][10]],
            [data[i][11], data[i][12]],
            [data[i][13], data[i][14]],
            [data[i][15], data[i][16]],
            [data[i][17], data[i][18]]]])

        # Select the RGB bands from the image
        rgb_tile = tile.select(bands_rgb)

        # Export RGB image to Google Drive
        rgb_task = ee.batch.Export.image.toDrive(**{
            'image': rgb_tile,
            'description': f'RGB_{event_type}Fire'
            + str(data[i][0]),  # Descriptive name for the task
            'folder': f'GEE_FireImagesRGB_{event_type}',  # Folder in Google Drive to save the image
            'scale': 10,  # Adjust the scale as needed
            'region': roi.getInfo()['coordinates'],  # Region of Interest for the export
            'crs': 'EPSG:4326',  # Coordinate reference system
            'fileFormat': 'GeoTIFF',  # File format for the export
        })
        rgb_task.start()

        # Monitor the task status until it completes or fails
        while not rgb_task.status()['state'] in ['COMPLETED', 'FAILED', 'CANCELLED']:
            print(f'RGB Task for {event_type} fire is', rgb_task.status()['state'])
            # Add a short wait before checking again

        # Check the final task status
        rgb_task_status = rgb_task.status()
        print(f"RGB Task Status for {event_type} fire:", rgb_task_status)
        print(f"RGB Task Error Message for {event_type} fire:", rgb_task_status.get("error_message",
"No error message"))
    else:
        # Skip images with more than 10% cloud coverage
        print(f"Skipping image {i} due to cloudy percentage ({cloudy_percentage} %) > 10 %")
```

**Program 1**. *Script to automate the download of pre- or post-wildfire images using GEE's Python API. It iterates through a DataFrame*

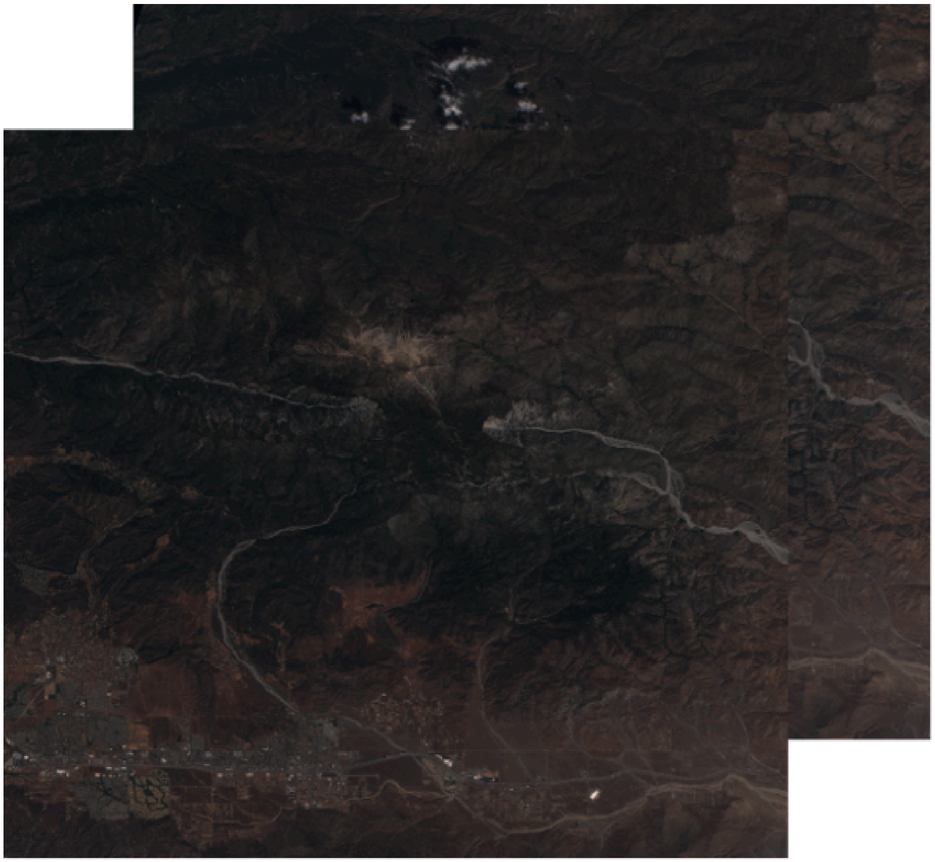


**Figure 2.** *Example of a pre and post-wildfire RGB image pair of a forested area downloaded using GEE's Python API.*

In the resulting ground truth masks, the pixel values are set to zero if they are outside of a wildfire polygon, indicating unaffected areas, and set to one if they are within the polygon boundaries, indicating areas affected by a forest wildfire. Figure 3 displays an example of a resulting ground truth mask.

### 2.4. Image Segmentation and Data Preparation for Deep Learning Architectures

Next, the satellite images and their corresponding ground-truth masks are cropped into smaller tiles that maintain the imagery's spatial resolution (10m). Often, satellite imagery needs to be resized and downscaled to accommodate deep learning (DL) architectures, which can result in the loss of essential details such as subtle indicators of early-stage wildfires. Moreover, using smaller images enhances the efficiency of DL models by lowering computational demands and speeding up training times [21], [22].

To do this, a tile size of 256x256 pixels is specified and each RGB image is downloaded individually from Google Drive to a temporary local folder. Using Python's rasterio library, the original RGB images are opened to obtain their dimensions. Then, the number of rows and columns for the tiles is calculated based on the chosen tile size. Next, a rasterio Window object is used to extract the corresponding portion from the original image and read the RGB data, ensuring the order of the bands (B4, B3, and B2).

The segmented RGB tiles are then saved as GeoTIFF files using the tifffile Python library [23]. This is a critical step to maintain the integrity of the three-channel RGB data, as the rasterio library alone can alter the color of the images during saving. Additionally, the metadata of

```python
import geopandas as gpd
import numpy as np
import rasterio
from shapely.geometry import box, shape
import os

# Define the path to your Shapefile - replace with your specific path
shapefile_path = "YourShapefileDirectory/FirePolygons.shp"

# Read the Shapefile using geopandas
gdf = gpd.read_file(shapefile_path)

# Reproject the shapefile to EPSG:4326 to match the satellite imagery coordinate system
gdf = gdf.to_crs(epsg=4326)

# Create a directory to store the output raster masks if it doesn't already exist
output_dir = "raster_masks"
if not os.path.exists(output_dir):
    os.makedirs(output_dir)

# Iterate through the rows in the attribute table
for index, row in gdf.iterrows():
    object_id = row["OBJECTID"]
    image_path = f"YourImageDirectory/RGB_AfterFire{object_id}.tif"
    # Check if the image exists
    if os.path.exists(image_path):
        # Open the image using rasterio
        with rasterio.open(image_path) as src:
            image_width = src.width
            image_height = src.height
            image_transform = src.transform
            image_bounds = box(
                src.bounds[0], src.bounds[1],
                src.bounds[2], src.bounds[3]
            )

        # Extract the geometry column
        geom = shape(row["geometry"])
        clipped_geom = geom.intersection(image_bounds.envelope)

        if not clipped_geom.is_empty:
            # Create a two-dimensional label by rasterizing the
            # clipped geometry
            clipped_mask = rasterio.features.geometry_mask(
                [clipped_geom],
                out_shape=(image_height, image_width),
                transform=image_transform,
                invert=True,
            )

            # Save the image with the two-dimensional label overlay
            output_image_path
            = f"{output_dir}/Masked_{object_id}.tif"
            with rasterio.open(
                output_image_path,
                "w",
                driver="GTiff",
                width=image_width,
                height=image_height,
                count=1,
                dtype=np.uint8,
                crs=src.crs,
                transform=image_transform,
            ) as dst:
                dst.write(clipped_mask.astype(np.uint8), 1)
```

**Program 2.** *Building Ground Truth Masks*

the saved tiles is updated to include georeferencing information and to modify parameters such as width, height, and transform (see Program 3).

A similar approach is used to segment the binary masks, specifying that the images contain only one band.

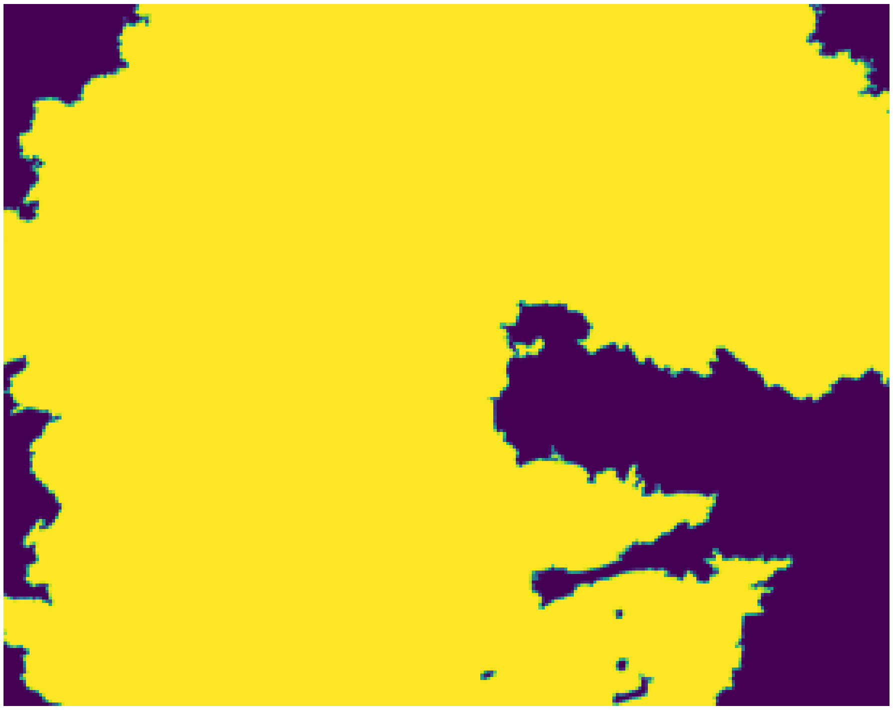


**Figure 3**. *Example of a resulting ground truth mask in a forested area affected by wildfires. The mask highlights wildfire-affected areas in yellow and unaffected areas in purple. This binary mask is used to train and validate deep learning models for accurate wildfire detection.*

By combining the capabilities of rasterio for efficient geospatial data handling and the tifffile library for preserving the RGB data during saving, the original images are cropped into smaller RGB tiles. This approach preserves the resolution and the georeferencing information of the images, preparing them to train DL applications.

### *2.5. Data Augmentation for Wildfire Damage Detection*

It is essential to have a balanced dataset that includes both forest wildfire-affected and unaffected areas. Initially, the dataset comprised 82,082 tiles, but only 4,847 of these showed signs of wildfire damage, indicating a significant class imbalance. This could lead to overfitting, with the model biased towards undamaged landscapes.

To address this imbalance and enhance the model's accuracy in detecting wildfire-affected areas, data augmentation techniques are implemented. Specifically, functions using the rasterio and NumPy [24] libraries are developed to perform image transformations, which include rotating the GeoTIFF tiles by 90°, 180°, and 270°, and flipping them horizontally and vertically. These transformations are applied to the imagery and the mask tiles that contain fire polygons and to their corresponding pre-wildfire RGB GeoTIFF tiles.

Finally, the augmentation process increased the diversity of the training data and resulted in a total of 106,317 pairs of labeled RGB GeoTIFF image tiles, with 29,082 positive instances (wildfire damage), improving the class balance.

Figure 5 illustrates the data augmentation process applied to the satellite imagery. The original image tile is shown alongside its augmented versions, which include rotations and flips. This augmentation increases the diversity of the dataset, helping to balance the classes and improve the model's ability to detect wildfire-affected areas accurately.

```
from rasterio import Window
from tifffile import imwrite

# Function to save image tiles without changing the data type
def save_rgb_tiles(image_path, output_folder, tile_size, parent_name):
    # Open the source image file
    with rasterio.open(image_path) as src:
        height = src.height # Get the height of the source image
        width = src.width # Get the width of the source image

        # Calculate the number of tiles in both dimensions
        num_rows = height // tile_size
        num_cols = width // tile_size

        tile_counter = 1  # Initialize the tile counter

        # Iterate over the number of rows of tiles
        for i in range(num_rows):
            # Iterate over the number of columns of tiles
            for j in range(num_cols):
                # Define the window for the current tile
                window = Window(j * tile_size, i * tile_size, tile_size, tile_size)

                # Read the original data without modifications
                # Assuming band order B4, B3, B2
                tile = src.read((1, 2, 3), window=window)
                # Create a unique name for the tile
                tile_name = f"{parent_name}_tile_{tile_counter}.tif"
                # Define the path to save the tile
                tile_path = os.path.join(output_folder, tile_name)

                # Save the tile using tifffile without changing
                # data type
                imwrite(tile_path, tile)

                # Copy the metadata from the source image
                meta = src.meta.copy()
                # Get the transformation matrix for the current window
                transform = src.window_transform(window)
            # Update the metadata with the new dimensions and transformation
                meta.update({
                    'width': tile_size,
                    'height': tile_size,
                    'transform': transform
                })
              # Save the tile with updated metadata using rasterio
                with rasterio.open(tile_path, 'w', **meta) as dst:
                    dst.write(tile)
                tile_counter += 1  # Increment the tile counter
```

**Program 3.** *Function to Crop RGB Image Tiles*

## 3. Evaluating the CWGID Using Convolutional Neural Networks

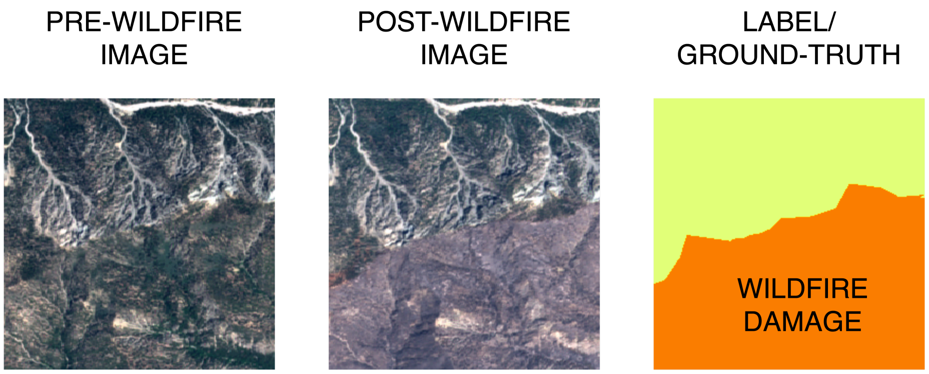


**Figure 4.** *Example of cropped pre- and post-wildfire images and their corresponding label.*

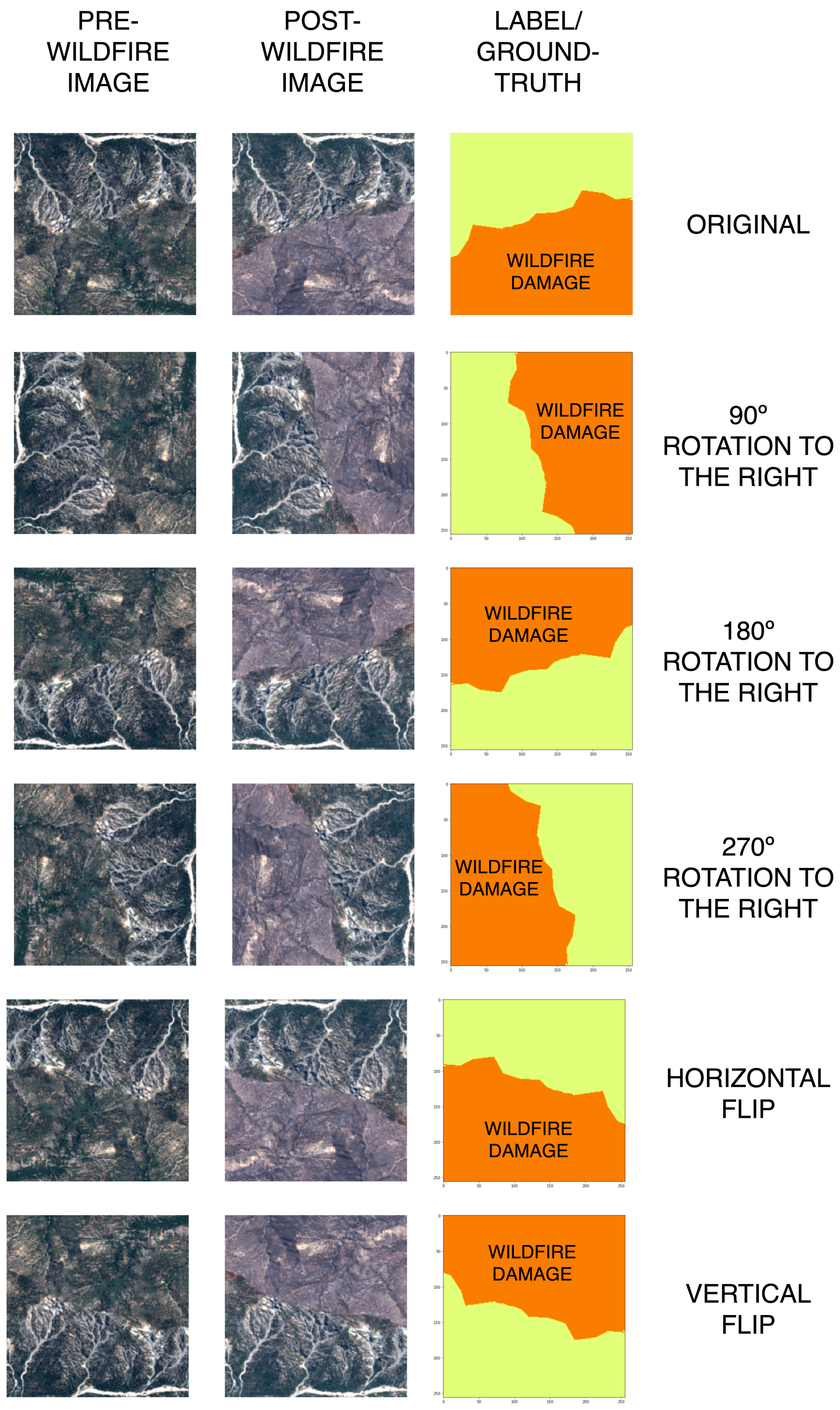


**Figure 5.** *Representation of the data augmentation performed to the 256px*256px tiles to increase positive examples of wildfire damage.*

The CWGID is tested for demonstration purposes using the well-known CNN architectures VGG16 and EfficientNet, both implemented using Python's TensorFlow's Keras API. It is important to mention that both models were executed locally on a MacBook Pro equipped with a 2.4 GHz Intel Core i9 processor and 16 GB of 2400 MHz DDR4 memory.

Satellite image data usually requires significant storage and processing power because each image contains a large amount of data across the chosen spectral bands and stores important geographic information. Hence, the scale and speed of model training observed in the results below is constrained.

For users with access to high-performance computing clusters or cloud-based GPU instances, processing times can significantly reduce.

### *3.1. VGG16 Implementation*

VGG16 [25] is a deep CNN designed for image processing and classification tasks. It consists of 13 convolutional layers, 5 pooling layers, and 3 fully connected layers. This model is adapted to train on the dataset and detect positive and negative instances of forest wildfires. The VGG16 architecture only trains on the post-wildfire 3-channel RGB imagery (i.e., images collected prior to the wildfire were not used with VGG16), which contains both.

To use this CNN architecture with GeoTIFF satellite images, which typically store geographic data not supported by most deep-learning libraries, a custom function is implemented to process and feed the data to the model using the rasterio library and the shuffle function from the Scikit-learn library [26], [27]. This function (see Program 4) is crucial for handling the unique structure of GeoTIFF files and converting them into a format suitable for deep learning. It is designed to read batches of GeoTIFF files, shuffle them to ensure randomness, and process them into a three-dimensional array compatible with VGG16. Here is how it works:

- File Paths Initialization: The function starts by taking the file paths of the GeoTIFF images and shuffling them using the shuffle function from Scikit-learn to ensure the batches are randomly ordered.
- Batch Processing: It processes the images in batches of 32 files. For each batch, the function initializes empty lists for images and labels.
- Reading and Transforming Images: It reads each GeoTIFF file using rasterio. The images are then transformed to have the channels last (i.e., changing the format from **channels, height, width** to **height, width, channels**) to use with Keras.
- Labels are assigned based on the presence of “Damaged” in the file paths. Specifically, if the file path contains the string “/Damaged/”, the image is labeled as 1 (indicating it shows a damaged area). If the file path does not contain this string, the image is labeled as 0 (indicating it shows an undamaged area). In this method, the ground-truth masks are not directly used but are leveraged to define the imagery file paths.
- Yielding Batches: Finally, the function yields batches of images and labels as Numpy arrays, which are then fed into the VGG16 model for training.

To detect wildfire-affected areas with VGG16, the model is initiated using the pre-trained weights from the ImageNet dataset. The convolutional base is frozen to preserve the integrity of the learned features and to focus on the training of the added layers. A Flatten operation is applied to the output to transform the two-dimensional feature maps into a one-dimensional vector. Then, a fully connected dense layer with 512 neurons, L2 regularization, and ReLU activation is added. A Dropout layer with a rate of 0.5 is included to further prevent overfitting. Finally, the network becomes a single-neuron dense layer with a sigmoid activation function that produces a probability score of wildfire damage. Before training, class weights are calculated using the Scikit-learn library. Furthermore, Keras callbacks are set up for early stopping and model checkpointing to avoid overfitting.

```python
from sklearn.utils import shuffle

# Define the base paths for training and testing
base_training_path = "Insert your training file path"
base_testing_path = "Insert your testing file path"

def custom_image_generator(file_paths, batch_size):
    while True:
        file_paths = shuffle(file_paths)

        for i in range(0, len(file_paths), batch_size):
            batch_files = file_paths[i : i + batch_size]
            images, labels = [], []

            for file in batch_files:
                with rasterio.open(file) as src:
                    image = src.read()
                    # Channels last
                    image = np.moveaxis(image, 0, -1)

                label = 1 if "/Damaged/" in file else 0
                images.append(image)
                labels.append(label)

            yield np.array(images), np.array(labels)
```

**Program 4**. *Custom Function to Feed GeoTIFF Files to the VGG16 Model: The function reads batches of 32 GeoTIFF files - shuffles them - and processes them into a three-dimensional array compatible with VGG16.*

The complete model is compiled with the Adam optimizer [28], binary cross-entropy loss [29], and accuracy, precision, and recall as the performance metrics (see Program 5). The Adam optimizer is chosen because it adapts the learning rate during training. In the context of satellite image classification, where the landscape can vary significantly across images, Adam's ability to adjust the learning rate is important. Also, the binary cross-entropy loss is chosen because it is tailored for binary classification tasks, such as classifying between damaged and undamaged forested areas in satellite images.

The VGG16 model is trained and validated on 49313 three-channel RGB images (divided as follows: 90% for training and 10% for validation) and is tested on 5479 images. Training took around 1010 minutes for 10 epochs. The detailed training and validation performance metrics for each epoch are provided in Table 1. The graphs showing the changes of the training and validation metrics over the 10 epochs can be seen in Figure 6.

During the training process, the model's performance on the training set showed a consistent increase in accuracy, precision, and recall. Specifically, the training accuracy improved from 0.6729 in the first epoch to 0.7692 in the tenth epoch. The precision increased from 0.7241 to 0.8622, and the recall fluctuated but overall showed an upward trend, starting at 0.6229 and ending at 0.6742.

On the validation set, the performance metrics also exhibited positive trends. The validation accuracy began at 0.7291 and varied across the epochs, reaching 0.7527 by the final epoch. Validation precision improved from 0.7689 in the first epoch to 0.8164 in the tenth epoch, while validation recall started at 0.6812 and ended at 0.6760.

### 3.2. *EfficientNet Implementation*

EfficientNet [30] is a CNN architecture that uniformly scales network width, depth, and resolution with a fixed set of scaling coefficients. EfficientNet's architecture begins with a base model, EfficientNet-B0, designed to find the optimal baseline network configuration. The following versions of the network are further scaled versions of B0, offering multiple models for different computational budgets.

```
from keras.models import Model
from keras.layers import Flatten, Dense
from keras.optimizers import Adam
from keras.applications import VGG16
from keras.metrics import Precision, Recall

# Define the base model using pre-trained ImageNet weights
base_model = VGG16(weights='imagenet', include_top=False, input_shape=(256, 256, 3))

# Freeze the convolutional base to prevent weights from being updated
for layer in base_model.layers:
    layer.trainable = False

# Add custom layers on top of the base model
x = Flatten()(base_model.output) # Flatten the output to make it suitable for dense layers
x = Dense(512, activation='relu', kernel_regularizer='l2')(x) # Add a dense layer with 512 units, L2
regularization and ReLU activation
output  =  Dense(1,  activation='sigmoid')(x)  #  Output  layer  with  a  sigmoid  activation  for  binary
classification

# Construct the complete model
model = Model(inputs=base_model.input, outputs=output)

#Callbacks to prevent overfitting
early_stopping = EarlyStopping(monitor='val_loss', patience=3, restore_best_weights=True)
model_checkpoint = ModelCheckpoint('best_modelVGG16.keras', monitor='val_loss', save_best_only=True)

# Compile the model with Adam optimizer and binary cross-entropy loss
model.compile(optimizer=Adam(learning_rate=0.0001),
              loss='binary_crossentropy',
              metrics=['accuracy', Precision(), Recall()])
```

**Program 5**. *Adaptation of VGG16 for Wildfire Damage Detection*

For this project, Efficient-B0 is adapted, trained, and tested using pre- and post-wildfire RGB GeoTIFF imagery pairs from the CWGID. To do this, an approach commonly known as Early Fusion (EF) is employed, where six-channel GeoTIFF files that combine the image pairs into a single input are built (see Figure 7). To use this, a custom function that allows the use of 6-channel GeoTIFF data with Efficient-B0 is built with the rasterio library and Keras' Sequence function (see Program 6).

Furthermore, the data labeling and its use are formulated using the same pipeline as VGG16, by specifying the base paths to the training and testing directories for both damaged and undamaged classes. As above, the labeling of this model is based on the presence or the absence of 'Damaged' in the imagery file paths. The base EfficientNet-B0 model is then

**Table 1**. *Performance Metrics for Training and Validation Sets Using VGG16 Across Epochs.*

| Epoch | Training Accuracy | Training Precision | Training Recall | Validation Accuracy | Validation Precision | Validation Recall |
|---|---|---|---|---|---|---|
| 1/10 | 0.6729 | 0.7241 | 0.6229 | 0.7291 | 0.7689 | 0.6812 |
| 2/10 | 0.7406 | 0.7892 | 0.6992 | 0.7471 | 0.7716 | 0.7256 |
| 3/10 | 0.7537 | 0.8079 | 0.7052 | 0.7473 | 0.7729 | 0.7261 |
| 4/10 | 0.7555 | 0.8306 | 0.6788 | 0.7541 | 0.7899 | 0.7155 |
| 5/10 | 0.7568 | 0.8354 | 0.6765 | 0.7543 | 0.8070 | 0.6906 |
| 6/10 | 0.7632 | 0.8439 | 0.6813 | 0.7539 | 0.8226 | 0.6655 |
| 7/10 | 0.7653 | 0.8514 | 0.6768 | 0.7569 | 0.8049 | 0.6998 |
| 8/10 | 0.7661 | 0.8504 | 0.6803 | 0.7635 | 0.8228 | 0.6887 |
| 9/10 | 0.7680 | 0.8536 | 0.6808 | 0.7504 | 0.8771 | 0.6008 |
| 10/10 | 0.7692 | 0.8622 | 0.6742 | 0.7527 | 0.8164 | 0.6760 |

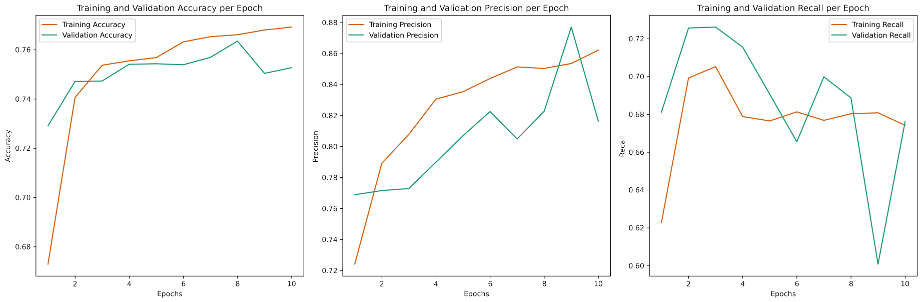


**Figure 6**. *Graphic Representation of the Performance Metrics for the Training and Validation Sets Using VGG16 Across 10 Epochs.*

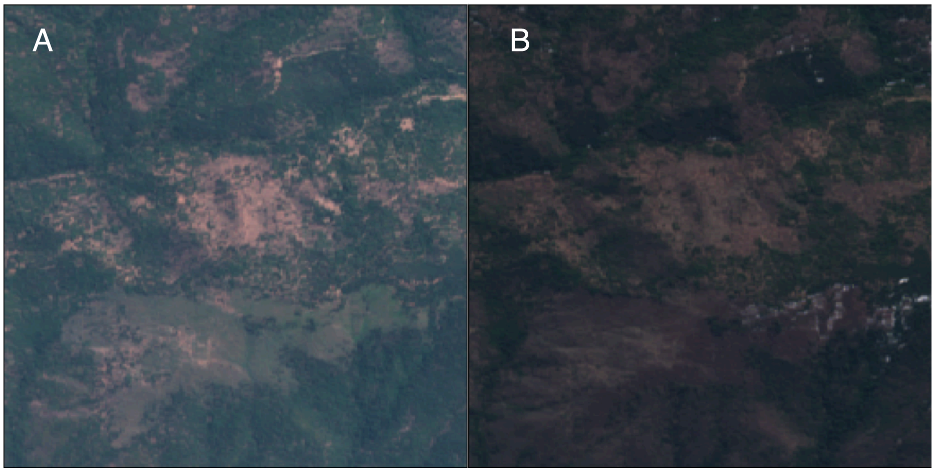


**Figure 7**. *Representation of a 6 Channel RGB GeoTIFF Input. A: Representation of a 3-channel RGB GeoTIFF forested area* before *a wildfire B: Visual example of a 3-channel RGB GeoTIFF forested area* after *a wildfire.*

loaded without its top layers (classification layers) and without pre-trained weights, as the pre-trained weights are specific to 3-channel images. A Global Average Pooling layer is used to reduce the spatial dimensions of the feature maps. To reduce overfitting, a Dropout layer is added with a specified rate of 30

The model is built with an input tensor defined to accept 256px by 256px images with six channels. Initially, a standard convolutional layer is applied to this input tensor to perform an initial convolution operation, reducing it to a three-channel format. The base EfficientNet-B0 model is loaded both without the top layers (classification layers) and the pre-trained weights, as these were trained using 3-channel images. A Global Average Pooling layer is used to reduce the spatial dimensions of the feature maps. To reduce overfitting, a Dropout layer is added with a specified rate of 30%. Following this, a dense layer is also added with 1024 neurons and ReLU activation (see Program 7). In the end, the network outputs a probability indicating the likelihood of the image showing ‘damaged’ versus ‘undamaged’ areas.

Before training, class weights are calculated using the Scikit-learn library. Furthermore, Keras callbacks are set up for early stopping, model checkpointing, and reducing the learning rate when the validation loss plateaus, which helps optimize the training process. As VGG16, the model is compiled with the Adam optimizer and binary cross-entropy loss, with accuracy, precision, and recall as performance metrics.

For testing purposes, this EF architecture is trained on 23833 pre- and post-wildfire image pairs (divided as follows: 90% for training and 10% for validation) and is tested on 2716 pre-

```
from keras.utils import Sequence
# Custom Data Generator for Six-Channel Images


class SixChannelGenerator(Sequence):
    # Change this depending on your own data
    # Initialize the generator
    def __init__(
        self,
        file_paths, # List of file paths to the six-channel GeoTIFF images
        labels, # Corresponding labels for the images
        batch_size=32, # Number of images to be returned in each batch
        dim=(256, 256), # Dimension of the images
        n_channels=6, # Number of channels in the images (6 in this case)
        shuffle=True, # Shuffle the data at the end of each epoch
    ):
        self.file_paths = file_paths
        self.labels = labels
        self.batch_size = batch_size
        self.dim = dim
        self.n_channels = n_channels
        self.shuffle = shuffle
        self.on_epoch_end()
    # Define the number of batches
    def __len__(self):
        return int(np.ceil(len(self.file_paths) / self.batch_size))
    # Generate batch of data
    def __getitem__(self, index):
        # Generate indices of the batch
        batch_paths = self.file_paths[
            index * self.batch_size: (index + 1) * self.batch_size
        ]
        batch_labels = self.labels[
            index * self.batch_size: (index + 1) * self.batch_size
        ]
        # Initialize the arrays for the batch data and labels
        batch_x = np.empty(
            (len(batch_paths), *self.dim,
             self.n_channels), dtype=np.float32
        )
        batch_y = np.array(batch_labels, dtype=np.float32)
        # Load and read the data for each file in the batch
        for i, path in enumerate(batch_paths):
            with rasterio.open(path) as src:
    # Read the image and select the first 'n_channels' channels
                img = src.read()[: self.n_channels,
                                 : self.dim[0],
                                 : self.dim[1]]
    # Convert from channels_first to channels_last format
                img = np.moveaxis(img, 0, -1)
                batch_x[i,] = img / 255.0  # Normalize images to [0, 1] range

        return batch_x, batch_y
    # Shuffle the data at the end of each epoch
    def on_epoch_end(self):
        if self.shuffle:
            temp = list(zip(self.file_paths, self.labels))
            np.random.shuffle(temp)
            self.file_paths, self.labels = zip(*temp)
```

**Program 6**. *Custom Function to Feed Multi-Channel GeoTIFF Files to the EfficientB0 Model*

and post-wildfire image pairs. Training took around 930 minutes for 13 epochs. The detailed training and validation performance metrics for each epoch are provided in Table 2. Furthermore, the graphs depicting the changes of the training and validation metrics over the 13 epochs can be seen in Figure 8.

During the training process, the model's performance on the training set showed a consistent increase in accuracy, precision, and recall. Specifically, the training accuracy improved from 0.7971 to 0.9588, precision increased from 0.7204 to 0.9665, and recall increased from 0.4053 to 0.8776 over the same period.

```
from keras.layers import Input, Conv2D, Dense, GlobalAveragePooling2D, Dropout
from keras.models import Model
from keras.applications import EfficientNetB0
from keras.callbacks import EarlyStopping, ModelCheckpoint, ReduceLROnPlateau

# Model Adaptation for Six-Channel Input


# Change this depending on your own data
def create_efficientnet_six_channel(input_shape=(256, 256, 6), dropout_rate=0.3):
    input_tensor = Input(shape=input_shape)
    x = Conv2D(3, (3, 3), padding='same', activation='relu')(input_tensor)
    base_model = EfficientNetB0(
        include_top=False, input_tensor=x, weights=None)
    x = GlobalAveragePooling2D()(base_model.output)
    x = Dropout(dropout_rate)(x)
    x = Dense(1024, activation='relu')(x)
    output = Dense(1, activation='sigmoid')(x)
    model = Model(inputs=input_tensor, outputs=output)
    return model


# Initialize the model
model = create_efficientnet_six_channel()
# Model Training with Callbacks for Optimal Training Control
early_stopping = EarlyStopping(
    monitor='val_loss', patience=5, restore_best_weights=True)
model_checkpoint = ModelCheckpoint('yourmodel.keras', save_best_only=True)
reduce_lr = ReduceLROnPlateau(
    monitor='val_loss', factor=0.2, patience=2, min_lr=1e-6, verbose=1)

# Compile the model
model.compile(optimizer='adam',
              loss='binary_crossentropy',
              metrics=['accuracy', Precision(), Recall()])
```

**Program 7.** *Initializing and Training EfficientNetB0 for Six-Channel Image Input*

On the validation set, the performance metrics exhibited positive trends as well. The validation accuracy began at 0.7843, peaked at 0.9359 around the eighth epoch, and remained high at 0.9333 by the final epoch. Validation, precision, and recall also showed improvements,

**Table 2.** *Performance Metrics for Training and Validation Sets Using EfficientNetB0 Across Epochs.*

| Epoch | Training Accuracy | Training Precision | Training Recall | Validation Accuracy | Validation Precision | Validation Recall |
|---|---|---|---|---|---|---|
| 1/13 | 0.7971 | 0.7204 | 0.4053 | 0.7843 | 0.7688 | 0.4973 |
| 2/13 | 0.8803 | 0.8299 | 0.6995 | 0.8703 | 0.7499 | 0.5754 |
| 3/13 | 0.8979 | 0.8599 | 0.7424 | 0.8152 | 0.8034 | 0.5404 |
| 4/13 | 0.9022 | 0.8668 | 0.7531 | 0.8179 | 0.7222 | 0.6024 |
| 5/13 | 0.9187 | 0.9016 | 0.7842 | 0.9125 | 0.7597 | 0.5647 |
| 6/13 | 0.9265 | 0.9156 | 0.8015 | 0.9310 | 0.7014 | 0.5948 |
| 7/13 | 0.9298 | 0.9181 | 0.8121 | 0.9363 | 0.7289 | 0.5726 |
| 8/13 | 0.9346 | 0.9260 | 0.8235 | 0.9359 | 0.7104 | 0.5946 |
| 9/13 | 0.9370 | 0.9274 | 0.8315 | 0.9057 | 0.7580 | 0.5464 |
| 10/13 | 0.9424 | 0.9387 | 0.8416 | 0.9291 | 0.7522 | 0.5835 |
| 11/13 | 0.9537 | 0.9590 | 0.8655 | 0.9333 | 0.9386 | 0.7959 |
| 12/13 | 0.9573 | 0.9644 | 0.8739 | 0.9363 | 0.9516 | 0.7959 |
| 13/13 | 0.9588 | 0.9665 | 0.8776 | 0.9333 | 0.9606 | 0.7757 |

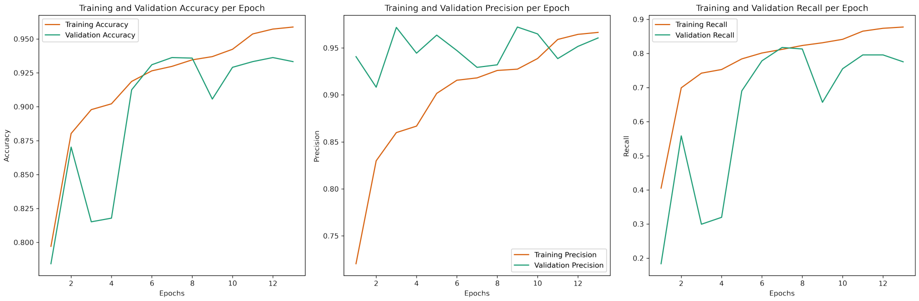


**Figure 8**. *Graphic Representation of the Fluctuation of the Performance Metrics for the Training and Validation Sets Using EfficientNetb0 Across 13 Epochs.*

with precision starting at 0.7688 and reaching up to 0.9606, while recall improved from 0.497 to 0.7757.

### *3.3. DL Results*

The results in Table 3 compare the performance of the two neural network architectures used on the CWGID. Each architecture's performance metrics are assessed, including loss, accuracy, precision, recall, and training time. Also, the percentage of the CWGID used to train each architecture is specified.

Both models are trained with subsets of the CWGID dataset, with the 6-channel input EF EfficientNet model and the VGG16 model using around 25% of the data.

The 6-channel input EfficientNet achieved the lowest loss (0.178) and highest accuracy (92.6%), making it the most effective model in terms of overall performance. Its precision was at 92.5% and its recall was 81.9%, indicating a strong ability to identify damaged areas correctly with some missed detections.

VGG16, had a higher loss (0.718) with an accuracy of 76.4%. Its precision was 83.7% and its recall was 68.9%. This architecture took a longer training time of 1010 minutes compared to 930 minutes for the EfficientNet model. The EfficientNet model shows a slight faster training time when considering the amount of training epochs.

The 6-channel input EfficientNet demonstrated superior performance in accurately classifying areas affected by wildfires with high precision and accuracy, making it an efficient choice for applications where accurate assessments are critical. Moreover, it is important to mention that VGG16 was tested with 6-channel inputs but proved computationally intensive.

**Table 3**. *Performance Metrics of the Models.*

| Model | 6-channel input EfficientNet | VGG16 |
|---|---|---|
| Data from the CWGID (%) | 25 | 25 |
| Loss | 0.178 | 0.718 |
| Accuracy | 0.926 | 0.764 |
| Precision | 0.925 | 0.837 |
| Recall | 0.819 | .689 |
| Time (minutes) | 933 | 1010 |

## 4. Discussion

The methodology developed in this study facilitates the creation of large, reliable, and cloud-free bi-temporal satellite imagery datasets, which are essential for accurate environmental monitoring. Also, by leveraging historical and curated datasets for the labeling, our approach maximizes the value of existing data, accurately labels the imagery, and minimizes the need for manual labeling.

Furthermore, the methodology was successfully evaluated, as the CWGID enabled the adapted CNN models to learn and identify the signs of wildfire damage with high accuracy, specifically with the EF EfficientNet-B0 model. Additionally, the proposed methodology produced a greater number of examples compared to other similar datasets in the field [7], [31], [32], which can boost the performance of deep learning models, making them more effective at detecting and analyzing environmental changes. Moreover, the high precision and recall metrics for EfficientNet, particularly the recall of 81.9%, display the model's capacity to correctly identify wildfire-affected areas with fewer false negatives, which remains critical for rapid and effective wildfire monitoring.

On top of that, the dataset's bi-temporal collection of pre- and post-wildfire satellite imagery also offers the possibility to use different and accurate approaches for detecting forest wildfires with satellite imagery. The comparative analysis of VGG16 and EfficientNet shows the advantage of employing a bitemporal approach to satellite imagery analysis under a single input. By evaluating *before* and *after* images as coupled inputs, the model identifies changes that a single post-event imagery might not reveal, as shown by VGG16.

## 5. Conclusion

This paper illustrates a Python methodology that integrates DL with satellite imagery to enhance environmental monitoring. Specifically, the development of the California Wildfire GeoImaging Dataset (CWGID) and its evaluation using DL architectures are outlined. The approach explained throughout this work allows for accurate detection of forest wildfires, as demonstrated using DL architectures, and can be applied to a broad range of environmental issues beyond just wildfire detection. Firstly, historical data, Python-based tools, and Google Earth Engine are leveraged to build and label a large, cloud-free Sentinel-2 satellite imagery dataset depicting forest wildfire examples. Then, this dataset is used to train well-known CNN architectures, enabling the effective detection of forest wildfires within the satellite imagery. An accuracy of 92.6 % with the EfficientNetB0 model was obtained, indicating the potential effectiveness of the methodology. Future studies will include using Fully Convolutional Networks (FCNs), like U-Nets [5], because they allow pixel-wise classification. It is anticipated that these networks will enhance the accuracy of detecting and mapping wildfire-affected areas, leveraging the detailed ground truth masks included within the CWGID.

## References

[1] International Union for Conservation of Nature, "Forests and Climate Change." [Online]. Available: https://www.iucn.org/resources/issues-brief/forests-and-climate-change

[2] R. Massey, L. T. Berner, A. C. Foster, S. J. Goetz, and U. Vepakomma, "Remote Sensing Tools for Monitoring Forests and Tracking Their Dynamics," in *Boreal Forests in the Face of Climate Change: Sustainable Management*, M. M. Girona, H. Morin, S. Gauthier, and Y. Bergeron, Eds., Cham: Springer International Publishing, 2023, pp. 637–655. doi: 10.1007/978-3-031-15988-6_26.

[3] H. Jiang *et al.*, "A Survey on Deep Learning-Based Change Detection from High-Resolution Remote Sensing Images," *Remote Sensing*, vol. 14, no. 7, 2022, doi: 10.3390/rs14071552.

[4] Y. Lecun and Y. Bengio, "Convolutional Networks for Images, Speech, and Time-Series," in *The Handbook of Brain Theory and Neural Networks*, 1995.

[5] O. Ronneberger, P. Fischer, and T. Brox, “U-Net: Convolutional Networks for Biomedical Image Segmentation,” *CoRR*, 2015, doi: 10.1007/978-3-319-24574-4_28.

[6] E. J. Parelius, “A Review of Deep-Learning Methods for Change Detection in Multispectral Remote Sensing Images,” *Remote Sensing*, vol. 15, no. 8, 2023, doi: 10.3390/rs15082092.

[7] A. M. Al-Dabbagh and M. Ilyas, “Uni-temporal Sentinel-2 imagery for wildfire detection using deep learning semantic segmentation models,” *Geomatics, Natural Hazards and Risk*, vol. 14, no. 1, 2023, doi: 10.1080/19475705.2023.2196370.

[8] L. Alzubaidi *et al.*, “Review of deep learning: concepts, CNN architectures, challenges, applications, future directions,” *Journal of Big Data*, vol. 8, 2021, doi: 10.1186/s40537-021-00444-8.

[9] A. Adegun, S. Viriri, and J. Tapamo, “Review of deep learning methods for remote sensing satellite images classification: experimental survey and comparative analysis,” *Journal of Big Data*, vol. 10, p. 93, 2023, doi: 10.1186/s40537-023-00772-x.

[10] N. Gorelick, M. Hancher, M. Dixon, S. Ilyushchenko, D. Thau, and R. Moore, “Google Earth Engine: Planetary-scale geospatial analysis for everyone,” *Remote Sensing of Environment*, 2017, doi: 10.1016/j.rse.2017.06.031.

[11] M. Abadi *et al.*, “TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems.” [Online]. Available: https://www.tensorflow.org/

[12] California Department of Forestry and Fire Protection, “Fire and Resource Assessment Program, Historical Fire Perimeter Data.” [Online]. Available: https://www.fire.ca.gov/what-we-do/fire-resource-assessment-program

[13] The Pandas Development Team, “pandas-dev/pandas: Pandas.” Zenodo, 2020. doi: https://doi.org/10.5281/zenodo.3509134.

[14] M. Drusch *et al.*, “Sentinel-2: ESA's Optical High-Resolution Mission for GMES Operational Services,” *Remote Sensing of Environment*, vol. 120, pp. 25–36, 2012, doi: https://doi.org/10.1016/j.rse.2011.11.026.

[15] A. D. Snow *et al.*, “PyProj: A Python interface to PROJ (cartographic projections and coordinate transformations library).” [Online]. Available: https://zenodo.org/record/8365173

[16] P. Lopez Gonzalez-Nieto *et al.*, “DESIGN AND DEVELOPMENT OF A VIRTUAL LABORATORY IN PYTHON FOR THE TEACHING OF DATA ANALYSIS AND MATHEMATICS IN GEOLOGY: GEOPY,” in *INTED2020 Proceedings*, in 14th International Technology, Education and Development Conference. Valencia, Spain, 2020, pp. 2236–2242. doi: 10.21125/inted.2020.0687.

[17] X. X. Zhu *et al.*, “Deep Learning in Remote Sensing: A Comprehensive Review and List of Resources,” *IEEE Geoscience and Remote Sensing Magazine*, vol. 5, no. 4, pp. 8–36, 2017, doi: 10.1109/MGRS.2017.2762307.

[18] K. Jordahl *et al.*, “geopandas/geopandas: v0.8.1.” [Online]. Available: https://doi.org/10.5281/zenodo.3946761

[19] Gillies, Sean, “Rasterio: Access to geospatial raster data.” [Online]. Available: https://rasterio.readthedocs.io/

[20] S. Gillies *et al.*, “Shapely: Manipulation and Analysis of Geometric Objects.” [Online]. Available: https://zenodo.org/record/7428463

[21] F. Hu, G.-S. Xia, J. Hu, and L. Zhang, “Transferring deep convolutional neural networks for the scene classification of high-resolution remote sensing imagery,” *Remote Sensing*, vol. 7, no. 11, pp. 14680–14707, 2015, doi: 10.3390/rs71114680.

[22] D. Marmanis, M. Datcu, T. Esch, and U. Stilla, “Deep learning Earth observation classification using ImageNet pretrained networks,” *IEEE Geoscience and Remote Sensing Letters*, vol. 13, no. 1, pp. 105–109, 2016, doi: 10.1109/LGRS.2015.2499239.

[23] C. Gohlke and Contributors, “Tifffile: Read and write TIFF files.” [Online]. Available: https://pypi.org/project/tifffile/%20doi%20=%20https://doi.org/10.5281/zenodo.6795861

[24] C. R. Harris *et al.*, “Array programming with NumPy,” *Nature*, vol. 585, no. 7825, pp. 357–362, 2020, doi: https://doi.org/10.1038/s41586-020-2649-2.

[25] K. Simonyan and A. Zisserman, “Very Deep Convolutional Networks for Large-Scale Image Recognition,” in *International Conference on Learning Representations*, 2015.

[26] F. Pedregosa *et al.*, “Scikit-learn: Machine Learning in Python,” *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[27] L. Buitinck *et al.*, “API design for machine learning software: experiences from the scikit-learn project,” in *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*, 2013, pp. 108–122.

[28] D. P. Kingma and J. Ba, “Adam: A method for stochastic optimization,” *arXiv preprint arXiv:1412.6980*, 2014.

[29] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. Cambridge, MA, USA: MIT Press, 2016.

[30] M. Tan and Q. V. Le, “EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks,” in *Proceedings of the 36th International Conference on Machine Learning (ICML)*, 2019, pp. 6105–6114.

[31] S. T. Seydi, M. Hasanlou, and J. Chanussot, “Burnt-Net: Wildfire burned area mapping with single post-fire Sentinel-2 data and deep learning morphological neural network,” *Ecological Indicators*, vol. 140, p. 108999, 2022, doi: https://doi.org/10.1016/j.ecolind.2022.108999.

[32] J. Xiang, Y. Xing, W. Wei, E. Yan, J. Jiang, and D. Mo, "Dynamic Detection of Forest Change in Hunan Province Based on Sentinel-2 Images and Deep Learning," *Remote Sensing*, vol. 15, no. 3, 2023, doi: 10.3390/rs15030628.

Open Access | https://doi.org/10.25080/WXRH8633

# Echostack: A flexible and scalable open-source software suite for echosounder data processing

**Wu-Jung Lee**[1], **Valentina Staneva**[2], **Landung "Don" Setiawan**[2], **Emilio Mayorga**[1], **Caesar Tuguinay**[1], **Soham Butala**[2], **Brandyn Lucca**[1], and **Dingrui Lei**[2]

[1]Applied Physics Laboratory, University of Washington, [2]eScience Institute, University of Washington

SciPy 2024

July 8 - July 14, 2024

Proceedings of the 23nd Python in Science Conference

ISSN: 2575-9752

## Abstract

Water column sonar data collected by echosounders are essential for fisheries and marine ecosystem research, enabling the detection, classification, and quantification of fish and zooplankton from many different ocean observing platforms. However, the broad usage of these data has been hindered by the lack of modular software tools that allow flexible composition of data processing workflows that incorporate powerful analytical tools in the scientific Python ecosystem. We address this gap by developing Echostack, a suite of open-source Python software packages that leverage existing distributed computing and cloud-interfacing libraries to support intuitive and scalable data access, processing, and interpretation. These tools can be used individually or orchestrated together, which we demonstrate in example use cases for a fisheries acoustic-trawl survey.

**Keywords** ocean sonar, echosounder, distributed computing, cloud computing, workflow orchestration, data standardization

## 1. Introduction

Echosounders are high-frequency sonar systems optimized for sensing fish and zooplankton in the ocean. By transmitting sound and analyzing the returning echoes, fisheries and ocean scientists use echosounders to "image" the distribution and infer the abundance of these animals in the water column [1], [2], [3] Figure 1. As a remote sensing tool, echosounders are uniquely suitable for efficient, continuous biological monitoring across time and space, especially when compared to net trawls that are labor-intensive and discrete in nature, or optical imaging techniques that are limited in range due to the strong absorption of light in water.

In recent years, echosounders have been installed widely on many ocean observing platforms Figure 1, resulting in a deluge of data accumulating at an unprecedented speed from all corners of the ocean. These extensive datasets contain crucial information that can help scientists better understand the marine ecosystems and their response to the changing climate. However, the volume of the data (100s of GBs to TBs [4]) and the complexity of the problem (e.g., how large- scale ocean processes drive changes in acoustically observed marine biota [5]) naturally call for a paradigm shift in the data analysis workflow.

It is crucial to have software tools that are developed and shared openly, scalable in response to data size and computing platforms, easily interoperable with diverse analysis tools and different types of oceanographic data, and straightforward to reproduce to facilitate iterative modeling, parameterization, and mining of the data. These requirements

**Published** Jul 10, 2024

**Correspondence to**
Wu-Jung Lee
leewj@uw.edu

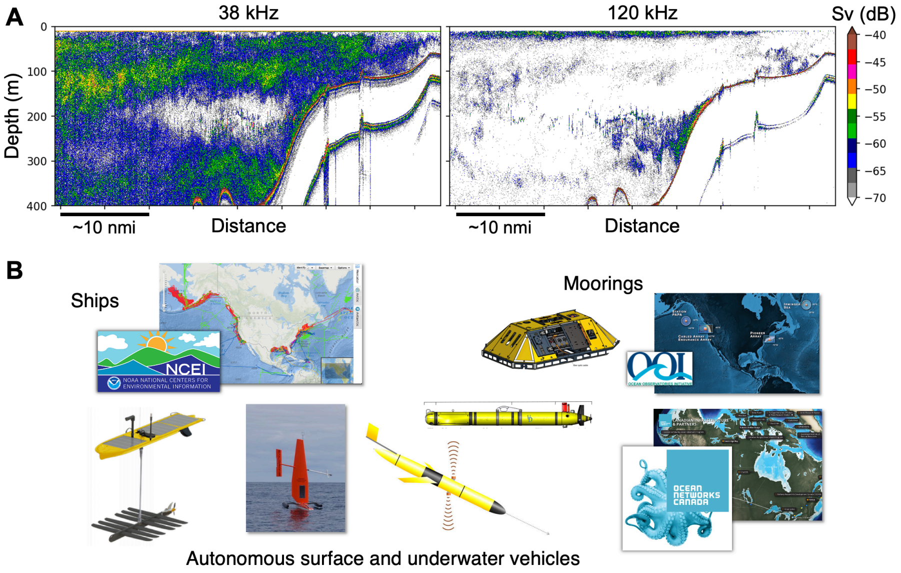


**Figure 1**. *(A) Echograms (sonar images formed by aligning echoes of consecutive transmissions) at two different frequencies. Echo strength variation across frequency is useful for inferring scatterer identity. Sv denotes volume backscattering strength (units: dB re 1 $m^{-1}$). (B) The variety of ocean observing platforms with echosounders installed.*

are challenging to meet by conventional echosounder data analysis workflows, which rely heavily on manual analysis on software packages designed to be used with Graphic User Interface (GUI) on a single computer [e.g., R. Korneliussen *et al.* [6], Y. Perrot *et al.* [7], Y. Ladroit, P. C. Escobar-Flores, A. C. G. Schimel, and R. L. O'Driscoll [8] Echoview (https://echoview.com/)]. Similarly, instead of storing data in manufacturer-specific binary formats, making echosounder data widely available in a standardized, machine-readable format will broaden their use beyond the original applications in fisheries surveys and specific research cruises.

In this paper, we introduce Echostack, an open-source Python software suite aimed at addressing these needs by providing the fisheries acoustics and ocean sciences communities with tools for flexible and scalable access, organization, processing, and visualization of echosounder data. Echostack is a domain-specific adoption of the core libraries in the Pandata stack, such as Zarr, Xarray, and Dask [9]. It streamlines the composition and execution of common echosounder data workflows, thereby allowing researchers to focus on the key interpretive stage of scientific data analysis. As echosounder data processing frequently requires combining heterogeneous data sources and models in addition to regular array operations, we designed the Echostack packages with modularity in mind, to 1) enable and facilitate broader code reuse, especially for routine processing steps that are often shared across echosounder data workflows [10], and 2) promote flexible integration of domain-specific operations with more general analytical tools, such as machine learning (ML) libraries. The Echostack also provides a friendly on-ramp for researchers who are not already familiar with the scientific Python software ecosystem but possess domain expertise and can quickly benefit from "out-of-the-box" capabilities such as native cloud access and distributed computing support.

Below, we will discuss what the Echostack tools aim to achieve in Section 2, outline the functionalities of individual libraries in Section 3, demonstrate how Echostack tools can be leveraged in Section 4, and conclude with a forward-looking discussion in the Section 5 section.

## 2. Design approach and considerations

### *2.1. Goals and motivations*

The development of the Echostack was motivated by our core goals to create software tools that can be:

1. Easily leveraged to construct workflows tailored to different application scenarios,
2. Easily integrated with new data analysis methodologies, and
3. Easily scalable according to the available computing resources and platforms.

Traditionally, the bulk of echosounder data analysis takes place on shore in a post-processing scenario, based on the acoustic data and related biological information (e.g., animal community composition and sizes derived from net catches) collected at sea and stored on hard drives. More recently, thanks to increased access to high-bandwidth communication and powerful embedded processing, scenarios requiring real-time capabilities have emerged, including continuous streaming of data from long-term moorings [11] and close-loop, adaptive sampling on autonomous vehicles [12]. Similarly on the rise are the incorporation of new inference and ML methods into echosounder data analysis workflows [13], [14], [15], [16], as well as a need for scalable and platform-agnostic computations that work out-of-the-box on systems ranging from personal computers to the cloud. The latter is particularly important for data workflows associated with ocean instruments such as echosounders, as both small and big data scenarios are common and equally important: On a sea-going research vessel, newly collected data are typically processed in small batches on a few local machines; in oceanographic data centers, however, large volumes of data need to be promptly ingested and made available for open use.

These requirements are challenging to fulfill by existing echosounder data processing software packages, but are in fact key advantages of the Pandata open-source Python stack. While some existing software packages do support real-time applications (e.g., Echoview live viewing, MOVIES3D real-time display) and allow the incorporation of new analysis methods [17], the exact code implementation may be limited by the data structures and access allowed in a given package, as opposed to the flexibility afforded by Pythonic programmatic composition. In terms of scalability, even though some existing packages can achieve out-of-core computation via memory mapping [6], [8], the natural and optional scalability across computing resources and platforms offered by Pandata libraries [9] significantly reduces the code implementation overhead.

### *2.2. Other guiding principles*

In addition to the above goals, our development was further guided by the following principles, with the long-term goal of catalyzing community collaboration and workflow transformation.

#### 2.2.1. *Accessibile and interoperable data:*

Echosounder data have traditionally been viewed as a highly specialized, niche data type that are hard to use and interpret despite the wealth of biological information embedded in the datasets. This is in part due to the highly heterogeneous, instrument-specific echosounder data format, as well as the inherent complexity in interpreting acoustic backscatter data [2]. We aim to provide convenient, open software tools that facilitate the creation of standardized data in a common, machine-readable format that is both easy to understand and conducive to integrative analysis with other oceanographic datasets.

#### 2.2.2. *Reproducible and shareable workflow:*

Just like in many scientific domains, echosounder data analysis workflows are typically built by researchers working closely within a single group, although knowledge sharing and exchanges do occur in community conferences or workshops. As data volumes rapidly increase and the urgency to comprehensively understand marine ecosystem functions escalates with global warming, efficient and effective collaboration across groups has become crucial. Ensuring that analysis workflows can be easily reproduced and shared alongside open, standardized data is key to scaling up our collective capability to extract information from these data.

#### 2.2.3. *Nimble code adjustments:*

As a domain-specific adaptation of the Pandata stack, we aim to develop and maintain a "shallow" and nimble stack: we employ only moderate layers of abstraction and encapsulation of existing functions and prioritize existing data structures unless modification is absolutely necessary. This allows new features, enhancements, or bug fixes in the underlying dependency libraries to be either naturally reflected (in cases when there are no breaking changes) or quickly incorporated (in cases when code changes are required) into the Echostack tools. This approach also keeps the code easily readable for potential contributors, even though the exact operations may be highly domain-specific.

## 3. The Echostack packages

### *3.1. Functional grouping for package organization*

In light of our goals and design principles, we aim for the Echostack libraries to be modular software "building blocks" that can be mixed and matched, not only with other elements within Echostack but also with tools from the wider scientific Python software ecosystem. Rather than encapsulating all functionalities into a single, monolithic package like many existing echosounder data analysis software [e.g., R. Korneliussen *et al.* [6] Echoview], the Echostack packages are organized based on functional groupings that emerge in typical echosounder data analysis workflows to enable flexible and selective installation and usage. Similar to the design concept of the Pandata tools, such grouping also allows each package in the Echostack to be independently developed and maintained according to natural variations of project focus and needs.

Typical echosounder data analysis workflows consist of the following common steps Figure 2:

1. Read, align, and aggregate the acoustic data and associated biological data (the "ground truth" from *in situ* samples) based on temporal and/or spatial relationships;
2. Infer the contribution of animals in different taxonomic groups to the total returned echo energy; and
3. Derive distributions of population estimates (e.g., abundance or biomass) and behavior (e.g., migration) over space and time for the observed animals.

The resulting biological estimates are then used in downstream fisheries and marine ecological research and for informing resource management decisions. Also embedded in these steps are the needs for:

- Interactive visualization of echo data and/or derived estimates, at different stages of the workflow, which is crucial for understanding data and motivating downstream analyses; and
- Robust provenance tracking of data origin and processing steps, which enhances the reproducibility and shareability of the workflows.

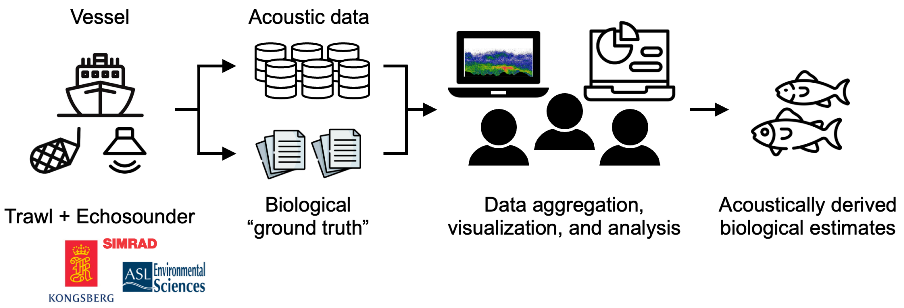


**Figure 2.** *Typical echosounder data analysis workflows.*

In a fisheries acoustic-trawl survey – a typical use case – the overall goal is to estimate the abundance and biomass of important fish and zooplankton species using a combination of observational data and analytical models. The observational data include acoustic data collected by the echosounders and biological data (animal species and sizes, etc) collected by trawls. These data are linked in the acoustic inference step, which partitions the acoustic observation into contributions from different animal groups in the water column based on empirical models (which rely heavily on morphological features of animal aggregations visualized on the echogram), physics-based models (which rely heavily on spectral or statistical features of the echo returns), ML models (which can combine various data features depending on the model setup), or a mixture of the above.

Based on the above, we identified natural punctuations in the typical workflows, and grouped computational operations that serve coherent high-level functionalities into separate packages Figure 3 that are described in the next section. The grouping decisions and the current implementations were driven by our own project goals to accelerate information extraction for a fisheries acoustic-trawl survey for Pacific hake (see Section 4 for detail) through ML and cloud technology. However, as the examples will show, these packages can be easily leveraged in different application scenarios.

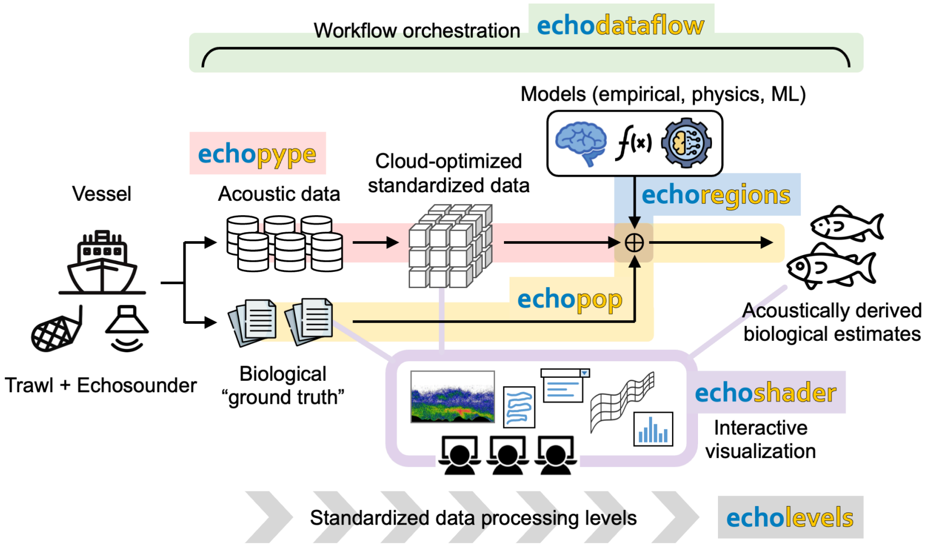
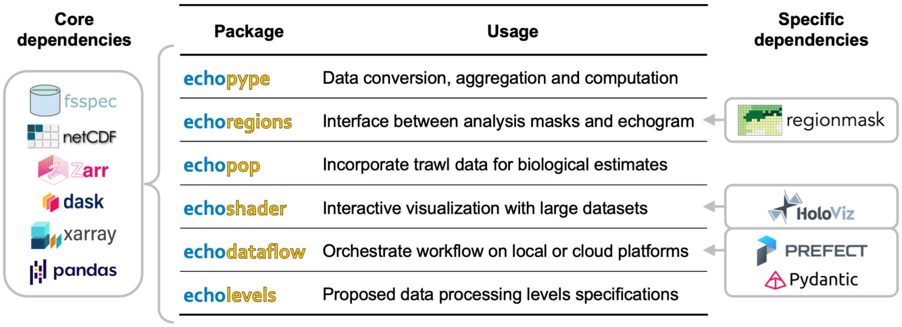


**Figure 3.** *The Echostack packages are modularized based on functional grouping of computational operations in echosounder data analysis workflows.*



| Package | Usage |
|---|---|
| echopype | Data conversion, aggregation and computation |
| echoregions | Interface between analysis masks and echogram |
| echopop | Incorporate trawl data for biological estimates |
| echoshader | Interactive visualization with large datasets |
| echodataflow | Orchestrate workflow on local or cloud platforms |
| echolevels | Proposed data processing levels specifications |

**Figure 4**. *Summary of the Echostack packages and key dependencies.*

## 3.2. *Package functionalities*

The Echostack includes five software packages (Echopype, Echoregions, Echopop, Echoshader, and Echodataflow) that each handles a specific set of functionalities in general echosounder data processing workflows Figure 3. In parallel, Echolevels provides specifications for a cascade of data processing levels from raw to highly processed data, to enhance data sharing and usage. The Echostack packages jointly relies on a set of core dependencies for interfacing with file systems (fsspec), data storage (netCDF and Zarr), and data organization and computation (Xarray, Pandas, and Dask). Additionally, some libraries have specific dependencies to support functionalities like interactive plotting (HoloViz), masking grids based on regions (Regionmask), and workflow configuration (Pydantic) and orchestration (Prefect), as shown in Figure 4.

The sections below provide brief descriptions of the functionalities of the Echostack packages, their current status, and future development plans.

### 3.2.1. *Echopype:*

Echopype (https://github.com/OSOceanAcoustics/echopype) is the workhorse that performs the heavy lifting of data standardization and routine computation to generate data products that are suitable for downstream analysis. Acoustic (backscatter) data collected by echosounders are typically stored in instrument-specific binary files along with metadata, such as deployment configurations. The raw data are often recorded in sensor units (e.g., counts) that require calibration to be mapped to physically meaningful quantities (e.g., sound pressure). Analyses are typically performed on aggregated data sections based on spatial and/or temporal logical groupings. For example, ship echosounder data are often grouped based on survey transects, and mooring echosounder data are often grouped into weekly or monthly sections. In addition, many analysis methods require data to be uniformly gridded across all major dimensions, such as time, geographic locations (latitude/longitude), and depth. Echopype provides functionalities for converting raw, binary data files into a standardized representation following the netCDF data model, which can be serialized into netCDF or Zarr files. It also provides functionalities for calibration, quality control, aggregating data across files, and data regridding. It forms the data churning backbone in Echostack-supported workflows, and encodes provenance and processing level information as specified in Echolevels (see below). Echopype was the first package developed in the Echostack and is described in detail in W.-J. Lee, E. Mayorga, L. Setiawan, I. Majeed, K. Nguyen, and V. Staneva [18].

### 3.2.2. *Echoregions:*

Echoregions (https://github.com/OSOceanAcoustics/echoregions) is a lightweight package that interfaces data products generated by Echopype, such as echograms, with annotations from humans or data interpretation algorithms, such as ML algorithms. These annotations typically consist of “regions” that indicate the presence of specific animal species, or “lines” that delineate ocean boundaries, such as seafloor or sea surface. As a lightweight interfacing package, Echoregions makes it possible to construct independent workflow branches that handle data interpretation annotations in concert with the data processing backbone and the analysis algorithms. Currently, due to our project priorities, Echoregions primarily supports the organization and generation of echogram analysis “masks” in ML problems [19] with bi-directional interfacing functionalities: to parse and organize annotations to prepare training and test datasets for ML developments (annotation to ML), and to generate annotations from ML predictions to be used for further downstream processing (ML to annotation). The current supported annotation formats are those generated by Echoview (.EVR and .EVL files), which is widely used in the fisheries acoustics community to manually analyze echograms. We plan to expand support for additional annotation formats and update both the representation of annotations and the generated masks to align with community conventions that may emerge in the near future.

### 3.2.3. *Echopop:*

Echopop (https://github.com/OSOceanAcoustics/echopop) functions as the “glue” that combines echo measurements with biological data from trawls and acoustic backscattering models to estimate spatial distributions of animal abundance, biomass, and other biologically meaningful metrics. Here, “pop” stands for animal “population” information that can be obtained by analyzing echo data. Biological trawl samples typically yield species compositions and associated biometrics, such as sex, length, age, and weight. Many of these parameters are crucial for accurately configuring the acoustic backscattering models, which strongly affect the estimates of biological quantities from echo data [2], which are important for informing fishery management strategies and for characterizing regional and global energy budgets. At present, the biological data agglomeration and echo data analysis steps implemented in Echopop are configured to work specifically with a fisheries acoustic-trawl survey targeting Pacific hake that is the focus of our project. However, the majority of the functions are generally applicable to other fish and zooplankton species. We therefore plan to expand the package to incorporate more general analysis functions in the near future, including support for biological data obtained by other *in-situ* ground-truthing methods, such as optical cameras [20].

### 3.2.4. *Echoshader:*

Echoshader (https://github.com/OSOceanAcoustics/echoshader/) enables interactive visualization of echosounder data to accelerate the data exploration and discovery process. Here, “shader” was inspired by Datashader, as echo data are often plotted as false color panels encoded with magnitude information (e.g., Figure 1). Building on the HoloViz suite of tools, Echoshader extends Xarray’s plotting functionality using accessors (https://docs.xarray.dev/en/latest/internals/extending-xarray.html) to include variations that are widely used and insightful for exploring and understanding echosounder data. For example, the absolute and relative echo magnitudes measured at different frequencies can be used to classify scatterers, and are typically visualized through joint displays of echograms at multiple frequencies. Echograms are water column “profiles” that are often more meaningful when inspected together with the ship track on a bathymetry map. Echoshader makes it possible to create these plots directly from echosounder datasets generated by Echopype (which are native Xarray datasets or data arrays), complete with control widgets such as frequency selectors and color range sliders. In the next stage of development, we plan to integrate external echogram region annotations (e.g., from Echoregions) into the functionality, optimize performance for visualizing large datasets to enable efficient screening of historical data, and compile a gallery of demo use cases as part of the documentation.

### 3.2.5. *Echodataflow:*

Echodataflow (https://github.com/OSOceanAcoustics/echodataflow) is a package that provides users with an efficient way to define, configure, and execute complex echosounder data processing workflows on either local or cloud platforms. It combines the power of Prefect (https://www.prefect.io/), an open-source workflow orchestration tool, and the flexibility of YAML configurations to leverage Echostack data processing functionalities in a modular and user-friendly way. Instead of writing code, users prepare "recipes" that are text-based configurations specifying data processing steps and associated parameters, data sources and output storage locations, logging options, and computing resources. Given a recipe, Echodataflow automatically translates these specifications into actions and augments the processes with the robust error tracking and retry mechanisms from Prefect. This makes it possible to systematically prototype and experiment with workflow variations and efficiently transition a prototype to a stable data pipeline, improving the collaboration between researchers, data engineers, and data managers. Currently, due to our project priorities, Echodataflow supports Echopype and Echoregions functionalities as well as custom functions that perform ML predictions and cloud data upload. We plan to create functional templates for users to easily "wrap" their own workflow elements to use with Echodataflow in the near future.

### 3.2.6. *Echolevels:*

Echolevels (https://github.com/OSOceanAcoustics/echolevels) is a set of proposed data processing level definitions for echosounder data, rather than a software package. A concept originated and championed by the satellite remote sensing community [21], "data processing levels" are categorizations of the continuum of data at different processing stages, from raw instrument-generated files to derived, often gridded and georeferenced data products that are ready for analysis or interpretation. Similar to data conventions, clear processing level definitions enhance data understanding and provenance tracking, which contribute directly to broader data use. We have added the proposed processing levels as dataset attributes in the output data of different Echopype functions to test these definitions in practice, and are collaborating with the echosounder user community to revise and improve them.

## 4. Example use cases

In this section, we present three example use cases of the Echostack libraries under the application scenario of a fisheries acoustic-trawl survey. Here, we use the Joint U.S. and Canada Pacific Hake Integrated Ecosystem and Pacific Hake Acoustic Trawl Survey (aka the "Hake survey") as an example, as we are familiar with its context and setup for our project to accelerate biological information extraction using ML and cloud technology. The Hake survey dates back to the late 1990s when echosounder technology was first widely adopted for fishery surveys. In recent years, this survey has occurred every odd-numbered year, covering the coastal waters, shelf break, and open ocean along the entire west coasts of the US and Canada [22].

The example use cases presented below demonstrate how Echostack tools are leveraged in echosounder data analysis and management. The examples utilize both raw data files archived in a data center and near real-time data continuously generated from a ship-mounted echosounder. We show how these tools can be used in a mix-and-match manner with each other and with other tools in the scientific Python software ecosystem to achieve user goals. Note that not all Echostack libraries are used in all use cases, which highlights the advantage of modular package design based on functional grouping.

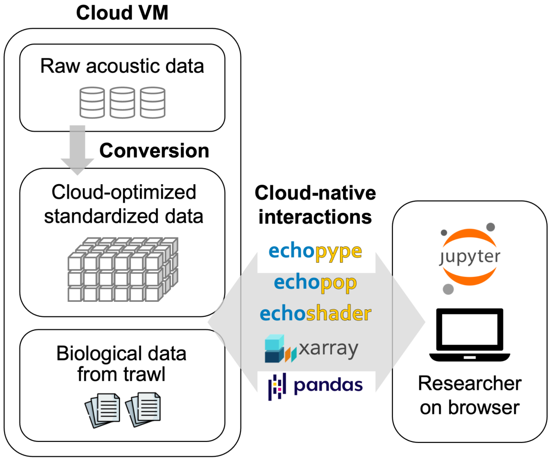


**Figure 5.** *Use case 1: Interactive data exploration and experimentation.*

### 4.1. *Use case 1: Interactive data exploration and experimentation*

This is a common use case in which researchers interact with echosounder data through the browser-based Jupyter computing interface on a cloud virtual machine Figure 5. They first assemble a Zarr store consisting of calibrated echo data for a single survey transect by directly processing a collection of raw files from a data center server (Echopype). They then interactively visualize the data (Echoshader) and experiment with the influence of different data cleaning options (Echopype and custom functions) on the resulting acoustically inferred biomass of a fish species (Echopop). These experimentations help the research identify parameters for setting up batch processing across all survey transects.

### 4.2. *Use case 2: A ship-to-cloud ML pipeline*

This is an experimental use case in which researchers construct a ship-to-cloud pipeline that uses an ML model to analyze ship echosounder data in near real-time and send the predictions and reduced data to the cloud for shore-side monitoring Figure 6. The pipeline is orchestrated by Echodataflow, which manages data organization actions triggered by new files and analysis-oriented actions through a scheduler. A "watchdog" monitors the ship hard drive for newly generated raw echosounder data files, and continuously converts, calibrates, regrids, and appends new echo data into a local Zarr store dedicated to ML analysis (Echopype). At regular intervals, the last section in the Zarr store is sliced and fed to an ML model to detect the occurrence of a target fish species on the echogram (custom functions). The echo data are applied with the ML predictions (Echoregions) and used to compute the abundance and biomass estimates (Echopop). The echo data slices are also further downsampled (Echopype) and sent to a cloud Zarr store for interactive visual monitoring (Echoshader). This pipeline paves the way for later optimization for an edge implementation on autonomous platforms.

### 4.3. *Use case 3: Mass production of analysis-ready, cloud-optimized (ARCO) data products*

This is a use case in which researchers or data managers create analysis-ready, cloud-optimized (ARCO) echosounder data stores that can be readily drawn for analysis and experimentation Figure 7. The pipeline is orchestrated by Echodataflow, which handles the efficient distribution of bulk computing tasks, communicates with data sources and destinations, and automatically logs and retries any processing errors. In the pipeline, instrument-generated binary data files sourced from a data center progress through multiple

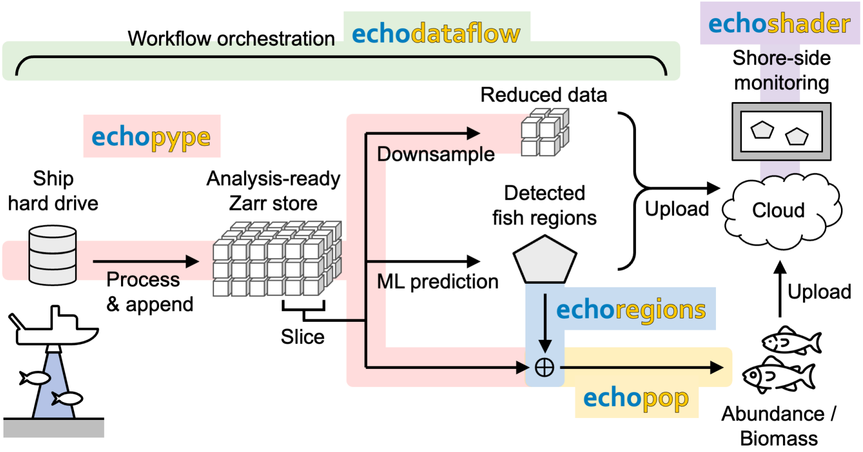


**Figure 6**. *Use case 2: A ship-to-cloud ML pipeline.*

processing steps and are enriched with auxiliary data and metadata that may not exist in the original binary form (Echopype). These steps span through multiple data processing levels (Echolevels), at which data products are stored with full provenance to maximize reprocessing flexibility and usability. The data stores assembled under this use case can be easily leveraged by researchers in Use case 1 to jump start their experimentation of data cleaning parameters.

## 5. Discussion

Echostack is an open-source software suite under active development. It will continue to evolve with the needs of the users and advancements in the scientific Python software ecosystem, in particular the Pandata and derived libraries in the wider geosciences domain. The Echostack tools were initially developed to address needs arising from our own research projects, but we quickly realized the potential value of these tools as open resources in the fisheries acoustic and ocean sciences communities. This paper serves as an interim report that describes our design considerations and discusses current capabilities and future plans together with three concrete use cases. Moving forward, we have two main goals: 1) enhancing Echostack features and performance, and 2) cultivating community user engagement. For the libraries, we aim to incorporate a wider set of echosounder data processing and visualization functions, ensure scalability with large datasets, improve documentation, and provide benchmark reports for library performance. For the community, we aim to broaden the discussion on development priorities and encourage code

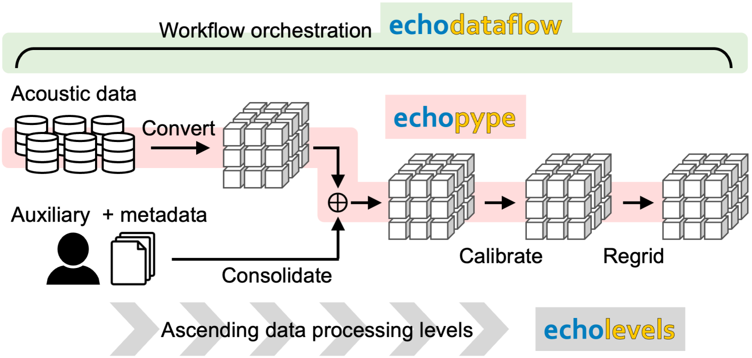


**Figure 7**. *Use case 3: Mass production of cloud-optimized, analysis-ready data products*

contribution from the user community, through intentional outreach efforts via both formal (conferences and workshops) and informal (interactions on code repositories and emails) channels. These goals are interdependent and form a positive feedback loop, as the more useful the software tools are, the more likely people will use it and contribute to its maintenance and development. The echosounder user community is diverse and includes members with a wide range of technical experiences and practical considerations, and would benefit from a suite of flexible tools that are both plug-and-play and customizable to meet specific requirements. It is our hope that the Echostack software packages can function as a catalyst for community-wide discussions and collaborations on advancing data processing workflow and information extraction in this domain.

## Acknowledgements

We thank Alicia Billings, Dezhang Chu, Julia Clemons, Elizabeth Phillips, and Rebecca Thomas for discussions on code implementations and use cases, and Imran Majeed, Kavin Nguyen, Praneeth Ratna, and Brandyn Reyes for contributing to the code. The software development received funding support from NOAA Fisheries, NOAA Ocean Exploration, NSF, NASA, and the Gordon and Betty Moore and Alfred P. Sloan Foundations (MSDSE Environments), as well as engineering support from the University of Washington Scientific Software Engineering Center funded by Schmidt Futures as part of the Virtual Institute for Scientific Software.

## References

[1] H. Medwin and C. S. Clay, *Fundamentals of acoustical oceanography*. Academic Press, 1998. doi: 10.1016/B978-0-12-487570-8.X5000-4.

[2] T. K. Stanton, “30 years of advances in active bioacoustics: A personal perspective,” *Methods in Oceanography*, pp. 49–77, 2012, doi: 10.1016/j.mio.2012.07.002.

[3] K. J. Benoit-Bird and G. L. Lawson, “Ecological insights from pelagic habitats acquired using active acoustic techniques,” *Annual Review of Marine Science*, vol. 8, no. 1, pp. 463–490, 2016, doi: 10.1146/annurev-marine-122414-034001.

[4] C. C. Wall, J. M. Jech, and S. J. McLean, “Increasing the accessibility of acoustic data through global access and imagery,” *ICES Journal of Marine Science: Journal du Conseil*, vol. 73, no. 8, pp. 2093–2103, 2016, doi: 10.1093/icesjms/fsw014.

[5] T. A. Klevjer, X. Irigoien, A. R̸stad, E. Fraile-Nuez, V. M. Benítez-Barrios, and S. Kaartvedt., “Large scale patterns in vertical distribution and behaviour of mesopelagic scattering layers,” *Scientific Reports*, vol. 6, no. 1, p. 19873, 2016, doi: 10.1038/srep19873.

[6] R. Korneliussen *et al.*, “The Large Scale Survey System - LSSS,” in *Proceedings of the 29th Scandinavian Symposium on Physical Acoustics*, Ustaoset, Norway, 2006. [Online]. Available: https://api.semanticscholar.org/CorpusID:204802910

[7] Y. Perrot *et al.*, “Matecho: An Open-Source Tool for Processing Fisheries Acoustics Data,” *Acoustics Australia*, vol. 46, no. 2, pp. 241–248, 2018, doi: 10.1007/s40857-018-0135-x.

[8] Y. Ladroit, P. C. Escobar-Flores, A. C. G. Schimel, and R. L. O’Driscoll, “ESP3: An open-source software for the quantitative processing of hydro-acoustic data,” *SoftwareX*, vol. 12, p. 100581, 2020, doi: 10.1016/j.softx.2020.100581.

[9] J. A. Bednar and M. Durant, “The Pandata Scalable Open-Source Analysis Stack,” *Proceedings of the 22nd Python in Science Conference*, pp. 85–92, 2023, doi: 10.25080/gerudo-f2bc6f59-00b.

[10] K. Haris, R. J. Kloser, T. E. Ryan, R. A. Downie, G. Keith, and A. W. Nau, “Sounding out life in the deep using acoustic data from ships of opportunity,” *Scientific Data*, vol. 8, no. 1, p. 23, 2021, doi: 10.1038/s41597-020-00785-8.

[11] J. Trowbridge *et al.*, “The Ocean Observatories Initiative,” *Frontiers in Marine Science*, vol. 6, 2019, doi: 10.3389/fmars.2019.00074.

[12] M. A. Moline, K. Benoit-Bird, D. O'Gorman, and I. C. Robbins, “Integration of scientific echo sounders with an adaptable autonomous vehicle to extend our understanding of animals from the surface to the bathypelagic,” *Journal of Atmospheric and Oceanic Technology*, vol. 32, no. 11, pp. 2173–2186, 2015, doi: 10.1175/JTECH-D-15-0035.1.

[13] O. Brautaset *et al.*, “Acoustic classification in multifrequency echosounder data using deep convolutional neural networks,” *ICES Journal of Marine Science*, vol. 77, no. 4, pp. 1391–1400, 2020, doi: 10.1093/ICESJMS/FSZ235.

[14] C. Choi, M. Kampffmeyer, N. O. Handegard, A.-B̸. Salberg, and R. Jenssen, “Deep semisupervised semantic segmentation in multifrequency echosounder data,” *IEEE Journal of Oceanic Engineering*, pp. 1–17, 2023, doi: 10.1109/JOE.2022.3226214.

[15] S. S. Urmy, A. De Robertis, and C. Bassett, “A Bayesian inverse approach to identify and quantify organisms from fisheries acoustic data,” *ICES Journal of Marine Science*, p. fsad102, 2023, doi: 10.1093/icesjms/fsad102.

[16] Y. Zhang, C. C. Wall, J. M. Jech, and Q. Lv, “Developing a hybrid model with multiview learning for acoustic classification of Atlantic herring schools,” *Limnology and Oceanography: Methods*, vol. 22, no. 5, pp. 351–368, 2024, doi: 10.1002/lom3.10611.

[17] L.-M. K. Harrison, M. J. Cox, G. Skaret, and R. Harcourt, “The R package EchoviewR for automated processing of active acoustic data using Echoview,” *Frontiers in Marine Science*, vol. 2, 2015, doi: 10.3389/fmars.2015.00015.

[18] W.-J. Lee, E. Mayorga, L. Setiawan, I. Majeed, K. Nguyen, and V. Staneva, “Echopype: A Python library for interoperable and scalable processing of water column sonar data for biological information,” *arXiv:2111.00187 [eess]*, 2021, doi: 10.48550/arXiv.2111.00187.

[19] M. Hauser, A. Spring, J. Busecke, M. v. Driel, R. Lorenz, and readthedocs assistant, “regionmask/regionmask: version 0.12.1.” [Online]. Available: https://zenodo.org/records/10849860

[20] K. Williams, N. Lauffenburger, M.-C. Chuang, J.-N. Hwang, and R. Towler, “Automated measurements of fish within a trawl using stereo images from a Camera-Trawl device (CamTrawl),” *Methods in Oceanography*, vol. 17, pp. 138–152, 2016, doi: 10.1016/j.mio.2016.09.008.

[21] R. Weaver, “Processing Levels,” in *Encyclopedia of Remote Sensing*, E. G. Njoku, Ed., in Encyclopedia of Earth Sciences Series., New York, NY: Springer, 2014, pp. 517–520. doi: 10.1007/978-0-387-36699-9_36.

[22] Northwest Fisheries Science Center, Fishery Resource Analysis and Monitoring Division, “The 2021 Joint U.S.-Canada Integrated Ecosystem and Pacific Hake Acoustic Trawl Survey: Cruise Report SH-21-06,” 2022, doi: 10.25923/0979-6D84.